Deep Priority Conflicts in the Wild: A Pilot Study

Luís Eduardo de Souza Amorim
Delft University of Technology
The Netherlands
l.e.desouzaamorim-1@tudelft.nl

Michael J. Steindorfer
Delft University of Technology
The Netherlands
m.j.steindorfer@tudelft.nl

Eelco Visser
Delft University of Technology
The Netherlands
e.visser@tudelft.nl

Abstract

Context-free grammars are suitable for formalizing the syntax of programming languages concisely and declaratively. Thus, such grammars are often found in reference manuals of programming languages, and used in language workbenches for language prototyping. However, the natural and concise way of writing a context-free grammar is often ambiguous.

Safe and complete declarative disambiguation of operator precedence and associativity conflicts guarantees that all ambiguities arising from combining the operators of the language are resolved. Ambiguities can occur due to *shallow conflicts*, which can be captured by one-level tree patterns, and *deep conflicts*, which require more elaborate techniques. Approaches to solve deep priority conflicts include grammar transformations, which may result in large unambiguous grammars, or may require adapted parser technologies to include data-dependency tracking at parse time.

In this paper we study deep priority conflicts *"in the wild"*. We investigate the efficiency of grammar transformations to solve deep priority conflicts by using a lazy parse table generation technique. On top of lazily-generated parse tables, we define metrics, aiming to answer how often deep priority conflicts occur in real-world programs and to what extent programmers explicitly disambiguate programs themselves. By applying our metrics to a small corpus of popular open-source repositories we found that in OCaml, up to 17% of the source files contain deep priority conflicts.

CCS Concepts • **Software and its engineering → Syntax**; *Parsers*;

Keywords Disambiguation, operator precedence, declarative syntax definition, grammars, empirical study.

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136020

ACM Reference Format:
Luís Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. 2017. Deep Priority Conflicts in the Wild: A Pilot Study. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136020

1 Introduction

In software engineering, the Don't Repeat Yourself (DRY) principle means that *"every piece of knowledge must have a single, unambiguous, authoritative representation within a system"* [11]. While in theory context-free grammars come close to fulfilling this principle for declaratively formalizing the syntax of a programming language, they still fail to deliver it in practice [13].

Natural and concise ways of writing a context-free grammar are often ambiguous and lead to Write Everything Twice (WET) solutions, i.e., the direct opposite of DRY. For example, the reference manual of the Java SE 7 edition [6] contains a natural and concise context-free reference grammar that describes the language, but a different grammar is used as the basis for the reference implementation. The refined Java SE 8 specification [7] contains a single unambiguous grammar, at the price of losing conciseness and readability.

A long-standing research topic in the parsing community is how to declaratively disambiguate concise expression grammars of programming languages. To address this issue, formalisms such as YACC [12] or SDF2 [22] extend context-free grammars with precedence and associativity declarations. In YACC, precedence is defined by a global ranking on the tokens of operators, and interpreted as choosing an alternative that solves a conflict in a parse table (i.e., a conflict should be resolved in favor of a specific action given a certain lookahead token). SDF2, on the other hand, constructs a partial order among productions using priority relations, deriving filters that reject conflicting patterns from the resulting tree. Because it supports the full class of context-free grammars and character-level grammars to enable modular syntax definitions and language composition, the YACC solution cannot be applied, which poses additional challenges when developing a solution to disambiguate SDF2 grammars.

Two desired properties for declarative disambiguation of precedence and associativity conflicts using SDF2 priorities are *safety* and *completeness*. To strive towards safety and

completeness, recent proposals rely either on grammar-to-grammar transformation techniques [3, 5], or they rely on data-dependent formalisms [2] that deflect performance overhead to run-time. On one hand, grammar-to-grammar transformations have the advantage to output (well-researched) pure context-free grammars. On the other hand, they blow up the resulting grammar by extensively duplicating productions,[1] which may result in large LR parse tables.

In this paper, we aim to investigate the efficiency and usefulness of grammar-to-grammar transformations for solving (deep) priority conflicts. We address the *efficiency* issue by inspecting how much of the resulting grammars are used and respectively, how much of their parse tables are exercised. To reason about *usefulness*, we investigate to what extent deep priority conflicts occur in real code and whether conflicts are explicitly disambiguated, looking into declarative disambiguation from the programmers' perspective. In particular we empirically address the following research questions concerning coding practices:

RQ1 To what extent do deep priority conflicts in declarative language specifications occur in real-world programs?

RQ2 How do deep priority conflicts impact the efficiency of declarative disambiguation techniques that rely on grammar transformations?

RQ3 To what extent do programmers use brackets for disambiguation of priority conflicts explicitly?

We study the aforementioned research questions for declarative context-free grammars of two programming languages — OCaml and Java — that inherently feature deep priority conflicts. In a pilot study, we empirically examine the top 10 trending open-source projects of each language on GitHub.

Contributions. We performed an empirical pilot study that investigates deep priority conflicts *"in the wild"*. In particular:

- We contribute a research method for measuring deep priority conflicts and for obtaining coverage metrics.
- We provide initial results on the frequency and circumstances under which deep priority conflicts occur.
- We present insights about explicit disambiguation of deep and shallow priority conflicts using brackets.

With our pilot study, we provide an indication on how much deep priority conflicts are an issue when parsing real-world code. We investigate the causes of deep priority conflicts, and provide guidance for subsequent studies.

The remainder of this paper is organized as follows. In Section 2 we provide background on (deep) priority conflicts and declarative disambiguation. Section 3 develops the research method necessary to empirically reason about deep priority conflicts. Section 4 presents the results of our empirical pilot study. We discuss threats to validity in Section 5. Finally, we present related work in Section 6, before concluding.

[1] Grammar transformations create copies of original productions, modifying only specific non-terminals.

2 A Primer on Declarative Disambiguation

Safe and complete disambiguation is a precondition for precisely reasoning about deep priority conflicts. Thus, we discuss the necessary background on the nature of (deep) priority conflicts, declarative disambiguation of such conflicts, and explain associated safety and completeness properties.

What is a priority conflict? Context-free grammars allow to declaratively and concisely define the syntax of a programming language. E.g., the following example defines productions rules for an expression grammar supporting integer literals, addition and multiplication:

```
Exp.Add = Exp "+" Exp
Exp.Mul = Exp "*" Exp
Exp.Int = INT
```

Although this grammar describes the basic syntax of arithmetic expressions properly, it fails to mention that multiplication binds stronger than addition, or that such operators are left associative. As a result, the input string `1 + 2 * 3` is ambiguous, because it could be parsed or interpreted as either `(1 + 2) * 3` or `1 + (2 * 3)`. Generalized parsers [1, 21, 22] typically derive all possible derivations and capture them in so-called ambiguity nodes:

```
AmbiguityList(
  Mul(Add(Int("1"), Int("2")), Int("3")),
  Add(Int("1"), Mul(Int("2"), Int("3")))
)
```

The ambiguity node represents a variable-length list of alternative interpretations, in our case for the string `1 + 2 * 3`.

How to declaratively disambiguate priority conflicts? In order to designate unambiguous parse interpretations, reference manuals of programming languages traditionally describe precedence and associativity relationships among a language's operators in supplementary tables. Syntax definition formalisms translate these tables into declarative constructs for determining the correct parse when combining such operators [8, 12, 22]. More recent context-free grammar formalisms directly integrate precedence and associativity using associativity attributes and priority relations, such as:

```
Exp.Mul = Exp "+" Exp {left}
Exp.Mul = Exp "*" Exp {left}
Exp.Mul > Exp.Add
```

These declarations, written in SDF3 [23] syntax, specify that addition and multiplication are left associative, and that multiplication binds stronger than addition. Technically, associativity attributes and priority relations define patterns used by a parse tree filter [14] to prohibit conflicts to occur. The filter defined using the priority and the precedence attribute in the example prohibits an addition to occur as a direct child of a multiplication, or additions (multiplications) to occur as a direct rightmost child of another additions (multiplications), respectively. Even though this approach is enough to

support the operator precedence and associativity of many programming languages, it is not *safe* nor *complete* [3].

What is safe and complete disambiguation? Ideally, a parser is assumed to deterministically produce exactly one valid parse tree for any valid input string of a language. When considering concise but ambiguous grammars, declarative disambiguation shifts the responsibility of resolving ambiguities due to operator precedence and associativity from the language engineer to the parser generator. *Safe disambiguation* denotes that valid inputs strings are not rejected by the parser, i.e., if an input string belongs to the language covered by the grammar, then the parser should produce at least one tree. *Complete disambiguation* states that the declarative priority relations specified together with the grammar can disambiguate all combinations of operators, i.e., for any input constructed combining the operators from the grammar, at most one tree is produced.

What is a deep priority conflict? Most priority conflicts can be solved by looking at the direct expansions of the symbols within a production, ruling out trees containing invalid patterns. Such conflicts will henceforth be called *shallow* priority conflicts. The previous example, of multiplication binding stronger than addition, is such a shallow conflict, as the priority relation states that an addition cannot be a direct child of a multiplication. In contrast, conflicts that cannot be solved via parent-child relations of productions are henceforth referred to as *deep* priority conflicts. Deep priority conflicts can occur arbitrarily nested due to indirections (e.g., intermediate productions) that hide directly conflicting productions. In the following, we will discuss the three types of deep priority conflicts by example.[2]

Deep Priority Conflict #1: Operator-Style. To illustrate deep conflicts with operator precedence,[3] we add if-else-expressions to our example expression grammar:

```
Exp.IfElse = "if" Exp "then" Exp "else" Exp
Exp.Add > Exp.IfElse
```

The declarative disambiguation rule on the last line specifies that addition binds stronger than if-else-expressions. Yet, parsing the string `1 + if e then 2 else 3 + 4` could produce two different interpretations:

```
1 + if e then 2 else (3 + 4)
(1 + if e then 2 else 3) + 4
```

If we consider disambiguation of shallow conflicts, the priority declaration `Exp.Add > Exp.IfElse` states that an if-else-expression cannot occur as a direct child of an addition. Safe disambiguation guarantees that an if-else-expression can still occur as the right child of an addition, i.e., the string `1 + if e then 2 else 3` should not be rejected as it is unambiguously accepted by the grammar. In general, operator-style conflicts may occur whenever nesting prefix operators and post-fix operators[4] of different precedences, and their precedence cannot be checked with a parent-child relation due to indirections. In our example, a deep conflict occurs because the if-else-expression can still occur as left child of the addition, hidden by another addition. When writing `Exp.Add > Exp.IfElse`, one would like to indicate that any addition to the right of the if-else-expression should always have higher precedence. That is, the correct interpretation should be only the first one: `1 + if e then 2 else (3 + 4)`.

Deep Priority Conflict #2: Dangling Else. To illustrate dangling-else conflicts, we add if-expressions without else-branches to our running example, the expression grammar:[5]

```
Exp.If     = "if" Exp "then" Exp
Exp.IfElse = "if" Exp "then" Exp "else" Exp
Exp.IfElse > Exp.If
```

The dangling-else conflict, which is present in grammars of many programming languages, arises when an if-expression is nested inside an if-clause of another if-else-expression, such as in the string `if e1 then if e2 then 3 else 4`. This input string results in two possible parses:

```
if e1 then (if e2 then 3 else 4)
if e1 then (if e2 then 3) else 4
```

The else-branch could be either connected to the first or the second if-expression. The root cause of the dangling-else conflict is that two productions of the same non-terminal share a common prefix, with the smaller production being right-recursive. The disambiguation rule `Exp.IfElse > Exp.If` indicates that an else-branch must be connected to the closest if-expression (cf. first interpretation). Note that even though the ambiguity above could be solved as a shallow conflict, dangling else conflicts are also deep conflicts, as the inner if-expression could be nested inside another expression.

Deep Priority Conflict #3: Longest Match. Longest match conflicts are caused when nesting lists of the same symbols inside each other. For example, consider the built-in match-expression of a language such as OCaml that has the following form:

```
Exp.Match    = "match" Exp "with" Pattern+
Pattern.Case = "|" Pattern "->" Exp
```

An ambiguity arises when a case-clause of a match-expression has an inner match-expression with multiple case-clauses.

[2]The formalization of deep priority conflicts [5] includes a description of the symmetric versions of the conflicts presented in this paper.

[3]The notion of *operators* has been extended to sentential forms in recursive productions. E.g., `"if" Exp "then" Exp "else"` is a prefix operator in a production `Exp.IfElse = "if" Exp "then" Exp "else" Exp`.

[4]Infix operators are considered both as prefix, and post-fix. A deep priority conflict does not occur between two infix operators, since in this case, the conflict can be disambiguated with filters based on one-level tree patterns.

[5]The if-else-expression was copied from the previous listing to emphasize that both if-variants share the same prefix.

In that situation, the parser cannot distinguish to which expression the subsequent case-clauses are connected to:

```
match value with
| pattern -> result
| pattern -> match value with
             | pattern -> result
             | pattern -> result

match value with
| pattern -> result
| pattern -> match value with
             | pattern -> result
| pattern -> result
```

The standard disambiguation of this conflict consists of expanding the list of case-clauses of the inner match-expression as much as possible, i.e., producing the longest match. Thus, in the previous example the first interpretation is correct.

3 Reasoning about Deep Priority Conflicts

In the previous section, we presented examples of three common types of (deep) priority conflicts that may arise in declarative context-free grammar specifications. This section incrementally introduces a method that we subsequently use to measure and analyze declarative disambiguation of deep priority conflicts in practice.

Although declarative disambiguation is widely used in syntax definition formalisms such as SDF, only recently, shortcomings concerning safe and complete disambiguation were reported [3]. That raises the question, why those shortcomings remained undetected for more than a decade? If and to what extent are deep priority conflicts indeed an issue in real-world grammars and programs? Not much is known about deep priority conflicts *in the wild*. In Section 3.1 we will discuss *contextual grammars* to quantify RQ1: *To what extent do deep priority conflicts in declarative language specifications occur in real-world programs?*

One may question the usefulness of declarative disambiguation techniques, as no research has been performed into deep priority conflicts in real-world settings. In particular, solving deep priority conflicts with grammar transformations has a cost attached, potentially resulting in large context-free grammars with many productions that are not used in practice. In Section 3.2 we will discuss *lazy parse table generation* to quantify RQ2: *How do deep priority conflicts impact the efficiency of declarative disambiguation techniques that rely on grammar transformations?*

Aside from the technical limitation, deep priority conflicts that are inherent to grammars of programming languages may impact common programming practice. One may ask if programmers need to be aware of the notion of deep priority conflicts, and if programmers are exposed to limitation of the disambiguation techniques respectively? Programmers can usually fall back to explicit disambiguation with brackets, in case the precedence rules are not clear or the parser is unable to parse an input string due to ambiguities. In Section 3.3 we will discuss a method for detecting explicit disambiguation to quantify RQ3: *To what extent do programmers use brackets for disambiguation of priority conflicts explicitly?*

3.1 Contextual Grammars

As mentioned in Section 2, generalized parsers produce a parse forest containing ambiguity nodes corresponding to the possible interpretations of a program. Ideally, a filter should be able to select only one correct interpretation and return it as result. Filtering ambiguities that arise from operator precedence and associativity after parsing is not practical though, as the number of ambiguities can grow exponentially with the number of operators in an expression. Thus, priority conflicts should preferably be solved either at parser generation time or at parse time.

To solve deep priority conflicts, we use a technique based on contextual grammars [5]. This approach consists of a grammar transformation that generates additional productions forbidding *deep* conflicting patterns before parser generation. Furthermore, this technique enables precise measurements of the number of deep conflicts by only duplicating the productions that contribute to solving a certain conflict. For example, the contextual grammar to solve the operator-style conflict described in Section 2, has the following form:

```
Exp.Add = Exp^IfElse "+" Exp
Exp.IfElse = "if" Exp "then" Exp "else" Exp
Exp.Int = INT

Exp^IfElse.Add = Exp^IfElse "+" Exp^IfElse
Exp^IfElse.Int = INT
```

The contextual symbol $\texttt{Exp}^{\texttt{IfElse}}$ indicates that any expression derived by this symbol cannot have an if-else-expression as its rightmost child. This semantics is implemented when duplicating the productions of the non-terminal `Exp`, passing the context **`IfElse`** to the respective rightmost symbols and forbidding the contextual symbol $\texttt{Exp}^{\texttt{IfElse}}$ to derive an **`IfElse`** production itself.

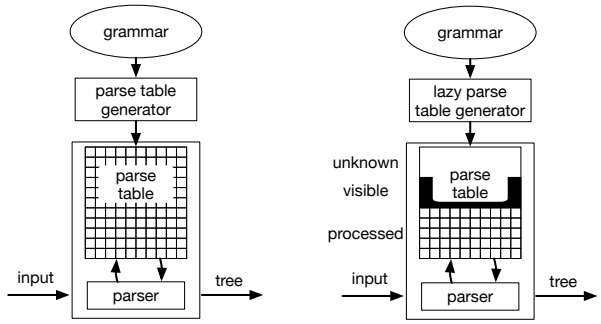
To count the number of deep priority conflicts of a specific type, we use a contextual grammar that solves all-but-one type of priority conflict. For example, to measure the number of operator style conflicts, we use a contextual grammar $\mathbf{G}_{\{\mathrm{DE,LM}\}}$ as a contextual grammar $\mathbf{G}$ that solves dangling else and longest match conflicts, i.e., $\mathbf{G}_{\{\mathrm{DE,LM}\}}$ does *not* contain the additional productions to solve operator-style conflicts. Thus, parsing programs using this grammar produces an ambiguity whenever this program contains an operator-style conflict. Similarly, the contextual grammars $\mathbf{G}_{\{\mathrm{OS,LM}\}}$, which solves only operator-style and longest match conflicts, and $\mathbf{G}_{\{\mathrm{OS,DE}\}}$, which solves only operator-style and dangling else conflicts, can be used to detect dangling else and longest match conflicts, respectively. To guarantee that

all ambiguities that arise in a program are related to deep priority conflicts, we use a contextual grammar $\mathbf{G}_{\{\mathrm{OS,DE,LM}\}}$ that solves all conflicts, verifying that the same program parses unambiguously.

3.2 Lazy Parse Table Generation

Transformation to contextual grammars can produce large grammars when considering languages containing a large number of deep priority conflicts and many different productions that refer to conflicting symbols. We derive contextual grammars from SDF3 [23] syntax definitions, using them in combination with a scannerless generalized LR parser (SGLR) [22]. In this case, the number of states in the generated parse tables can also grow considerably, because states are split to handle each possible interpretation of a conflict.

Since we suspect that a considerable portion of states generated from contextual productions is not used in practice, we adopted *lazy parse table generation* [9]. With that technique, the parser generates parse states on demand and as a result, only those states actually needed for parsing a program or a series of programs are generated. In addition to improving the performance of parse table generation, this technique provides an alternative to measure parse table coverage of a program or corpus of programs.



(a) Conventional Table Generator **(b)** Lazy Table Generator

Figure 1. Table-driven parser with a conventional and a lazy parse table generator, as presented in [9].

A common scenario for most table-driven parsers is described in Figure 1a. A (complete) parse table is generated from the grammar, and a generic parser reads the actions in the parse table to process the input. The table stays the same as long as the grammar has no changes, but whenever compiling a modified grammar, the table generator produces a new full parse table containing all processed states. That is, for larger grammars with many states, generating and loading a large parse table in which only a few states are used, is inefficient.

In a lazy parser generation scenario, whenever the parser requests a state, a lazy generator either processes a new state or returns an already processed state to the parser. Processing a state may create actions that refer to unprocessed states, making them visible. The processed and visible states from all previous parses are cached until the grammar has been changed, and the subsequent parses of the same program do not have any impact on parser generation time. Thus, if most of the programs do not exercise the full grammar, the parser can be regenerated without a big penalty in performance as parser generation time is amortized over parsing many input programs. Figure 1b illustrates the scenario of a table-driven parser in combination with a lazy table generator. Note that all states that are not processed or visited remain unknown.

Applying the conventional SDF3 parser generator and a lazy parser generator to the contextual grammar presented before (replacing $\texttt{Exp}^{\texttt{IfElse}}$ by `Exp1`) produces a parse table defined by the automaton of Figure 2. The complete automaton is generated by the conventional parse table generator, whereas only the highlighted states are processed when using the lazy table generator to parse the program `1 + 2`.

We measure the coverage of contextual grammars by the number of all states visible and processed in the parse tables generated by our lazy generator. We use two different tables: a fresh parse table to parse each separate program and a different table that accumulates the number of (visible and processed) states when parsing all the programs for each corpus. We also measure the number of productions that have been used considering both scenarios.

3.3 Explicit Disambiguation

Programmers might use brackets to explicitly specify the precedence of operators when writing a program. For example, when writing a program `(1 + 2) * 3`, a programmer uses brackets to explicitly state that in this expression the addition should have higher precedence over the multiplication. However, writing a program `1 + (2 * 3)`, does not change the actual precedence of the operators, since the multiplication already has higher precedence over the addition. In the last case, the brackets are redundant as they do not change the shape of the resulting abstract syntax tree (AST).

Programmers might also use brackets to disambiguate deep priority conflicts explicitly. For example, the brackets in the following two expressions result in ASTs that would be forbidden by the original contextual grammar, if we consider the same expressions without brackets:

```
(1 + if e then 2 else 3) + 4
1 + (if e then 2 else 3) + 4
```

In both expressions, the brackets specify a different operator precedence from the one defined in the grammar. To measure the number of brackets used for explicit disambiguation, we investigate the unambiguous ASTs produced by the contextual grammar $\mathbf{G}_{\{\mathrm{OS,DE,LM}\}}$. Because brackets do not appear explicitly in the AST, we added an attribute to AST nodes to indicate whether a node is surrounded by brackets.

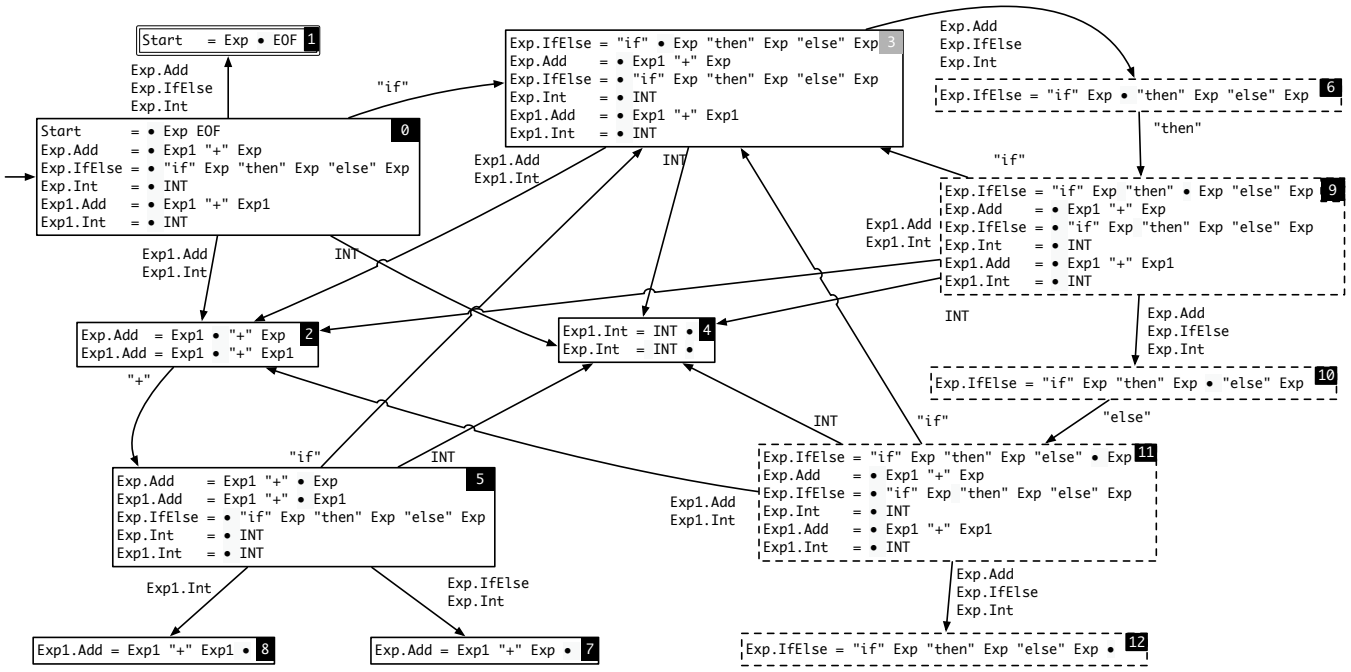


Figure 2. States generated by the conventional and lazy table generation algorithms when applied to the contextual grammar of Section 3.1. The states with striped lines are the ones still unknown by the lazy generator after parsing 1 + 2. Note that state 3 is visible but has not been processed yet.

First, we collect all nodes that have a bracket attribute, calculating the total number of the (pairs of) brackets present in the program. Then, we navigate through the program's AST searching for conflicting patterns, as such patterns are forbidden by the grammar and can only occur inside brackets. We remove the bracket nodes found this way from the initial list, counting the ones that disambiguate deep conflicts and the ones that disambiguate shallow conflicts. The remaining bracket nodes are marked as redundant.

Note that brackets can disambiguate a shallow and a deep conflict at the same time, as illustrated in the example below:

```
1 + (2 + if e then 2 else 3) + 4
```

The brackets are used to disambiguate a deep conflict involving an addition and an if-else-expression, and a shallow conflict which states that the addition inside the brackets should be right associative with respect to the outer one. In cases where brackets disambiguate deep and shallow conflicts, we opted to consider the brackets used in these cases to disambiguate only a deep priority conflict.

4 Evaluation

In the previous section, we devised measurement techniques that enable empirical investigation into how deep priority conflicts occur in practice. In this section, we are applying our method to answer to what extent deep priority conflicts do actually occur in real programs. This pilot study specifically focuses on the syntax of two programming languages — OCaml and Java — that have inherently different attributes.

The OCaml syntax is for the most part expression-oriented, and a large number of deep priority conflicts originate from the expression part. We have used the grammar from the OCaml reference manual, which contains all three types of deep priority conflicts that were discussed in Section 2: operator-style, dangling else and longest match.

In contrast to OCaml, Java is a predominantly statement-oriented programming language. Dangling-else conflicts may apply to if-statements, while expressions are the main subject to operator-style priority conflicts. The Java grammar does not contain longest-match constructs.

Based on the inherently different syntaxes of the two languages, we present our hypotheses of the expected results, grouped according to the research questions.

Hypotheses for RQ1: The first research question is related to the number of conflicts that occur in real programs:

H1 We expect more ambiguities triggered by the expression-oriented grammar of OCaml than by Java's grammar.

H2 The majority of OCaml deep conflicts are longest-match, because many expressions can have pattern matches.

H3 Deep priority conflicts are expected to be sparse in Java programs, as most priority conflicts are shallow.

H4 Overall, deep priority conflicts are sparse and do not occur frequently across programs of both languages.

Hypothesis for RQ2: The second research question considers the efficiency of grammar transformation approaches to solve deep priority conflicts. Our hypothesis is based on the coverage of contextual grammar productions (and parse table states respectively) that are used to solve deep conflicts.

H5 For both languages we expect that only a minor part of grammar productions and parse table states is exercised, even after parsing all programs in the corpus.

Hypotheses for RQ3: The third research question is concerned with explicit disambiguation. Our expectations are:

H6 Due to its expression-oriented syntax, OCaml programs use considerably more brackets than Java programs.

H7 The majority of the brackets in Java and OCaml are necessary for disambiguating shallow priority conflicts.

4.1 Experimental Setup

We directly transcribed the declarative context-free grammar of the OCaml version 4.04 reference manual[6] to SDF3. The natural and ambiguous OCaml grammar contains 1793 productions.[7] The original Java SE 8 reference grammar[8] encodes conflict resolution in the grammar itself. To make deep priority conflicts of Java detectable with our method, we have replaced the syntax for expressions by a natural (and ambiguous) syntax, defining operator precedence and associativity by means of SDF3 priorities. The resulting Java grammar contains 1327 productions.

In our pilot study we examine the top 10 trending open-source projects on GitHub for each language.[9] Two out of the top 10 OCaml projects were misclassified by GitHub, i.e., not containing any OCaml file at all. We removed the misclassified projects and added the subsequent projects from the list. Furthermore, we had to clean one project in order to avoid data duplication. The *bucklescript* repository duplicated the whole *ocaml* project into a subfolder. We removed the subfolder from *bucklescript*, because the *ocaml* project itself is part of our pilot study corpus of OCaml projects.

4.2 Results of the OCaml Case Study

Our pilot study corpus contains 3296 OCaml source files, from which 95.9% (i.e., 3161 files) were successfully parsed with our grammar while 4.1% (i.e., 135 files) could not be parsed due to language extensions that we do not support.[10]

Table 1 summarizes our findings with respect to occurrences of deep priority conflicts and bracket usage considering each project in the OCaml corpus. The table presents the number of affected files of each project, the number of deep priority conflicts found and how frequent each type occurs. The table also shows information about brackets usage, highlighting the number of brackets that have been used for disambiguation in each project. Note that the remaining percentage of brackets for each project corresponds to redundant brackets. In the following paragraphs we discuss data points from the aforementioned table.

Affected Files. 530 OCaml files (i.e., 16.8% of all files) contained deep priority conflicts. Concerning the individual categories, the most frequent priority conflict type was longest match (in 356 files), followed by operator-style conflicts (in 278 files) and dangling-else conflicts (in 7 files).

When looking at combinations of categories, our data shows that 79.6% of files with deep priority conflicts contained conflicts of just a single category, whereas 19.8% mixed two conflict categories, and 0.6% contained three categories.

Deep Priority Conflicts. In total, we discovered 1657 deep priority conflicts in all files. From these, 48% are operator-style conflicts, 51.5% are longest match and only 0.5% are dangling else. When looking at the number of conflicts per project, six projects contained more longest match conflicts, whereas four projects contained a majority of operator-style conflicts. Three out of ten projects had dangling else conflicts. On a per file basis, the maximum number of conflicts observed were 38 operator-style conflicts, 21 longest match conflicts, and 2 dangling-else conflicts.

Disambiguation with Brackets. We observed a total number of 248830 pairs of brackets. From these, 31.3% are redundant, i.e., removing them does not affect the resulting tree. The remaining 68.7% of the brackets account for resolving priority conflicts (67.1% shallow and 1.6% deep conflicts).

Discussion. When looking at the files with most conflicts, we found that the most common patterns of operator-style conflicts have the following form:

```
exp1 op fun param -> exp2 op2 exp3
exp1 op function pattern -> exp2 op2 exp3
```

In most cases, **op** and **op2** are user-defined operators such as >>? and >>=?, whereas **fun** and **function** are function definitions in the OCaml language.

For most dangling else conflicts, we noticed that the problematic **if** was hidden by an inner **let** expression, such as:

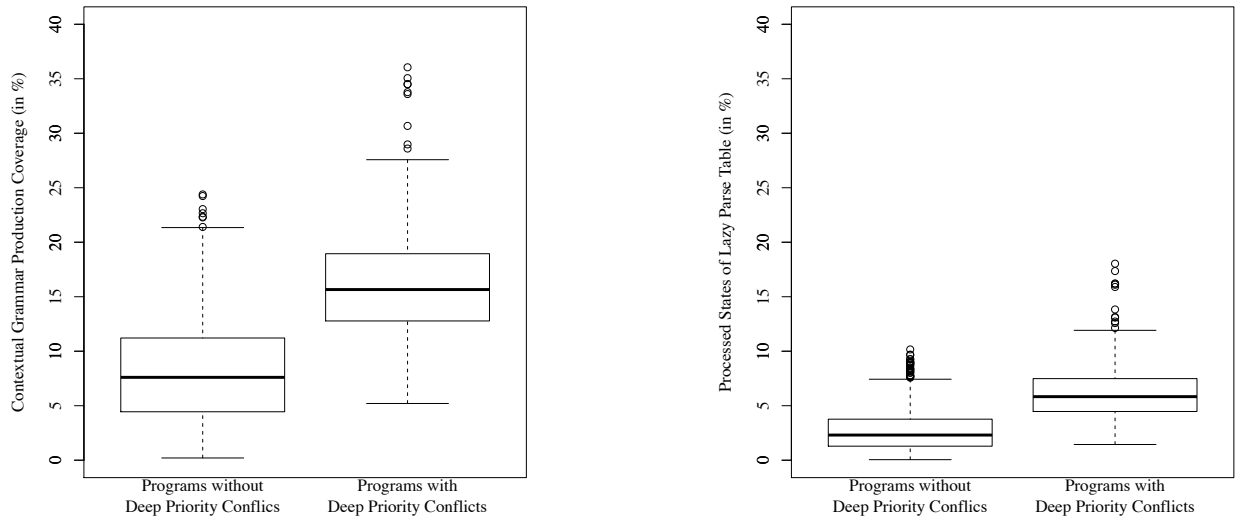
```
if exp1 then let binding1 in
             if exp2 then exp3
             else exp4
```

One interesting remark is that all dangling else conflicts were indented in a way that is consistent with how the conflict is solved by the contextual grammar.

[6] http://caml.inria.fr/pub/docs/manual-ocaml/language.html
[7] We consider the number of productions after SDF3 normalization.
[8] https://docs.oracle.com/javase/specs/jls/se8/html/index.html
[9] https://github.com/trending/ accessed on May 19, 2017.
[10] The grammar currently does not support some language extensions defined in http://caml.inria.fr/pub/docs/manual-ocaml/extn.html.

Table 1. Overview of Deep Priority Conflicts and Bracket Usage in OCaml Corpus.

Project	Affected Files	Deep Priority Conflicts				Disambiguation with Brackets	
		Total Number	Operator Style	Dangling Else	Longest Match	Deep Conflicts	Shallow Conflicts
FStar	6 / 160 (3.8%)	6	33.3%	0.0%	66.7%	607 (1.7%)	12487 (35.7%)
bincat	5 / 26 (19.2%)	26	57.7%	0.0%	42.3%	28 (0.8%)	2735 (81.3%)
bucklescript	85 / 885 (9.6%)	305	50.2%	1.3%	48.5%	924 (2.1%)	29238 (65.3%)
coq	158 / 417 (37.9%)	441	35.4%	0.5%	64.2%	1039 (1.3%)	56083 (69.9%)
flow	52 / 305 (17.0%)	117	36.8%	0.0%	63.2%	278 (1.6%)	13374 (78.9%)
infer	33 / 234 (14.1%)	52	23.1%	0.0%	76.9%	376 (2.5%)	10720 (70.6%)
ocaml	112 / 909 (12.3%)	275	28.4%	0.7%	70.9%	737 (1.6%)	35010 (77.4%)
reason	4 / 36 (11.1%)	14	7.1%	0.0%	92.9%	25 (1.5%)	1194 (73.2%)
spec	4 / 40 (10.0%)	5	100.0%	0.0%	0.0%	15 (1.0%)	1293 (86.1%)
tezos	71 / 149 (47.7%)	416	79.3%	0.0%	20.7%	130 (1.6%)	6969 (84.3%)
All	530 / 3161 (16.8%)	1657	48.0%	0.5%	51.5%	4159 (1.6%)	169103 (67.1%)



(a) Grammar Coverage

(b) Lazy Parse Table Coverage

Figure 3. Comparison of Grammar and Parse Table Coverage between OCaml Programs with / without Deep Priority Conflicts.

Longest match conflicts did not follow a unique form. However, occasionally pattern-matching constructs that contain **`match`**, **`function`**, or **`try`** expressions caused conflicts of the following form:

```
begin function (e, info) -> match e with
  | pattern1
  | pattern2
end
```

In the example above, the indentation does not clarify whether `pattern2` belongs to **`function`** `(e, info)` or **`match`** `e`.

4.3 Results of the Java Case Study

From 9935 Java source files we successfully parsed 97.3% with our grammar. Manual inspection of the failing 2.7% files (i.e., 268 files) revealed that they indeed had syntax errors. All these files contained only snippets of Java code and belonged to a *testData* folder from the *kotlin* repository. Table 2 presents the information about deep priority conflicts and bracket usage in Java, for each project we studied.

Affected Files. In total, only 2 Java files from the corpus contained deep priority conflicts.

Table 2. Deep Priority Conflicts and Bracket Usage in Java.

Project	Affected Files	Disamb. with Brackets	
		Deep Conflicts	Shallow Conflicts
Matisse	0 / 41 (0.0%)	0 (0.0%)	33 (94.3%)
RxJava	0 / 1469 (0.0%)	0 (0.0%)	398 (78.3%)
aurora-imui	0 / 55 (0.0%)	0 (0.0%)	57 (74.0%)
gitpitch	0 / 45 (0.0%)	0 (0.0%)	1 (1.6%)
kotlin	0 / 3854 (0.0%)	0 (0.0%)	4892 (53.3%)
leetcode	0 / 94 (0.0%)	0 (0.0%)	30 (44.8%)
litho	0 / 510 (0.0%)	0 (0.0%)	297 (66.3%)
lottie-android	0 / 109 (0.0%)	0 (0.0%)	134 (87.6%)
spring-boot	2 / 3444 (0.06%)	0 (0.0%)	630 (55.4%)
vlayout	0 / 46 (0.0%)	0 (0.0%)	285 (76.2%)
All	2 / 9667 (0.02%)	0 (0.0%)	6757 (56.1%)

Table 3. Grammar and Parse Table Coverage Statistics.

	Grammar		Parse Table		
	# Prod.	Used	# States	Lazy Expansion	
				Proc.	Visible
OCaml	3420	59.3%	20200	36.4%	45.8%
Java	1916	54.8%	4674	49.0%	56.8%

Deep Priority Conflicts. The pilot study revealed in total two operator-style conflicts involving lambda expressions. The conflicts adhered to the following form:

```
(CastType) () -> exp1 == exp2
```

Lambda expressions have lower priority than expressions for equality comparison (==). In turn, cast expressions have the highest priority amongst the three operators in the previous example. Incomplete disambiguation would allow two different interpretations:

```
((CastType) () -> exp1) == exp2
(CastType) () -> (exp1 == exp2)
```

Our experimental setup made this conflict measurable with context-free grammars that use declarative disambiguation.

Explicit Disambiguation with Brackets. The total number of observed pairs of brackets was 12049. None of these brackets actually avoided deep priority conflicts, however 56.08% of brackets account for resolving shallow priority conflicts. The remaining 43.92% of brackets are redundant.

4.4 Grammar and Parse Table Coverage Statistics

Table 3 lists the statistics for parsing the corpus with the transformed contextual grammars and resulting parse tables.

OCaml. The transformed contextual grammar, capable of resolving all three categories of conflicts that we investigate, expands to 3420 productions. In terms of grammar coverage, parsing all files together exercised 59.3% of the productions.

From the grammar productions, a lazy parse table was generated that could extend to 20200 states, i.e., the number of states that a conventional SDF3 parse table generator would produce. From the possible number of states, 36.4% of the states were processed during parsing, and 45.8% of the states were visible, using the lazy parse table generator.

When looking at individual files, the mean coverage observed was 9.5% (range 0.2–36.1%). Figure 3a splits coverage data between programs free of deep priority conflicts and programs exhibiting deep priority conflicts. We observed that programs free of deep conflicts use on average 8.1% of contextual productions per file, while programs with deep conflicts use, in average, 16%. Figure 3b shows the corresponding data for processed states of the lazy parse table that was generated from the contextual grammar. We observed that programs free of deep priority conflicts exercise on average 2.7% of all possible states compared to 6.17% of programs that do have deep conflicts.

Java. The transformed contextual grammar consists of 1916 productions. Parsing the whole Java corpus exercised 54.8% of the productions. When looking at individual files, the mean coverage measured was 12.95% (range 0.52–34.08%).

The corresponding full parse table has 4674 states. Parsing all Java files with the lazy parse table generator resulted in 49% processed states and 56.8% visible states. Due to the low number of deep priority conflicts found in Java, we omitted separate statistics for programs with and without deep priority conflicts.

4.5 Recapitulation of Hypotheses

Based on the pilot study results reported in previous subsections, we do conclude:

Confirmation of Hypothesis 1: Parsing the OCaml corpus with OCaml's expression-oriented grammar triggered deep priority conflicts in about one out of six files. In contrast, parsing the Java corpus resulted in total in only two deep priority conflicts.

Confirmation of Hypothesis 2: With 51.5%, the majority of deep priority conflicts were of type longest match. To our surprise, with 48%, operator-style conflicts were almost as frequent as longest match conflicts.

Confirmation of Hypothesis 3: Deep priority conflicts are sparse when parsing Java with grammars that use declarative disambiguation. Deep conflicts exclusively occurred in the context of lambda expressions.

Rejection of Hypothesis 4: We did not expect that deep priority conflicts would occur in about one out of six files when parsing OCaml, nor that *all* projects would have files with deep priority conflicts. With a frequency of 16.8% those conflicts can be considered common case, requiring support for (declarative) disambiguation.

Confirmation of Hypothesis 5: The results indicate that there is indeed a high cost attached to declarative disambiguation with grammar transformation techniques. On a per-file basis, our expectations were met. E.g., parsing OCaml files yielded a mean coverage of contextual productions of 12.95% (range 0.52–34.08%). Contrary to our intuition, parsing all files exercised more than 50% of the contextual productions but processed slightly less than 50% of the parse table states in both languages.

Confirmation of Hypotheses 6 and 7: Brackets are more excessively used in OCaml than in Java. In both languages brackets are mainly used to disambiguate (shallow) priority conflicts. However, 1.6% of bracket pairs in OCaml are used to disambiguate deep priority conflicts, suggesting that language users are exposed to deal explicitly with deep conflicts, and also that they rely on the disambiguation policy of the language for disambiguating deep priority conflicts.

Considering both languages, even though only up to 17% of the files contained deep priority conflicts, such conflicts do occur, and there is a need for supporting efficient disambiguation in combination with readable, concise, but inherently ambiguous context-free grammars. One may question whether it is a good language design practice to allow deep priority conflicts to occur in the first place, due to the problems they cause.

If we consider the efficiency of grammar transformation techniques to solve deep priority conflicts, we can conclude that there is room for improvement. Producing an unambiguous grammar that disallows deep conflicts results in many duplicate productions that do not seem to be used, even after parsing a considerable number of programs. This conclusion should lead to follow-up studies that can improve the efficiency of these grammar transformations.

From the programmer's point of view, deep priority conflicts can be even more confusing, as it is necessary to *really* understand the operator precedence specified with the grammar. Deep priority conflicts could even contribute to the decision of whether to use or learn a language, if we consider this extra burden imposed on novice programmers. Therefore, now that we have an indication that deep priority conflicts occur in real code, we can ask: are programmers aware of such conflicts?

From our study, we observed that programmers use brackets relatively often, but that up to 40% of the brackets were redundant. Therefore, we may ask if programmers use redundant brackets for readability or because they did not fully understand the precedence of the language. Considering this aspect, language design may also play a role in how often programmers need to use brackets explicitly. Future empirical studies could lead to more insights that connect language design, declarative disambiguation and how both of them affect programmers.

5 Threats to Validity

In our pilot study, we have decided to examine the number of occurrences of deep priority conflicts by the number of ambiguities that occur when we *turn off* the solutions for such conflicts. Because ambiguities may occur nested within each other, this number corresponds to an under-approximation of the number of deep priority conflicts.

As we mentioned before, a program `1 + if e then 2 else 3 + 4` contains an ambiguity that cannot be solved by a parent-child relation on the addition and if-then-else expressions, causing a deep conflict. Such conflict is captured by our approach via the top-level ambiguity node of the whole expression. However, any operator-style conflict that occurs inside the expression `e` will not be counted, since it is hidden inside the top-level ambiguity of the outer expression.

Furthermore, we do not validate the ASTs produced when parsing each program in our test suite. In order to test the correctness of our grammars, we have used Java and OCaml program snippets that compare expected ASTs with the (unambiguous) ASTs produced when parsing such programs with contextual grammars. These tests stress syntactic elements of each language, including cases of deep conflicts.

When counting brackets that disambiguate deep conflicts, we do not include brackets that would produce the same AST if removed from the program. For example, in the program `1 + (if e then 2 else 3 + 4)` the brackets disambiguate a deep conflict, choosing explicitly how the program should be interpreted. However, when removing the brackets and parsing the same program with a contextual grammar that solves operator-style conflicts, the same AST is produced, i.e., the brackets are redundant.

Operator-style conflicts in Java only occur due to lambda expressions. However, these expressions are relatively new in the Java language, being introduced in Java SE 8. It may be the case that lambda expressions are not yet used frequently by programmers or are only used in newer projects, which could explain the low number of deep priority conflicts we found in Java programs.

Finally, the size of our empirical study could hinder the conclusions we draw in this paper. However, we still believe that this study can give significant insights to the research questions we raised.

6 Related Work

In this section we highlight previous work on parsing and related work on empirical studies (based on corpus analysis).

6.1 Safe and Complete Disambiguation

Grammar-to-grammar transformations. Afroozeh et al. [3] proposed a new semantics for SDF2 priorities [14] that is safe and complete. The approach consists of rewriting the grammar, duplicating the productions for the non-terminals such that shallow and deep conflicting patterns cannot be produced. Even though this approach produces an unambiguous context-free grammar as result, the size of the resulting grammar can be quite large for languages containing many conflicts. Furthermore, the approach does not handle dangling else nor longest match conflicts.

Contextual grammars [5] is a grammar transformation that extends the approach from Afroozeh et al. [3]. The grammar productions are only duplicated to handle deep conflicts, and the technique also solves dangling else and longest match ambiguities. However, the resulting contextual grammars can still have many duplicated productions for languages with many deep conflicts.

Data-dependent grammars. A dynamic solution to safe and complete disambiguation is presented by Afroozeh and Izmaylova [2]. Instead of producing pure and unambiguous context-free grammars, this approach consists of producing a data-dependent grammar that checks for priority conflicts at parse time. Priorities are automatically translated into data-dependent productions that passes the precedence levels of the expression currently being parsed and checks whether it produces a priority conflict. This solution does not handle longest match nor dangling else conflicts, and it is used with a top down parser that supports data-dependency tracking.

6.2 Lazy Parser Generation

IPG. The incremental parser generator IPG [9] was developed with the purpose of speeding-up parser generation in a highly interactive environment. At the early stages of language design, the language's syntax is constantly being changed, invalidating the current parse table. In incremental table generation, only the parts of a partial table that are affected by a change in the grammar are reconstructed at parse time. We have only adopted the lazy generation of IPG, i.e., when the grammar changes, our generator starts from scratch with an empty table. Lazy parse table generation allowed us to measure the coverage of contextual grammars.

ANTLR. A top-down approach to lazy parser generation is used in the Adaptive LL(*) parsing algorithm [18]. ANTLR 4 generates ALL(*) parsers that *adapts* to the input sentences presented to it at parse time. The parser dynamically constructs a prediction automaton that matches the lookahead of input being parsed deciding which production rule to expand. Intermediate results are memoized, i.e., the parser incrementally constructs the prediction automaton by need.

6.3 Corpus Analysis and Grammar Coverage

Empirically studying source code allows researchers to learn from real-world programs, for example to uncover coding practices [10, 17] or to evaluate the performance of tools and analyses. Studies do either reuse existing corpuses of various sizes, or construct corpuses that are suitable for answering their research questions. E.g., Landman et al. refuted common knowledge about the correlation of two source code metrics [16] by constructing and analyzing large corpuses.[11]

In contrast to large scale empirical studies, we conducted a pilot study for getting an intuition of how deep priority conflicts occur *in the wild*. The outcomes of this pilot study pinpoint and characterize real-world issues that arise with deep priority conflicts, guiding future large scale studies.

Context-dependent branch coverage [15] can give more insights on the coverage of contextual grammars. In our experiment, we adopted a rather simplistic approach by counting the number of productions used by the parser. However, our approach also includes information about the coverage of lazily generated tables in order to investigate the efficiency of grammar transformations to solve deep priority conflicts.

6.4 Code Readability and Programming Style

Buse and Weimer [4] define source code readability as *"as a human judgment of how easy a text is to understand"* and proposed a metric for measuring readability. Stefik and Siebert [20] and Sedano [19] provide an overview on empirical studies concerning code readability and programming style.

Instead of focusing on the human judgement perspective, we investigated deep priority conflicts that may hamper unambiguous parsing of source code. We studied how often brackets are used for readability purpose or to disambiguate (deep) priority conflicts. There is empirical evidence that if-statements that remove parentheses and braces are easier to comprehend by novice programmers [20], however such syntaxes are more likely to trigger (deep) priority conflicts.

7 Conclusion and Future Work

We have presented an experiment to analyze deep priority conflicts in real-world programs. Our experiment uses contextual grammars to produce unambiguous grammars that solve deep conflicts. By turning off the generation of productions to solve each type of conflict, we were able to categorize and count occurrences of each type. Furthermore, we used lazy parse table generation to investigate the efficiency of contextual grammars when solving deep conflicts. We have also looked into explicit disambiguation, i.e., counting and analyzing the number of brackets in each program.

[11] One Java corpus containing 17.6M methods that are spread out over 1.7M files, and one C corpus containing 6.3M functions extracted from 462K files.

Our experiment indicates that deep conflicts do occur often in real programs. However, when looking into the efficiency of grammar transformations to solve deep conflicts, we observed that many productions and parse states resulting from the grammar transformation are not used after parsing corpuses of real-world programs.

We also observed that a large percentage of deep conflicts are explicitly disambiguated by brackets, which suggests that the default precedence of the language does not correspond to how programmers typically use the language constructs. The analysis of explicit disambiguation also gave us the insight that brackets are not used exclusively for disambiguation purposes. Further investigation is necessary to understand the actual intention of the programmer when using redundant brackets.

As future work, we propose extending the case study to other languages, i.e., including other SDF3 grammars, and to consider larger corpuses. Based on our findings, we propose further investigation on grammar transformation techniques used for declarative disambiguation, analyzing the reasons for the lack of coverage, and aiming to improve their efficiency when solving deep priority conflicts.

Acknowledgments

The work presented in this paper was partially funded by CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brazil) and by the NWO VICI *Language Designer's Workbench* project (639.023.206).

References

[1] Ali Afroozeh and Anastasia Izmaylova. 2015. Faster, Practical GLL Parsing. In *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Björn Franke (Ed.), Vol. 9031. Springer, 89–108. DOI: http://dx.doi.org/10.1007/978-3-662-46663-6_5

[2] Ali Afroozeh and Anastasia Izmaylova. 2016. Operator precedence for data-dependent grammars. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 13–24. DOI: http://dx.doi.org/10.1145/2847538.2847540

[3] Ali Afroozeh, Mark G. J. van den Brand, Adrian Johnstone, Elizabeth Scott, and Jurgen J. Vinju. 2013. Safe Specification of Operator Precedence Rules. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings (Lecture Notes in Computer Science)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.), Vol. 8225. Springer, 137–156. DOI: http://dx.doi.org/10.1007/978-3-319-02654-1_8

[4] Raymond P.L. Buse and Westley R. Weimer. 2008. A Metric for Software Readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 121–130. DOI: http://dx.doi.org/10.1145/1390630.1390647

[5] Luis Eduardo de Souza Amorim, Timothée Haudebourg, and Eelco Visser. 2017. *Declarative Disambiguation of Deep Priority Conflicts*. Technical Report TUD-SERG-2017-014. Delft University of Technology.

[6] James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, and Alex Buckley. 2013. *The Java Language Specification, Java SE 7 Edition.*

[7] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification. Java SE 8 Edition.*

[8] Jan Heering, P. R. H. Hendriks, Paul Klint, and Jan Rekers. 1989. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices* 24, 11 (1989), 43–75. DOI: http://dx.doi.org/10.1145/71605.71607

[9] Jan Heering, Paul Klint, and Jan Rekers. 1989. Incremental Generation of Parsers. In *PLDI*. 179–191.

[10] Mark Hills, Paul Klint, and Jurgen Vinju. 2013. An Empirical Study of PHP Feature Usage: A Static Analysis Perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 325–335. DOI: http://dx.doi.org/10.1145/2483760.2483786

[11] Andrew Hunt and David Thomas. 1999. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[12] S. C. Johnson. 1975. *YACC—yet another compiler-compiler*. Technical Report CS-32. AT & T Bell Laboratories, Murray Hill, N.J.

[13] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. 2010. Pure and declarative syntax definition: paradise lost and regained. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, 918–932. DOI: http://dx.doi.org/10.1145/1869459.1869535

[14] Paul Klint and Eelco Visser. 1994. Using Filters for the Disambiguation of Context-free Grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano, Milano, Italy.

[15] Ralf Lämmel. 2001. Grammar Testing. In *Fundamental Approaches to Software Engineering, FASE 2001 (Lecture Notes in Computer Science)*, Heinrich Hußmann (Ed.), Vol. 2029. Springer, 201–216. DOI: http://dx.doi.org/link/service/series/0558/bibs/2029/20290201.htm

[16] Davy Landman, Alexander Serebrenik, Eric Bouwers, and Jurgen J. Vinju. 2016. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions. *Journal of Software: Evolution and Process* 28, 7 (2016), 589–618. DOI: http://dx.doi.org/10.1002/smr.1760 JSME-15-0028.R1.

[17] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 507–518. DOI: http://dx.doi.org/10.1109/ICSE.2017.53

[18] Terence John Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: the power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 579–598. DOI: http://dx.doi.org/10.1145/2660193.2660202

[19] T. Sedano. 2016. Code Readability Testing, an Empirical Study. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. 111–117. DOI: http://dx.doi.org/10.1109/CSEET.2016.36

[20] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.* 13, 4, Article 19 (Nov. 2013), 40 pages. DOI: http://dx.doi.org/10.1145/2534973

[21] Masaru Tomita. 1985. An Efficient Context-Free Parsing Algorithm for Natural Languages. In *IJCAI*. 756–764.

[22] Eelco Visser. 1997. *Syntax Definition for Language Prototyping*. Ph.D. Dissertation. University of Amsterdam. Advisor(s) Paul Klint.

[23] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. 2012. Declarative specification of template-based textual editors. In *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*, Anthony Sloane and Suzana Andova (Eds.). ACM. DOI: http://dx.doi.org/10.1145/2427048.2427056