ANTLR2PEG: A tool to convert ANTLR grammars into PEG grammars

José Eduardo Santos
DIMAP, UFRN
Natal, Brazil
jose.santos.712@ufrn.edu.br

Sérgio Queiroz de Medeiros
ECT, UFRN
Natal, Brazil
sergio.medeiros@ufrn.br

Abstract

Context-Free Grammars (CFGs) and Parsing Expression Grammars (PEGs) are formalisms for describing the syntax of languages, where the visual description of a CFG for a language is similar to the corresponding PEG one. Due to this similarity and the fact that CFGs predate PEGs by several decades, it is common practice to derive a PEG from a BNF-like description of a CFG, intended for parser generators like ANTLR, rather than writing the PEG parser from scratch. However, the distinct semantics of PEGs may make this manual translation error-prone. We describe several automatically transformations that help to produce, from an ANTLR grammar, a PEG parser that recognizes the same language. We present ANTLR2PEG, a tool that implements these transformations, and we evaluated it against a corpus of more than 200 community-made ANTLR grammars, comparing the result of the ANTLR parser with the PEG one. Each translatable grammar is evaluated against 100 automatic generated inputs and against the examples that accompany the grammar in the ANTLR repository. From the 262 grammars used from the ANTLR/grammars-v4 repository, ANTLR2PEG can generate a PEG parser for 146 of them. Considering the examples, the PEG parser and the ANTLR one agree perfectly for 42.5% of these 146 grammars. On generated tests, we have a percentage of 35.8% on parsers agreement. We also perform a more in-depth analysis for 10 ANTLR grammars, evaluating how ANTLR2PEG's output can help to manually fix PEG parsers.

Keywords

Parsing, Parser Generators, ANTLR, Parsing Expression Grammars, Context-Free Grammars

1 Introduction

Parsing Expression Grammars (PEGs) are a recognition-based formalism for describing languages that can be an alternative to Context-Free Grammars (CFGs) [3]. Given the amount of time and work invested in the research and use of CFGs, there are numerous tools capable of generating parsers in many programming languages from a CFG description, such as YACC [12], JavaCC [11], and ANTLR [17].

To make the process of writing a PEG parser for an existing language easier, it is common to modify an existing CFG description of such a language to suit the PEG formalism. Although a BNF-like description of a CFG is similar to its corresponding PEG, a naive translation from a CFG to a PEG is an error-prone approach, due to the semantic differences between both formalisms.

ANTLR is a parser generator tool with thousands of monthly downloads, and is widely used in industry and open-source libraries. It has a repository, maintained by the community, containing more than 200 grammars for existing languages.

We implement ANTLR2PEG, a tool that transforms ANTLR grammars into PEG parsers, and provide a detailed description of each step required to achieve a working PEG parser while recognizing the same language as the ANTLR grammar. When ANTLR2PEG cannot determine how to translate a given part of an ANTLR grammar, it reports the conflicts that need manual analysis.

Verifying that two CFGs are equivalent is an undecidable problem [1], and so is verifying equivalence between PEGs [3]. As a consequence, verifying the equivalence between a PEG and a CFG is also undecidable. Therefore, we consider that both parsers are equivalent based on input-based testing. To check that a generated PEG parser is equivalent to its corresponding ANTLR parser, we run both parsers against a set of inputs and we expect that they present the same behavior for each input.

ANTLR2PEG aims to help developers who want to switch from the ANTLR grammar description syntax to a PEG paradigm. Unlike ANTLR, which requires a separate code-generation step and a language-specific runtime, PEG libraries such as LPEG[9] allow a parser to be built as a value in the host programming language, without an additional tool in the pipeline. Beyond that, ANTLR2PEG's extensible nature allows support for multiple programming languages and PEG libraries, helping teams migrate away from ANTLR without being locked into rewriting the parser by hand for each target.

Figure 1 shows an overview of our approach. From an ANTLR grammar, we generate a PEG parser using ANTLR2PEG. We then validate both parsers by testing them against manual inputs, which were originally crafted to validate the ANTLR parser, and inputs generated by *Grammarinator*.

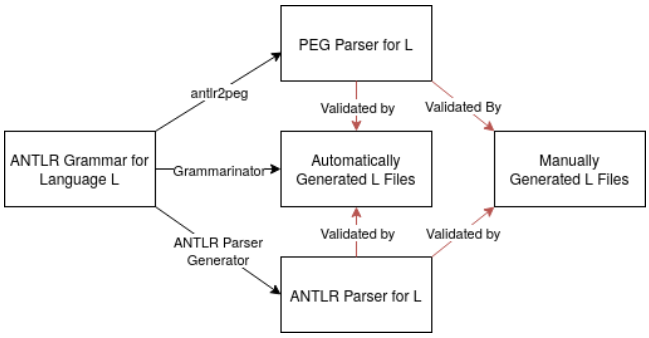

Figure 1: Overview of ANTLR2PEG validation, where we generate a PEG parser from an ANTLR grammar and test it against manual and generated inputs

From the 262 ANTLR4 grammars used, we generate 146 valid PEG grammars and validate them against 100 generated inputs and the examples available for each grammar.

We also present an in-depth analysis of 10 grammars, detailing how many manual changes were required, how many original rules were modified, and how many times each transformation could be applied to those specific grammars.

The rest of the paper is divided as follows: Section 2 describes PEGs and their semantics in detail, highlighting the differences between PEGs and CFGs. Section 3 discusses the rationale, requirements, and implementation of each transformation required to achieve the desired outcome, and emphasizes the grammars that cannot be translated, along with the underlying reasons.

Section 4 presents the validation methodology by using automatically generated inputs together with the examples available from each ANTLR parser.

In Section 5, we report how many of the corpus grammars yielded valid PEGs and validate each of them against their examples and generated tests. Section 6 discusses related work, and Section 7 presents our conclusions and possibilities for future work.

2 Background

2.1 Notation

Throughout this paper, we describe ANTLR grammars using rules of the form $A : a$, where A is a non-terminal and a, the right-hand side of a rule, is an expression. The description of a is based on the EBNF (Extended Backus-Naur Form) notation. ANTLR has three kinds of rules: syntactic, lexical, and fragment. Lexical rules must start with an uppercase letter and describe tokens. Fragment rules contain the word *fragment* before their names and are used exclusively by lexical rules; they describe subparts of a token. Syntactic rules must start with a lowercase letter. ANTLR does not use the classical EBNF notation for repetition and optional; instead, it uses the symbols $*$ and ?. It also defines the symbol + to express repetitions of one or more elements.

We use $A \leftarrow e_1\ e_2$ to describe PEG rules, while rules in the form $A \rightarrow \alpha_1\ \alpha_2$ are CFG ones.

Examples of input will be `monospaced`, literals will be in SANS-SERIF, and references to rules will be in *italics*.

2.2 Parsing Expression Grammars

A PEG is defined as a 4-tuple (N, Σ, P, e_s) where:

- N is a set of non-terminals.
- Σ is the set of terminals.
- P is a set of parsing rules in the form $A \leftarrow e$, where $A \in N$.
- e_s is the starting expression.

A parsing expression e can be inductively defined as:

$$e ::= \epsilon \mid a \mid e_1 e_2 \mid e_1 / e_2 \mid !e_1 \mid e_1*$$

where $a \in \Sigma$ and e_1, e_2 are parsing expressions.

One of the key differences between CFGs and PEGs is the ordered choice operator (/), which tries each alternative once from left to right, backtracking the input position when an alternative fails and stopping trying other alternatives when one succeeds.

It is important to note that PEG ordered choice has local backtracking. This implies that if an expression after the choice fails, the parser will not backtrack and try the next alternative of the choice.

The main difference between PEGs and CFGs is that PEGs are unambiguous. Ambiguity in CFGs occurs when at least one string in the language admits more than one parse tree. A classical example is the "dangling else", presented as a CFG in Example 1, in which it is not possible to determine if an ELSE should be associated with the closest or an earlier IF.

$$\begin{aligned} \text{if\_stmt} \rightarrow\ & \text{`if'}\ \mathit{COND}\ \text{`then'}\ \mathit{COND} \\ \mid\ & \text{`if'}\ \mathit{COND}\ \text{`then'}\ \mathit{STMT}\ \text{`else'}\ \mathit{COND} \end{aligned} \tag{1}$$

Resolving this ambiguity in CFGs typically requires tool-specific mechanisms, such as rule precedence or longest-match directives. PEGs can express the correct behavior with the use of the / operator, making the grammar unambiguous.

The ordered choice operator eliminates this ambiguity by enforcing deterministic semantics, so if we simply change the CFG choice ("|") to a PEG ordered choice ("/"), the grammar becomes unambiguous; however, it may describe a different language. Moreover, this implies that the behavior of the associated PEG parser must be specified at the grammar level, following PEG semantics, while the behavior of a CFG parser is also determined by implementation details that are difficult to specify by using only CFG semantics. If we simply change the operators, as in Example 2, when trying to consume an IF followed by an ELSE, the first option will succeed, but the `else` will remain in the input. This will lead the parser to fail if no other place consumes the `else`, since the parser will not backtrack once the first alternative succeeds.

$$\begin{aligned} \text{if\_stmt} \leftarrow\ & \text{`if'}\ \mathit{COND}\ \text{`then'}\ \mathit{STMT} \\ /\ & \text{`if'}\ \mathit{COND}\ \text{`then'}\ \mathit{STMT}\ \text{`else'}\ \mathit{STMT} \end{aligned} \tag{2}$$

However, the expected behavior of most languages is that the `else` is linked to the closest `if`. To express this, the previous grammar must be modified so that the first alternative always tries to match an `else` following the `if`, as in Example 3.

$$\begin{aligned} \text{if\_stmt} \leftarrow\ & \text{`if'}\ \mathit{COND}\ \text{`then'}\ \mathit{STMT}\ \text{`else'}\ \mathit{STMT} \\ /\ & \text{`if'}\ \mathit{COND}\ \text{`then'}\ \mathit{STMT} \end{aligned} \tag{3}$$

Beyond ordered choices, PEGs have the operators ! and &, which match an expression without consuming the input. This permits solving ambiguities that require arbitrary *lookahead* at the grammar level. The expression $!e$ succeeds when e does not match, while the expression $\&e$ succeeds when the input matches the expression e. $\&e$ is omitted from the original definition since it can be defined as $!(!e)$ [3, §3.2].

Another solution to the dangling-else problem without changing the order is to use the operator "!", as described in the example below.

$$\begin{aligned} \text{if\_stmt} \leftarrow\ & \text{`if'}\ \mathit{COND}\ \text{`then'}\ \mathit{STMT}\ !\text{`else'} \\ /\ & \text{`if'}\ \mathit{COND}\ \text{`then'}\ \mathit{STMT}\ \text{`else'}\ \mathit{STMT} \end{aligned} \tag{4}$$

In this example, the !'else' ensures that the first condition will only match when it is not followed by an 'else'. However, this option introduces extra *lookahead*. Therefore, changing the order instead of using a *lookahead* operator, when possible, is encouraged.

2.3 ANTLR

ANTLR [17] is a parser generator tool based on the ALL(∗)algorithm. It describes grammars using EBNF syntax, which is then parsed with ALL(∗) [18], a top-down parsing algorithm capable of handling ambiguities by using adaptive *lookahead*.

The use of ALL(∗)allows ANTLR to solve conflicts such as the "dangling else" by longest-match while not fixing a *lookahead*. ANTLR can also extend the recognized language without extending the grammar by using directives – such as *case-insensitive*, which extends lexical rules to match both lowercase and uppercase letters.

The user can solve a conflict by using semantic rules, which allows the current parser state to be inspected before taking a decision. Along with that, it is also possible to modularize the description of an ANTLR parser by separating it into multiple parts, such as tokens, lexer, and grammar, or just lexer and grammar.

Given the adaptive *lookahead*, ANTLR can describe if a repetition is greedy or lazy. By default, repetition operators (?, +, and ∗) are eager, but it is possible to set their behavior to be lazy by using ? as a suffix, turning them into ??, +?, and ∗?.

While using an algorithm that belongs to the LL class, ANTLR handles direct left recursion by rewriting it internally. However, indirect and hidden recursion is not supported [18]

Equipped with all these capabilities, ANTLR proves to be a very expressive and appealing tool to generate parsers from a grammar description.

2.4 From ANTLR To PEG

Establishing the equivalence between a PEG and a CFG is an undecidable problem. Additionally, ANTLR relies on implementation details and has ways to extend a CFG without modifying it. This makes a complete formalization of the equivalence between a PEG-based parser and an ANTLR-generated parser difficult.

However, we show that it is possible to translate several expressions used by ANTLR grammars into equivalent PEG expressions. Furthermore, we can also perform approximate transformations and report which rules could not be translated, requiring a manual analysis. This allows the user to continue the work from a grammar where all translatable rules were converted and the remaining ones were flagged.

We extend the classical definition of FIRST and FOLLOW sets [1] to deal with expressions that appear on the right-hand side of ANTLR rules. Moreover, instead of being sets of terminals, we define them as sets of tokens. A token is either a non-terminal that appears on the left-hand side of an ANTLR lexical rule or a string literal that appears on the right-hand side of an ANTLR syntactic rule.

We use $!\,\mathrm{FIRST}(e)$ and $!\,\mathrm{FOLLOW}(e)$ as syntactic sugar for a parsing expression that does not match any token in, respectively, $\mathit{FIRST}(e)$ and $\mathit{FOLLOW}(e)$.

To generate a PEG parser, we convert the ANTLR grammar received as input into its respective AST. This AST goes through our transformations, which convert the ANTLR grammar into a PEG grammar that accepts the same language. This PEG grammar is also represented as an AST, which allows us to target different languages and PEG implementations when converting it into a PEG parser. We implement an LPEG [10] backend, which yields an LPEG parser written in Lua from the PEG AST.

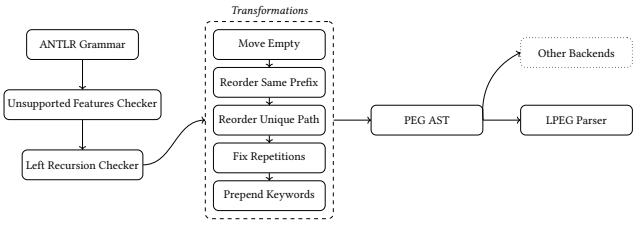

Figure 2: ANTLR2PEG pipeline. From an ANTLR grammar, after ensuring that the grammar is supported, we pass the grammar through our defined transformation, which yields a PEG AST that can be translated into a PEG parser.

Section 3 describes each proposed translation in detail and makes the requirements explicit.

Before applying any transformation, we check if the grammar contains any unsupported feature such as semantic actions to handle ambiguities or multiple lexer modes, and reject the grammar if it is found. These features are not initially supported due to their complexity, but can be implemented in the future.

After that, we check for left-recursive rules in the grammar—which are supported by ANTLR—rejecting the grammar if at least one is found. By definition, left-recursive rules cause PEG rules to loop, which makes the parser never terminate.

For this reason, our current implementation rejects grammars containing left-recursion, since LPEG does not support it and it is not supported by traditional PEG semantics. However, there are extensions to PEG semantics that allow and handle left-recursion to some extent [15, 23, 26], and PEG implementations that implement these extensions, such as OhmJS and parboiled2. In the future, additional backends for those libraries and their respective programming languages can be implemented.

Figure 2 shows the described ANTLR2PEG pipeline and transformation ordering in detail.

3 Transformations

When deriving a PEG from an ANTLR grammar, some precautions must be followed. One of the challenges is to translate ANTLR peculiarities, such as directives and separated lexer and grammar definitions, into a PEG parser.

The subsections below describe the problems being tackled and the solutions to each. Subsection 3.1 shows when a CFG choice can be naively transformed into a PEG ordered choice and proposes heuristics that can be used when the naive transform is not possible. Subsection 3.2 describes how to transform both greedy and lazy ANTLR repetitions into a PEG repetition by using PEG predicates, and Subsection 3.3 shows how lexical rules are handled.

3.1 Alternatives

Mascarenhas et al. proved that an LL(1) grammar, without expressions that match the empty string, describes the same language either if considered as the description of a CFG or as the description

```
      stmt : node_stmt | edge_stmt | attr_stmt |
             id '=' id | subgraph
 node_stmt : node_id attr_list?
  subgraph : ('subgraph' id?)? '{' stmt_list '}'
 attr_list : ('[' a_list? ']')+
 edge_stmt : (node_id | subgraph) edgeRHS attr_list?
   edgeRHS : (edgeop (node_id | subgraph))+
    edgeop : '->' | '--'
   COMMENT : '/*' .*? '*/'
```

Figure 3: ANTLR4 grammar for DOT

of a PEG. Furthermore, they also proved that when such a grammar has a choice where one of the alternatives matches the empty string, we can still establish the correspondence by putting such an alternative as the last one of the choice [13].

To determine if a grammar is LL(1), we must verify that for every production of the form $A \rightarrow \alpha_1 \mid \alpha_2$, the two conditions below are true:

- $FIRST(\alpha_1) \cap FIRST(\alpha_2) = \emptyset$
- $FIRST(\alpha_1) \cap FOLLOW(A) = \emptyset$ When $\epsilon \in FIRST(\alpha_2)$

For each choice, we check whether it is LL(1). If it is, we change the CFG choice operator to the PEG ordered choice operator. However, when a choice is not LL(1), we do not have the correspondence established by Mascarenhas et al. Moreover, determining whether an arbitrary context-free grammar is unambiguous is undecidable [8]. So we cannot swap the CFG choice with the PEG ordered choice. Instead, ANTLR2PEG reports all choices where two alternatives may match the same input and expects the end user to correct the grammar.

Nonetheless, before reporting it, we try heuristics that may determine the order of the choice. Those heuristics are described below.

3.1.1 Nullable Alternatives. We say that an expression e is *nullable* when $\epsilon \in \text{FIRST}(e)$. Since matching ϵ always succeeds in a PEG, any nullable alternative must be placed after all non-nullables before transforming the choice into an ordered choice.

3.1.2 Prefix Choices. A common case where two alternatives share overlapping FIRST sets happens when one alternative is a prefix of another. An expression e_1 is said to be a prefix of e_2 if e_2 can be defined as $e_2 : e_1\ e_3$, where e_3 is an arbitrary expression. Section 2.2 illustrates this with the *dangling else* problem.

ANTLR2PEG detects prefix relationships between alternatives by comparing their syntactic structure. When one alternative is identified as a prefix of another, the choice is reordered so that the longer alternative appears first.

3.1.3 Unique Paths. We use the notions of *unique tokens* and *unique paths* for PEGs described by Medeiros et al. [2]. A token a is considered a *unique token* when only one of the rules in the grammar matches it.

For example, in the grammar for the DOT language described in Figure 3, tokens '->' and '--' are *unique*, since they appear only once, on the right-hand side of rule *edgeop*. The same applies to token '[', which only appears in rule *attr_list*.

From the notion of *unique tokens*, the concept of a *unique path* is defined. A syntactic rule A is a *unique path* when it must match a *unique token* or another *unique path*. It is safe to say that when a *unique token* is matched, no other rule in the grammar would succeed. Therefore, it is safe to reorder choices by placing an alternative that matches a *unique path* before other alternatives that do not match one.

For example, consider the rule *stmt* from the DOT grammar. The right-hand side of this rule is a choice that is not $LL(1)$, since *id* belongs to the FIRST set of *node_stmt*, *edge_stmt*, and *id*'='*id*. Thus, we cannot determine the correct order of these alternatives.

However, since *edge_stmt* matches the *unique path edgeop*, which must match '->' or '--' to succeed, it is safe to put it before the other alternatives. Alternative *attr_stmt* is also a *unique path* because it must match '['. Therefore, it is also moved. Turning the rule *stmt* into:

$$stmt \leftarrow edge\_stmt \,/\, attr\_stmt \,/\, node\_stmt \,/\, id\ \text{'='}id \,/\, subgraph$$

3.1.4 Reordering Tokens By Length. Since ANTLR separates the lexical and syntactic phases, operators and keywords such as '<=', '<', 'for' and 'forall' are resolved in the lexical phase, and the syntactic rules see them only as tokens. This implies that the operators '<' and '<=' will be referred to in the grammar as the *LESS_THAN* and *LESS_EQUAL*.

This is a problem when those tokens are used in choices. For example, consider the following rule:

$$operators : LESS\_THAN \mid LESS\_EQUAL$$

In this case, $FIRST(LESS\_THAN) \cap FIRST(LESS\_EQUAL) = \emptyset$, so no conflict is reported and the choice operator is simply changed to the PEG ordered choice. However, *LESS_THAN* is a textual prefix of *LESS_EQUAL*. So it must come first, or else it may leave an input that will make the parser fail.

To circumvent this, when a rule is a choice exclusively composed of literals or lexical rules, we reorder the choice by token length. Turning the example above into:

$$operators : LESS\_EQUAL \mid LESS\_THAN$$

This would be correctly translated into the following PEG:

$$operators \leftarrow LESS\_EQUAL \,/\, LESS\_THAN$$

3.2 Repetitions

ANTLR implements two kinds of repetitions, lazy and greedy. Greedy repetitions consume as much of the input as possible, while lazy repetitions consume just enough until the next required token.

ANTLR can decide when a repetition should stop consuming due to the adaptive *lookahead*. For example, in the following rule:

$$A : a* a$$

If we try to match the input `aaa`, ANTLR will correctly bind the first two `a`'s to the `a`$*$ and leave the last `a` for the remainder of the rule, so it succeeds.

3.2.1 Greedy Repetitions. However, if we naively translate the same rule into a PEG, the whole input will be consumed by $a*$ and the rule will fail since no a remains to match the last terminal. This occurs because PEG repetitions are blind and consume as much input as possible without *lookahead*.

To identify when this problem occurs, consider the following rule:

$$E \leftarrow e_1* \; e_2$$

The expression e_2 can fail if e_1* consumes input that should be left for e_2. This occurs when:

$$\text{FIRST}(e_1) \cap \text{FOLLOW}(e_1*) \neq \emptyset$$

That is, when e_1 can match something that is also expected to follow the repetition. When the two sets are disjoint, it means that e_1* does not consume anything that comes after it. Therefore, no treatment is needed. However, when there is an intersection, the repetition must be rewritten using PEG predicates to simulate ANTLR's adaptive *lookahead* and backtrack when needed.

Thus, the repetition $e*$ is transformed into e', where e' is a new rule, defined as $\leftarrow e\; e' \,/\, \&\,\text{FOLLOW}(e*)$. This way, e' repeats itself until failure. However, when a failure occurs, it backtracks and checks if it matches what is expected after $e*$, having the same behavior as ANTLR's greedy repetitions for cases where *lookahead* is needed.

3.2.2 Lazy Repetitions. Lazy repetitions consume the input as little as possible, stopping when the next token or rule can succeed. For example, the rule *COMMENT* from Figure 3 uses a lazy repetition to ensure that once a comment is opened by '/*', it always closes on the nearest '*/'. Consider the example below.

```
1   /* Start of File */
2   graph {}
3   /* End of File */
```

If the greedy operator were used instead, *COMMENT* would start on '/*' from line 1 and only stop consuming on '*/' from line 3, incorrectly consuming the whole input as a single comment.

Unlike ANTLR, PEGs do not have lazy operators. To translate *COMMENT*, we use the "!" operator together with the tokens that may follow the repetition. In this case, the token is */, which transforms *COMMENT* into:

$$COMMENT \leftarrow \text{'/*'} \;\; (!\text{'*/'} \;\; .) * \;\; \text{'*/'}$$

The expression !(*/) ensures that the repetition only occurs if the closing delimiter has not been reached yet, replicating the lazy behavior.

Generalizing, a lazy repetition $e_1*?\; e_2$ must be transformed into a PEG expression that prevents e_1* from consuming anything that should be matched by e_2. Unlike greedy repetitions, this transformation is always required. The transformation is:

$$(!\,\text{FIRST}(e_2)\; e_1)* \; e_2$$

$!\,\text{FIRST}(e_2)$ checks whether the current input does not match e_2. If so, e_1 keeps repeating. When e_2 can match, the repetition stops. This directly replicates the behavior of ANTLR's lazy repetitions.

3.3 Lexical Elements

ANTLR separates lexical and syntactic analysis into two different phases. During the lexical phase, keywords are tokenized before the parsers run, ensuring that an input like `if` is always recognized as a keyword and never as an identifier. PEGs, however, are *scannerless*, i.e., both phases are unified into a single pass over the input. This means that, without disambiguation, an identifier rule may consume a keyword before the correct rule has a chance to do so.

Consider the following PEG grammar:

$$\begin{aligned} identifier &\leftarrow LETTER \;\; (LETTER \,/\, DIGIT)*; \\ assign &\leftarrow identifier \text{ '='} \; expr \text{ ';'} \\ stmt &\leftarrow assign \,/\, \text{'if'} \; expr \text{ 'then'} \; expr \text{ 'else'} expr; \end{aligned}$$

If we try to match any input starting with 'if', the *assign* rule will succeed, since `if` matches the *identifier* rule. The *stmt* rule will never reach the second alternative, even for valid `if` expressions.

To fix this, ANTLR2PEG collects all keywords defined in the grammar and introduces a new rule, named *keywords*, that enumerates each keyword into an ordered choice, sorted by keyword length. A not operator (!) followed by *keywords* (*!keywords*) is prepended to the identifier rule, ensuring that no identifier can match a reserved keyword. Turning the previous grammar into:

$$\begin{aligned} identifier &\leftarrow !keywords \; LETTER \;\; (\; LETTER \,/\, DIGIT \;)*; \\ assign &\leftarrow identifier \text{ '='} \; expr \text{ ';'} \\ stmt &\leftarrow assign \,/\, \text{'if'} \; expr \text{ 'then'} \; expr \text{ 'else'} \; expr; \\ keywords &\leftarrow \text{'if'} / \ldots \end{aligned}$$

This ensures that `if` is not matched as an identifier, while `iff` is.

However, using only *!keywords* does not suffice. Consider this valid assignment as input `iff = true`. Rule *keywords* would match `iff` because it has `if` as an textual prefix. But it would not consume the input, making the parser fail.

To address this, we must not only verify that *identifier* does not match a keyword, but also that rule *keywords* does not match a valid identifier. To ensure this, we check if *keywords* does not match the prefix of an identifier. To do so, we define a rule *idRest* that describes what can compose an identifier.

$$idRest \leftarrow (LETTER \,/ DIGIT \,/\, \text{'\_'})$$

We then append this rule with a "!" operator to the *keywords* rule, ensuring that *keywords* matches only the literals specified.

$$keywords \leftarrow (\text{'if'} \,/\, \text{'else'}) \;\; !idRest$$

ANTLR2PEG searches for common names such as *IDENT*, *IDENTIFIER*, and *ID* to select what the identifier rule is. If the rule that matches an identifier is named differently, the user must inform the rule name so ANTLR2PEG prepends the correct rule with *!keywords*.

We define *idRest* as specified above since it is a common pattern. If a grammar contains a different definition of *idRest*, the user must specify it.

3.3.1 Conflicting Lexical Rules. ANTLR allows lexical rules to be defined in terms of other lexical rules or fragments. However, lexical rule are always treated as tokens in the FIRST calculation. Therefore, two rules that share the same structure would have disjoint FIRST sets. Consider two rules, *REAL* and *INTEGER*, defined in terms of *DIGIT*:

$$INTEGER : DIGIT+$$
$$REAL : DIGIT+ \text{ '.' } DIGIT*$$

If both of these rules were in a choice, no conflict would be detected and the choice would be transformed into an ordered choice. However, if put first, the rule *INTEGER* could consume a prefix of rule *REAL*, and the PEG parser would fail.

To address this, during the conflict check phase, we extend the calculation of FIRST to expand on lexical rules defined in terms of others. This way, ANTLR2PEG can report when two lexical rules in the same choice can match the same input. For example, in the analysis phase, $\text{FIRST}(INTEGER) = \{DIGIT\}$ and $\text{FIRST}(REAL) = \{DIGIT\}$, making their intersection non-empty and triggering a conflict report.

When such a conflict is detected, ANTLR2PEG reports it and requires manual intervention. An automatic fix is not applied because the choice may contain additional rules beyond the conflicting pair, and determining the correct order in the general case is undecidable.

Section 5.1.2 presents a concrete case where this conflict required restructuring beyond simple reordering.

4 Testing The Equivalence

We aim to verify that for every input *s*, if the ANTLR parser recognizes *s*, then the generated PEG parser must also recognize *s*. We use the ANTLR parser as a source of truth since ANTLR grammars are the input to our tool. Therefore, any input accepted by the ANTLR parser belongs to the language that the PEG parser should recognize.

Relying only on the community examples available in the corpus does not suffice. The number and complexity of examples vary across grammars, and they may not cover all grammar rules.

Therefore, to examine the validity of PEG parsers, we verify if they recognize generated inputs that are not present in the testing corpus. In addition, we also verify how much change is required to make a totally or partially failing parser recognize all the examples and extra inputs.

To generate them, we use *Grammarinator* [7], a grammar-based fuzzer that generates syntactically valid inputs from an ANTLR4 grammar. We generate 100 new inputs using *Grammarinator* with a derivation depth of 20.

5 Evaluating Results

To evaluate ANTLR2PEG, we use manual test files used to validate the original ANTLR grammars, and also tests generated by *Grammarinator* from the ANTLR grammar. Each grammar in the corpus of available ANTLR grammars has a set of examples and a JSON file with metadata specifying the language name, the parser, the separated lexer (when present), the starting rule, and the available examples. We run the tests on the ANTLR/grammars-v4 repository at the commit hash *55d2bd37ca9b4271ff1f5cb3868e7d58e68f5a0f*.

Table 1: Distribution of the 262 ANTLR4 grammars from the ANTLR/grammars-v4 repository by translation outcome. Left-recursive and unsupported grammars cannot be translated automatically by ANTLR2PEG.

Category	Total	Percent
Valid	146	55.7%
Left-Recursive	73	27.9%
Unsupported Features	40	15.2%
Incorrect	3	1.1%

Table 2: Results of running the generated PEG parsers by ANTLR2PEG against the corpus examples of each grammar.

Status	# Examples	Percentage
All passed	62	42.5%
All failed	54	37.0%
Mixed Results	30	20.5%

Table 3: Results of running the generated PEG parsers from the 134 valid grammars that Grammarinator generated tests against their generated tests. Agreement indicates both parsers accepted or rejected the same inputs; disagreement indicates diverging accepted/rejected sets.

Status	#	%
Agree	48	35.8%
All passed	28	21.0%
All failed	11	8.2%
Mixed results	9	6.7%
Disagree	86	64.2%
ANTLR accepts more	65	48.5%
PEG accepts more	21	15.7%

Before running each ANTLR grammar against its examples, we try to derive a PEG from it and categorize the ANTLR grammars into the specified categories as follows:

- **Valid**: grammars that ANTLR2PEG translates into a PEG.
- **Left-recursive**: grammars that have left recursion and therefore are not supported.
- **Unsupported Features**: grammars that have unsupported ANTLR4 features, such as semantic actions, multiple lexer modes, or UTF-16.
- **Incorrect**: grammars that have errors and are not recognized by ANTLR itself.

Table 1 shows the distribution across the 262 grammars in the testing corpus. Of these, 146 are valid and produce a PEG parser. The remaining 116 cannot be translated automatically: 73 due to left recursion, 40 due to unsupported features, and 3 because they are incorrect.

We also tested each valid grammar against 100 inputs generated by *Grammarinator*. Due to internal errors, *Grammarinator* failed to

Table 4: The number of rules and conflicts detected in each ANTLR grammar listed, followed by the number of manual changes required and how many times each transformation was applied

Grammar	#Rules	#Conflicts Detected	#Manual Changes	Transformations Applied				
				Unique Path	Empty Rules	Repetitions	Prefixes	Literals
ABNF	24	1	0	4	0	1	0	0
Acme	214	16	2	0	1	14	0	4
B	35	12	1	46	0	3	0	3
Cap'n Proto	45	18	2	26	0	1	0	0
Datalog	20	18	0	13	1	0	4	0
DOT	33	4	1	1	2	0	0	3
Modelica	91	17	3	36	1	5	1	3
Pascal	178	9	4	0	3	5	1	7
PDP7	35	4	3	2	0	5	0	1
Thrift	60	3	1	9	3	0	0	3

generate tests for 12 of the 146 valid grammars. Therefore, we only use generated tests for 134 valid grammars.

For the corpus examples, where the ANTLR parser recognizes all inputs, we classify each PEG parser into one of three categories:

- **All Passed**: The PEG parser accepts all inputs.
- **All Failed**: The PEG parser rejects all inputs.
- **Mixed Results**: The PEG parser rejects some inputs and accepts others.

The results for the corpus examples are available in Table 2.

Of the 146 grammars, 62 (42.5%) yielded PEG parsers that could successfully parse all of their examples. The other 84 (57.5%) split between 54 (37.0%) parsers that could not parse any example and 30 (20.5%) that could parse some but not all of their inputs. These 84 grammars indicate cases where our transformations were not enough to yield a PEG parser that recognizes the same inputs as the ANTLR parser. However, ANTLR2PEG reports which parts of the grammar may be problematic, guiding the user to fix them manually. Therefore, with a small number of changes, grammars may go from rejecting all or some inputs to accepting all inputs. See Section 5.1 for concrete examples.

For generated inputs, we use a different classification, since *Grammarinator* occasionally produces inputs that the ANTLR4 parser itself rejects, as *Grammarinator* does not consider the lexical ordering significance of ANTLR rules. Instead of discarding these inputs, we use them to verify that the PEG parser rejects inputs that ANTLR also rejects. Therefore, we classify generated input results into two categories:

- **Agree**: The ANTLR parser and PEG parser produce the same results, whether accepted or rejected, for all generated inputs.
- **Disagree**: The ANTLR parser and PEG parser diverge on at least one input, either because ANTLR accepted it and the PEG parser rejected it, or vice versa.

Table 3 summarizes the results obtained in each category.

Of the 134 grammars, 48 (35.8%) produce ANTLR and PEG parsers that agree on all generated inputs. The remaining 86 (64.2%) disagree, falling into two different groups.

In 65 cases, the ANTLR parser accepts strictly more inputs than the PEG parser. Of these, 58 are mismatches where the PEG parser never accepts inputs that the ANTLR parser rejected. The remaining 7 are cases where each parser accepts at least one input that the other rejects.

In 21 cases, the PEG parser accepts strictly more inputs than the ANTLR parser. Of these, 17 are cases where the PEG parser accepts all the inputs that ANTLR accepts plus additional inputs. The remaining 4 are again cases where each parser accepts at least one input while the other rejects.

Cases where the PEG parser accepts more inputs than the ANTLR parser, without rejecting any input that ANTLR rejects, are due to the difference in how both parsers handle lexical tokens. In ANTLR, when a token is defined as a lexical rule, it will always be matched during the lexical phase. However, if the same string appears in a syntactic rule as a literal, rather than as a reference to a lexical rule, ANTLR will treat them distinctly. This duplicated definition leads the ANTLR parser to reject, by mistake we think, inputs where this token appears.

PEG parsers do not have this behavior because parsing is *scannerless*. Therefore, a string literal in a rule matches the same input as a lexer rule with the same definition. Consequently, the PEG parser recognizes inputs that ANTLR rejects due to this tokenization mismatch.

5.1 Deeper Analysis

We analyze a sample of 10 grammars from the 146 valid PEG grammars: 2[1] for which the ANTLR and PEG parsers accept exactly the same examples and generated inputs, and 8[2] for which the ANTLR parser accepts all examples and generated inputs while the PEG parser rejects some or all of them. Of these 8, 6 (ACME, Cap'n Proto, DOT, Modelica, PDP7, Thrift) fall in the "Mixed Results" category and 2 (B, Pascal) in the "All Failed" category of Table 2.

For each grammar, we examine four aspects: the number of rules, the number of conflicts detected by ANTLR2PEG, the number of manual changes required to make the PEG parser recognize all

[1] ABNF and Datalog.
[2] ACME, B, Cap'n Proto, DOT, Modelica, Pascal, PDP7 and Thrift.

examples and generated inputs, and the number of times each transformation from Section 3 was applied. Table 4 summarizes the results. For the "Transformations Applied," Repetitions indicates how many repetitions were transformed; Literals indicates how many rules composed of only literals were sorted. The remaining columns indicate how many times a single swap between two alternatives occurred due to the transformation specified.

In the case of 6 grammars (ACME, B, Cap'n Proto, DOT, Modelica, and Pascal), we fixed the PEG parser simply by following ANTLR2PEG reports to solve conflicts in choices. However, the grammars for PDP7 and Thrift required changes that ANTLR2PEG could not report. We discuss below how the issues related to these two grammars were fixed and why they could not be reported.

5.1.1 PDP7. For PDP7 ANTLR grammar, ANTLR2PEG reports:

```
Warning: At Rule line, choice (declarations (comment)?)
    and (comment) may match the same input
Warning: At Rule declarationRight, choice (instruction)
    and (assignment) may match the same input
Warning: At Rule declarationRight, choice (assignment)
    and (expression) may match the same input
Warning: At Rule opcode, choice ('rlpd')
    and ('rlpd') may match the same input
```

However, the real problem lies in rule *symbol*, defined as:

$$symbol \leftarrow opcode \,/\, variable \,/\, LOC \,/\, RELOC$$

The problem happens because *LOC* and *RELOC* are lexical rules defined as '.' and '..', respectively, meaning that *LOC* is a prefix of *RELOC*. Transformation 3.1.4 is not capable of solving this because *symbol* is a choice that is not exclusively composed of literals or tokens. ANTLR2PEG does not report this conflict because $\text{FIRST}(LOC) \cap \text{FIRST}(RELOC) = \emptyset$. This problem also occurs in other rules of the PDP7 grammar. After correctly putting *RELOC* before *LOC* in all choices where both rules appear, the PEG parser generated by ANTLR2PEG recognizes all examples and generated inputs.

5.1.2 Thrift. For the Thrift grammar, ANTLR2PEG reports:

```
Warning: At Rule const_value, choice (integer)
and (DOUBLE) may match the same input
Warning: At Rule integer, choice (INTEGER)
and (HEX_INTEGER) may match the same input
```

While ANTLR2PEG correctly reports the conflicts that are causing the problem, a simple reordering of the alternatives does not suffice. The rules cited above are defined as:

$$\begin{aligned}
const_value &\leftarrow integer \,/\, DOUBLE \,/\, LITERAL \,/\, \ldots \\
integer &\leftarrow INTEGER \,/\, HEX_INTEGER \\
INTEGER &\leftarrow (\text{`+'} \,/\, \text{`-'})?DIGIT+ \\
HEX_INTEGER &\leftarrow \text{`-'}?\,\text{`0x'}HEX_DIGIT+ \\
DOUBLE &\leftarrow (\text{`+'} \,/\, \text{`-'})? \\
&\qquad (DIGIT+\ (\text{`.'}\,DIGIT+)? \,/\, \text{`.'}DIGIT+) \\
&\qquad ((\text{`E'} \,/\, \text{`e'})\,INTEGER)?
\end{aligned}$$

In this initial state, the conflict occurs because *integer* can match *INTEGER*, which is a prefix of *DOUBLE*, and in rule *const_value*, the alternative *integer* comes before the alternative *DOUBLE*.

However, simply swapping these alternatives introduces a new problem. Now *DOUBLE* will always be tried before *integer*, but *DOUBLE* is a prefix of *HEX_INTEGER*.

Therefore, since *DOUBLE* is a prefix of *INTEGER* and both *INTEGER* and *DOUBLE* are prefixes of *HEX_INTEGER*, the correct order would be: *HEX_INTEGER*, *DOUBLE*, and *INTEGER*. Achieving this ordering is not possible by simply permuting the alternatives, making a grammar refactoring necessary.

5.2 Threats To Validity

The order of definition of ANTLR lexical rules is semantically significant, changing the recognized language when rules are reordered. However, *Grammarinator* does not take it into account [7] [3] when generating inputs; this causes invalid inputs to be generated by *Grammarinator*.

Beyond that, *Grammarinator* does not support the case-insensitive directive used by some grammars in the corpus and supported by ANTLR2PEG. For grammars that use these directives, the generated inputs may be valid, but they will not exercise all lexical rules, underestimating the true PEG parser coverage.

We determined the equivalence between parsers based on the tests performed. Thus, the quality of the examples available in the ANTLR repository of grammars and of the tests generated by *Grammarinator* influenced our results.

The 10 grammars used in our in-depth analysis were chosen arbitrarily while inspecting the initial results. The choice of different grammars could have influenced the results if more manual changes or undetectable changes were involved.

6 Related Work

ANTLR lexical rules use regexes, i.e., regular expressions extended with constructs such as lazy repetitions. The translation of regexes to PEGs has been discussed in different works [9, 14, 16]. The work of Ierusalimschy [9] shows how we can describe several repetitions (lazy, greedy, blind, non-blind) used by regexes tools in PEGs, and argues that we can represent in PEGs, with a clear semantics, several extensions introduced by these tools. Our translation of repetitions is based on Ierusalimschy [9]. While we focus on translating grammars, Ierusalismchy's work is more focused on using PEGs to implement search patterns similar to those used in tools like POSIX regex. In our scenario, there are several lexical and syntactical rules, while in the context of search patterns we are essentially translating a single rule.

The works of Oikawa et al. [16] and Medeiros et al. [14] present transformations to automatically convert a regex into an equivalent PEG. Again, the focus is on translating a single regex, not an entire grammar. Our translation of lexical rules is based on these works, but it also deals with other issues. For example, as the previous works do not consider the translation of a regular expression in the context of an actual parser, they did not deal with the correct matching of reserved words.

The work of Mascarenhas et al. [13] discusses the general correspondence between CFGs and PEGs, and the equivalence between some classes of grammars, such as $LL(1)$, and PEGs. The transformation that reorder the alternative of a choice that matches the

[3] https://github.com/renatahodovan/grammarinator/issues/26

empty string was described by Mascarenhas et al. [13]. However, the work of Mascarenhas et al. [13] has a more theoretical focus, so it does not discuss the implementation of a tool to get an actual parser, which would involve to deal also with the lexical elements of a language. Moreover, they do not consider grammar descriptions using EBNF, only plain BNF.

Redziejowski discussed the correspondence between a CFG and a PEG in a series of papers [20–22]. He presented a tool that indicates choices that may need rewriting to get a correct PEG parser. In a way similar to ours, the user should inspect the output of Redziejowski's tool. One novelty of our work is the usage of the grammar itself to generate input tests to validate the correspondence between the CFG-based parser and the PEG-based one. Moreover, we make a distinction between lexical and syntactical rules, which lead us to consider only whole tokens in our FIRST, set, and not individual characters.

Schmitz [24] also investigated the equivalence between CFGs and PEGs by comparing implementations of a ML parser in different tools. To solve conflicts in a choice, he proposed a solution that is based on the non-deterministic automaton generated by tools like yacc. It seems that the proposed approach was not implemented in a tool public available.

Grimm [5] and Redziejowski [19] did a good account of the conflicts they faced when manually translating Java and C grammars to parser generators based on PEGs. We tried to do in an automatic way several of the transformations (e.g., using predicates to not mismatch reserved words and identifiers) they performed manually.

There are several approaches for grammar-based testing which explore different coverage results [25]. Our use of *Grammarinator* [4, 6] was due mainly because it is a tool easily available and that accepts an ANTLR grammar as input

7 Conclusion And Future Work

ANTLR2PEG is a translation tool that makes the development of PEGs from existing ANTLR grammars easier. Additionally, the tool provides valuable feedback to correct the generated PEG parser, when necessary, after all possible automatic transformations are done. We evaluated this translation by comparing, for more than 100 ANTLR grammars, the behavior of the generated PEG parser with the corresponding ANTLR one. The parsers presented the same behavior for 42.5% of the grammars in the case of manual examples, and for 35.8% of the grammars in the case of generated tests. Furthermore, we analyzed 10 grammars in depth by verifying how much automated work was done by ANTLR2PEG when deriving a PEG parser. Of these 10 grammars, we manually fixed 8 grammars to recognize all inputs, which allowed us to measure how many manual changes were required.

Future work includes implementing alternative *backends*, so ANTLR2PEG can be used with other languages and PEG libraries such as OhmJs[4], Pest[5], and parboiled2[6]. Furthermore, manual investigation of failing test cases will be done to explore improvements to the transformations described or to propose new ones. To better evaluate the equivalence between parsers, we expect to test all grammars with at least 100 valid inputs and to generate purposely invalid inputs. This will ensure that we do not rely on accidental invalid inputs. For more robust testing, we intend to compare the parse trees generated by each parser, ensuring that they not only recognize the same input, but also recognize it in the same way.

Finally, the use of predicates to correctly match keywords, identifiers, and repetitions may lead the PEG parser to backtrack more, which may impact its performance. We plan to investigate this impact and how to mitigate it.

[4]https://github.com/ohmjs/ohm
[5]https://pest.rs/
[6]https://github.com/sirthias/parboiled2

Artifact Availability

ANTLR2PEG source code is available and licensed under the MIT license. The code with instructions on how to run the tool is available at: https://zenodo.org/records/21729424. Each ANTLR grammar used during tests belongs to ANTLR/grammars-v4 and has its own license at the start of each file.

Acknowledgements

We would like to thank the anonymous reviewers for all the valuable feedback. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.
[2] Sérgio Queiroz de Medeiros and Fabio Mascarenhas. 2018. Towards automatic error recovery in parsing expression grammars. In *Proceedings of the XXII Brazilian Symposium on Programming Languages* (Sao Carlos, Brazil) *(SBLP '18).* Association for Computing Machinery, New York, NY, USA, 3–10. doi:10.1145/3264637.3264638
[3] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04).* ACM, Venice, Italy, 111–122. doi:10.1145/964001.964011
[4] Grammarinator 2017. *ANTLRv4 grammar-based test generator.* Retrieved June 07, 2022 from https://github.com/renatahodovan/grammarinator
[5] Robert Grimm. 2006. Ambiguity. PEG mailing list. Available at https://lists.csail.mit.edu/pipermail/peg/2006-October/000051.html.
[6] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. *Grammarinator: A Grammar-Based Open Source Fuzzer.* Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/3278186.3278193
[7] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: A Grammar-Based Open Source Fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista FL USA, 2018-11-05). ACM, 45–48. doi:10.1145/3278186.3278193
[8] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation* (3 ed.). Pearson Education.
[9] Roberto Ierusalimschy. 2009. A text pattern-matching tool based on Parsing Expression Grammars. *Softw. Pract. Exper.* 39, 3 (March 2009), 221–258.
[10] Roberto Ierusalimschy. 2009. A Text Pattern-Matching Tool Based on Parsing Expression Grammars. In *Proceedings of the 2009 ACM Symposium on Applied Computing.* ACM, 687–694.
[11] JavaCC Contributors. 2025. *JavaCC: The Java Compiler Compiler.* GitHub. https://javacc.github.io/javacc/ Accessed: 2025-12-30.
[12] Stephen C. Johnson. 1975. *Yacc: Yet Another Compiler-Compiler.* Computing Science Technical Report 32. Bell Laboratories, Murray Hill, NJ.
[13] Fabio Mascarenhas, Sérgio Medeiros, and Roberto Ierusalimschy. 2014. On the Relation between Context-Free Grammars and Parsing Expression Grammars. 89 (2014), 235–250. doi:10.1016/j.scico.2014.01.012
[14] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. 2014. From regexes to parsing expression grammars. *Science of Computer Programming* 93 (2014), 3–18. doi:10.1016/j.scico.2012.11.006 Special Issue with Selected Papers from the Brazilian Symposium on Programming Languages (SBLP 2011).
[15] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. 2014. Left recursion in Parsing Expression Grammars. *Science of Computer Programming*

96 (2014), 177–190. doi:10.1016/j.scico.2014.01.013 Selected and extended papers of the Brazilian Symposium on Programming Languages 2012 (SBLP 2012).

[16] Marcelo Oikawa, Roberto Ierusalimschy, and Ana Moura. 2010. Converting regexes to Parsing Expression Grammars. In *SBLP '10: Proceedings of the 14th Brazilian Symposium on Programming Languages* (Salvador, Brazil).

[17] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.

[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*. ACM, 579–598. doi:10.1145/2660193.2660202

[19] Roman R Redziejowski. 2010. Mouse sampe grammars for Java and C. http://www.romanredz.se/Mouse/index.htm. Accessed 11 June 2022.

[20] Roman R Redziejowski. 2013. From ebnf to peg. *Fundamenta Informaticae* 128, 1-2 (2013), 177–191.

[21] Roman R Redziejowski. 2014. More about converting BNF to PEG. *Fundamenta Informaticae* 133, 2-3 (2014), 257–270.

[22] Roman R. Redziejowski. 2018. Trying to Understand PEG. *Fundam. Inf.* 157, 4 (jan 2018), 463–475. doi:10.3233/FI-2018-1638

[23] Roman R Redziejowski. 2021. Left recursion by recursive ascent. In *Proceedings of the International Workshop on Concurrency, Specification and Programming CS&P'2021*. 72–82. http://ceur-ws.org/Vol-2951

[24] Sylvain Schmitz. 2006. *Modular syntax demands verification*. Technical Report. LABORATOIRE I3S, UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS.

[25] Phillip van Heerden, Moeketsi Raselimo, Konstantinos Sagonas, and Bernd Fischer. 2020. Grammar-Based Testing for Little Languages: An Experience Report with Student Compilers. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering* (Virtual, USA) *(SLE 2020)*. Association for Computing Machinery, New York, NY, USA, 253–269. doi:10.1145/3426425.3426946

[26] Alessandro Warth, James R. Douglass, and Todd Millstein. 2008. Packrat Parsers Can Support Left Recursion. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (San Francisco, California, USA) *(PEPM '08)*. Association for Computing Machinery, New York, NY, USA, 103–110. doi:10.1145/1328408.1328424