Science of Computer Programming 187 (2020) 102373

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Automatic syntax error reporting and recovery in parsing expression grammars

Sérgio Queiroz de Medeiros[a,*], Gilney de Azevedo Alvez Junior[b], Fabio Mascarenhas[c]

[a] School of Science and Technology, UFRN, Natal, Brazil
[b] Institute Digital Metropolis, UFRN, Natal, Brazil
[c] Department of Computer Science, UFRJ, Rio de Janeiro, Brazil

A R T I C L E I N F O

Article history:
Received 13 February 2019
Received in revised form 17 September 2019
Accepted 12 November 2019
Available online 27 November 2019

Keywords:
Parsing expression grammars
Labeled failures
Error reporting
Error recovery

A B S T R A C T

Error recovery is an essential feature for a parser that should be plugged in Integrated Development Environments (IDEs), which must build Abstract Syntax Trees (ASTs) even for syntactically invalid programs in order to offer features such as automated refactoring and code completion.

Parsing Expressions Grammars (PEGs) are a formalism that naturally describes recursive top-down parsers using a restricted form of backtracking. Labeled failures are a conservative extension of PEGs that adds an error reporting mechanism for PEG parsers, and these labels can also be associated with recovery expressions to provide an error recovery mechanism. These expressions can use the full expressivity of PEGs to recover from syntactic errors.

Manually annotating a large grammar with labels and recovery expressions can be difficult. In this work, we present two approaches, Standard and Unique, to automatically annotate a PEG with labels, and to build their corresponding recovery expressions. The Standard approach annotates a grammar in a way similar to manual annotation, but it may insert labels incorrectly, while the Unique approach is more conservative to annotate a grammar and does not insert labels incorrectly.

We evaluate both approaches by using them to generate error recovering parsers for four programming languages: Titan, C, Pascal and Java. In our evaluation, the parsers produced using the Standard approach, after a manual intervention to remove the labels incorrectly added, gave an acceptable recovery for at least 70% of the files in each language. By it turn, the acceptable recovery rate of the parsers produced via the Unique approach, without the need of manual intervention, ranged from 41% to 76%.

* Corresponding author.
E-mail addresses: sergiomedeiros@ect.ufrn.br (S. Queiroz de Medeiros), gilneyjnr@gmail.com (G. de Azevedo Alvez Junior), mascarenhas@ufrj.br (F. Mascarenhas).

https://doi.org/10.1016/j.scico.2019.102373
0167-6423/© 2019 Elsevier B.V. All rights reserved.

1. Introduction

Integrated Development Environments (IDEs) often require parsers that can recover from syntax errors and build syntax trees even for syntactically invalid programs, in other to conduct further analyses necessary for IDE features such as automated refactoring and code completion.

Parsing Expression Grammars (PEGs) [1] are a formalism used to describe the syntax of programming languages, as an alternative for Context-Free Grammars (CFGs). We can view a PEG as a formal description of a recursive top-down parser for the language it describes. PEGs have a concrete syntax based on the syntax of regexes, or extended regular expressions. Unlike CFGs, PEGs avoid ambiguities in the definition of the grammar's language by construction, due to the use of an ordered choice operator.

The ordered choice operator naturally maps to restricted (or local) backtracking in a recursive top-down parser. The alternatives of a choice are tried in order; when the first alternative recognizes an input prefix, no other alternative of this choice is tried, but when an alternative fails to recognize an input prefix, the parser backtracks to the same input position it was before trying this alternative and then tries the next one.

A naive interpretation of PEGs is problematic when dealing with inputs with syntactic errors, as a failure during parsing an input is not necessarily an error, but can be just an indication that the parser should backtrack and try another alternative. Labeled failures [2,3] are a conservative extension of PEGs that address this problem of error reporting in PEGs by using explicit error labels, which are distinct from a regular failure. We throw a label to signal an error during parsing, and each label can then be tied to a specific error message.

We can leverage the same labels to add an error recovery mechanism, by attaching a recovery expression to each label. This expression is just a regular parsing expression, and it usually skips the erroneous input until reaching a synchronization point, while producing a dummy AST node [4,5].

Labeled failures produce good error messages and error recovery, but they can add a considerable annotation burden in large grammars, as each point where we want to signal and recover from a syntactic error must be explicitly marked.

In a previous work [6], we presented the Algorithm Standard, which automatically annotates a PEG with labels and builds their corresponding recovery expressions. We evaluated the use of such algorithm to build an error recovering parser for the Titan programming language.

This paper extends the previous one by also evaluating the use of Algorithm Standard to build error recovering parsers for C, Pascal and Java.

As pointed out in [6], Algorithm Standard may add some labels incorrectly, which would prevent the parser from recognizing syntactically valid programs.

In this paper we try to address this issue by proposing the Algorithm Unique, which inserts labels in a more conservative way. The use of Algorithm Unique avoids the problem of adding labels incorrectly, although it inserts less labels than Algorithm Standard.

Overall, our experiments show that Algorithm Standard can be used to produce error recovering parsers with the help of manual intervention, which was small in case of our Titan, C, and Pascal grammars, and more significant in case of Java. By its turn, Algorithm Unique can be used to automatically generate functional error recovering parsers, whose error recovery quality is lower when compared to the parsers got via Algorithm Standard.

The remainder of this paper is organized as follows: Section 2 discusses error recovery in PEGs using labeled failures and recovery expressions; Section 3 shows Algorithm Standard, which automatically annotates a PEG with labels and associates a recovery expression to each label; Section 4 evaluates the use of Algorithm Standard to annotate the grammars of four programming languages: Titan, C, Pascal, and Java; Section 5 discusses conservative approaches to insert labels and presents Algorithm Unique, which inserts only correct labels; Section 6 compares the use of both algorithms to annotate Titan, C, Pascal and Java grammars; Section 7 discusses related work on error reporting and error recovery; finally, Section 8 gives some concluding remarks.

2. Error recovery in PEGs with labeled failures

In this section we present an introduction to labeled PEGs and discuss how to build an error recovery mechanism for PEGs by attaching a recovery expression to each labeled failure.

A labeled PEG G is a tuple $(V, T, P, L, R, \texttt{fail}, p_S)$, where V is a finite set of non-terminals, T is a finite set of terminals, P is a total function from non-terminals to parsing expressions, L is a finite set of labels, R is a function from labels to parsing expressions, $\texttt{fail} \notin L$ is a failure label, and p_S is the initial parsing expression. We will use the term recovery expression when referring to the parsing expression associated with a given label. We will assume that $V = V_{Lex} \cup V_{Syn}$, where V_{Lex} is the set of non-terminals that match lexical elements, also known as tokens, and V_{Syn} represents the non-terminals that match syntactical elements. When describing the PEG for a given language, we will use names in uppercase for the lexical non-terminals. From now on, unless otherwise noted, we will use PEG as synonymous to labeled PEG.

We describe the function P as a set of rules of the form $A \leftarrow p$, where $A \in V$ and p is a parsing expression. A parsing expression p, when applied to an input string s, either succeeds or fails. When the matching of p succeeds, it consumes a prefix of the input and returns the remaining suffix, and when it fails, it produces a label, associated with an input suffix.

$$\textbf{Empty}\quad \frac{}{G[\varepsilon]\ R\ x \overset{\text{PEG}}{\rightsquigarrow} x}\ (\textbf{empty.1}) \qquad \textbf{Non-terminal}\quad \frac{G[P(A)]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} X}{G[A]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} X}\ (\textbf{var.1})$$

$$\textbf{Terminal}\quad \frac{}{G[a]\ R\ ax \overset{\text{PEG}}{\rightsquigarrow} x}\ (\textbf{term.1}) \quad \frac{b \neq a}{G[b]\ R\ ax \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail},\ ax)}\ (\textbf{term.2}) \quad \frac{}{G[a]\ R\ \varepsilon \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail},\ \varepsilon)}\ (\textbf{term.3})$$

$$\textbf{Sequence}\quad \frac{G[p_1]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} y \quad G[p_2]\ R\ y \overset{\text{PEG}}{\rightsquigarrow} X}{G[p_1\ p_2]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} X}\ (\textbf{seq.1}) \quad \frac{G[p_1]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} (f,\ y)}{G[p_1\ p_2]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} (f,\ y)}\ (\textbf{seq.2})$$

$$\textbf{Ordered Choice}\quad \frac{G[p_1]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} y}{G[p_1\ /\ p_2]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} y}\ (\textbf{ord.1}) \quad \frac{G[p_1]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} (l,\ y) \quad l \neq \texttt{fail}}{G[p_1\ /\ p_2]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} (l,\ y)}\ (\textbf{ord.2})$$

$$\frac{G[p_1]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail},\ y) \quad G[p_2]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} X}{G[p_1\ /\ p_2]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} X}\ (\textbf{ord.3})$$

$$\textbf{Repetition}\quad \frac{G[p]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail},\ y)}{G[p*]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} xy}\ (\textbf{rep.1}) \quad \frac{G[p]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} (l,\ y) \quad l \neq \texttt{fail}}{G[p*]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} (l,\ y)}\ (\textbf{rep.2})$$

$$\frac{G[p]\ R\ xyz \overset{\text{PEG}}{\rightsquigarrow} yz \quad G[p*]\ R\ yz \overset{\text{PEG}}{\rightsquigarrow} z}{G[p*]\ R\ xyz \overset{\text{PEG}}{\rightsquigarrow} z}\ (\textbf{rep.3})$$

$$\textbf{Negative Predicate}\quad \frac{G[p]\ \{\}\ xy \overset{\text{PEG}}{\rightsquigarrow} (f,\ y)}{G[!p]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} xy}\ (\textbf{not.1}) \quad \frac{G[p]\ \{\}\ xy \overset{\text{PEG}}{\rightsquigarrow} y}{G[!p]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail},\ xy)}\ (\textbf{not.2})$$

$$\textbf{Recovery}\quad \frac{l \notin Dom(R)}{G[\Uparrow^l]\ R\ x \overset{\text{PEG}}{\rightsquigarrow} (l,\ x)}\ (\textbf{throw.1}) \quad \frac{G[R(l)]\ R\ x \overset{\text{PEG}}{\rightsquigarrow} X}{G[\Uparrow^l]\ R\ x \overset{\text{PEG}}{\rightsquigarrow} X}\ (\textbf{throw.2})$$

Fig. 1. Semantics of PEGs with labeled failures.

The abstract syntax of parsing expressions is as follows, where p, p_1 and p_2 are parsing expressions: ε represents the empty string, $a \in T$ denotes a terminal, $A \in V$ represents a non-terminal, $p_1 p_2$ is a concatenation, $p_1\ /\ p_2$ is an ordered choice, $p*$ indicates zero or more repetitions, $!p$ is a negative predicate, and $\Uparrow^l$ throws a label $l \in L$.

Fig. 1 presents the semantics of labeled PEGs with error recovery as a set of inference rules for a $\overset{\text{PEG}}{\rightsquigarrow}$ function. The notation $G[p]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ represents a successful matching of the parsing expression p in the context of a PEG G against the subject xy with a map R from labels to recovery expressions, consuming x and leaving the suffix y. By its turn, the notation $G[p]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} (f,\ y)$ represents an unsuccessful match of p, where label $f \in L \cup \{\texttt{fail}\}$, was thrown when trying to match the suffix y. We will usually use f to represent a label in the set $L \cup \{\texttt{fail}\}$, and l to represent a label in L. The notation $G[p]\ R\ xy \overset{\text{PEG}}{\rightsquigarrow} X$ indicates that the matching result can be either y or $(f,\ y)$.

The semantics given here is essentially the same semantics for PEGs with labels presented in previous work [5,4], with two simplifications: neither we are tracking the farthest failure, nor keeping a list of the errors that occurred during a match. We did this to make more amenable a formal discussion about the correct insertion of labels.

We can see in Fig. 1 that failing to match a terminal (rules **term.2** and **term.3**) gives us the label `fail`, while a throw expression (rules **throw.1** and **throw.2**) may give us a label different from `fail`. The recovery map R is simply passed along. The exceptions are the rules for the syntactic predicate and for throwing labels.

A label $l \neq \texttt{fail}$ thrown by $\Uparrow^l$ cannot be caught by an ordered choice or a repetition (rules **ord.2** and **rep.2**), so it indicates an actual error during parsing, while `fail` is a regular failure and it indicates that the parser should backtrack. In the original formalization of PEGs [1], there is only label `fail`, thus the parser always tries to backtrack after failing to match a parsing expression.

The lookahead operator ! captures any label and turns it into a success (rule **not.1**), while turning a success into a `fail` label (rule **not.2**). In both rules we used an empty recovered map to make sure that errors are not recovered inside the predicate. The rationale is that errors inside a syntactic predicate are expected and not actually syntactic errors in the input.

$$\begin{aligned}
\mathit{prog} &\leftarrow \texttt{PUBLIC CLASS NAME LCUR PUBLIC STATIC VOID MAIN} \\
&\qquad \texttt{LPAR STRING LBRA RBRA NAME RPAR}\ \mathit{blockStmt}\ \texttt{RCUR EOF} \\
\mathit{blockStmt} &\leftarrow \texttt{LCUR}\ (\mathit{stmt}) *\ \texttt{RCUR} \\
\mathit{stmt} &\leftarrow \mathit{ifStmt}\ /\ \mathit{whileStmt}\ /\ \mathit{printStmt}\ /\ \mathit{decStmt}\ /\ \mathit{assignStmt}\ /\ \mathit{blockStmt} \\
\mathit{ifStmt} &\leftarrow \texttt{IF LPAR}\ \mathit{exp}\ \texttt{RPAR}\ \mathit{stmt}\ (\texttt{ELSE}\ \mathit{stmt}\ /\ \varepsilon) \\
\mathit{whileStmt} &\leftarrow \texttt{WHILE LPAR}\ \mathit{exp}\ \texttt{RPAR}\ \mathit{stmt} \\
\mathit{decStmt} &\leftarrow \texttt{INT NAME}\ (\texttt{ASSIGN}\ \mathit{exp}\ /\ \varepsilon)\ \texttt{SEMI} \\
\mathit{assignStmt} &\leftarrow \texttt{NAME ASSIGN}\ \mathit{exp}\ \texttt{SEMI} \\
\mathit{printStmt} &\leftarrow \texttt{PRINTLN LPAR}\ \mathit{exp}\ \texttt{RPAR SEMI} \\
\mathit{exp} &\leftarrow \mathit{relExp}\ (\texttt{EQ}\ \mathit{relExp})* \\
\mathit{relExp} &\leftarrow \mathit{addExp}\ (\texttt{LT}\ \mathit{addExp})* \\
\mathit{addExp} &\leftarrow \mathit{mulExp}\ ((\texttt{PLUS}\ /\ \texttt{MINUS})\ \mathit{mulExp})* \\
\mathit{mulExp} &\leftarrow \mathit{atomExp}\ ((\texttt{TIMES}\ /\ \texttt{DIV})\ \mathit{atomExp})* \\
\mathit{atomExp} &\leftarrow \texttt{LPAR}\ \mathit{exp}\ \texttt{RPAR}\ /\ \texttt{NUMBER}\ /\ \texttt{NAME}
\end{aligned}$$

Fig. 2. A PEG for a tiny subset of Java.

Rule **throw.1** is related to error reporting, while rule **throw.2** is where error recovery happens. $R(l)$ denotes the recovery expression associated with the label l. When a label l is thrown we check if R has a recovery expression associated with it. If it does not (**throw.1**), the matching result is l plus the current input, and this error is propagated, so parsing finishes after reaching the first syntactical error.

If label l has a recovery expression $R(l)$ (rule **throw.2**), we try to match the current input by using $R(l)$. As $R(l)$ is a regular parsing expression, its matching may succeed, which essentially resumes regular parsing, or may fail, which may finish the parsing or not (the parser can still recover from this second error).

When a PEG does not throw labels via expression $\Uparrow^l$, we say it is an unlabeled PEG, as the following definition states:

Definition 1 *(Unlabeled PEG). A PEG $G = (V, T, P, L, R, \texttt{fail}, p_S)$ is unlabeled when $\forall A \in V$ we have that expression $\Uparrow^l$ does not appear in $P(A)$.*

In an unlabeled PEG G, the function R is not relevant, as no label different from $\texttt{fail}$ is thrown and thus rules **throw.1** and **throw.2** will not be used. In this case, the result of a matching is more specific, as stated by the following lemma, where x' and x'' are suffixes of x:

Lemma 1. *Given an unlabeled PEG $G = (V, T, P, L, R, \texttt{fail}, p_S)$, $\forall A \in V$, let p be a subexpression of $P(A)$, either $G[p]\ R\ x \overset{PEG}{\rightsquigarrow} x'$, or $G[p]\ R\ x \overset{PEG}{\rightsquigarrow} (\texttt{fail}, x'')$.*

Proof. By induction on the heights of the proof trees for $G[p]\ R\ x \overset{\text{PEG}}{\rightsquigarrow} x'$ and $G[p]\ R\ x \overset{\text{PEG}}{\rightsquigarrow} (\texttt{fail}, x'')$. As G is an unlabeled PEG, we do not have a case related to expression $\Uparrow^l$, which would involve rules **throw.1** and **throw.2**. The proof related to the other expressions is straightforward. □

Below, we discuss an example that illustrates how to deal with syntax errors in PEGs by using labeled failures and recovery expressions.

2.1. Handling syntax errors in PEGs

In Fig. 2 we can see a PEG for a tiny subset of Java, where lexical rules (shown in uppercase) have been elided. While simple (this PEG is almost equivalent to an LL(1) CFG), this subset is a good starting point to discuss error recovery in the context of PEGs.

To get a parser with error recovery, we first need to have a parser that correctly reports errors. One popular error reporting approach for PEGs is to report the farthest failure position [7,3], an approach that is supported by PEGs with labels [4]. However, the use of the farthest failure position makes it harder to recover from an error, as the error is only known after parsing finishes and all the parsing context at the moment of the error has been lost. Because of this, we will focus on using labeled failures for error reporting in PEGs.

$$\begin{aligned}
prog &\leftarrow \texttt{PUBLIC CLASS NAME LCUR PUBLIC STATIC VOID MAIN} \\
&\quad \texttt{LPAR STRING LBRA RBRA NAME RPAR}\ blockStmt\ \texttt{RCUR EOF} \\
blockStmt &\leftarrow \texttt{LCUR}\ (stmt)*\ [\texttt{RCUR}]^{\texttt{rcurlyblk}} \\
stmt &\leftarrow ifStmt\ /\ whileStmt\ /\ printStmt\ /\ decStmt\ /\ assignStmt\ /\ blockStmt \\
ifStmt &\leftarrow \texttt{IF}\ [\texttt{LPAR}]^{\texttt{lparif}}\ [exp]^{\texttt{condif}}\ [\texttt{RPAR}]^{\texttt{rparif}}\ [stmt]^{\texttt{thenstmt}}\ (\texttt{ELSE}\ [stmt]^{\texttt{elsestmt}}\ /\ \varepsilon) \\
whileStmt &\leftarrow \texttt{WHILE}\ [\texttt{LPAR}]^{\texttt{lparwhile}}\ [exp]^{\texttt{condwhile}}\ [\texttt{RPAR}]^{\texttt{rparwhile}}\ [stmt]^{\texttt{bodywhile}} \\
decStmt &\leftarrow \texttt{INT}\ [\texttt{NAME}]^{\texttt{namedec}}\ (\texttt{ASSIGN}\ [exp]^{\texttt{expdec}}\ /\ \varepsilon)\ [\texttt{SEMI}]^{\texttt{semidec}} \\
assignStmt &\leftarrow \texttt{NAME}\ [\texttt{ASSIGN}]^{\texttt{assign}}\ [exp]^{\texttt{rval}}\ [\texttt{SEMI}]^{\texttt{semiassign}} \\
printStmt &\leftarrow \texttt{PRINT}\ [\texttt{LPAR}]^{\texttt{lparprint}}\ [exp]^{\texttt{expprint}}\ [\texttt{RPAR}]^{\texttt{rparprint}}\ [\texttt{SEMI}]^{\texttt{semiprint}} \\
exp &\leftarrow relExp\ (\texttt{EQ}\ [relExp]^{\texttt{relexp}})* \\
relExp &\leftarrow addExp\ (\texttt{LT}\ [addExp]^{\texttt{addexp}})* \\
addExp &\leftarrow mulExp\ ((\texttt{PLUS}\ /\ \texttt{MINUS})\ [mulExp]^{\texttt{mulexp}})* \\
mulExp &\leftarrow atomExp\ ((\texttt{TIMES}\ /\ \texttt{DIV})\ [atomExp]^{\texttt{atomexp}})* \\
atomExp &\leftarrow \texttt{LPAR}\ [exp]^{\texttt{parexp}}\ [\texttt{RPAR}]^{\texttt{rparexp}}\ /\ \texttt{NUMBER}\ /\ \texttt{NAME}
\end{aligned}$$

Fig. 3. A PEG with labels for a small subset of Java.

```
1  public class Example {
2    public static void main(String[] args) {
3      int n = 5;
4      int f = 1;
5      while(0 < n {
6        f = f * n;
7        n = n - 1
8      }
9      System.out.println(f);
10   }
11 }
```

Fig. 4. A Java program with syntax errors.

We need to annotate our original PEG with labels, which indicate the points where we can signal a syntactical error. Fig. 3 annotates the PEG of Fig. 2 (except for the *prog* rule). The expression $[p]^l$ is syntactic sugar for $(p\ /\ \Uparrow^l)$. It means that if the matching of p fails we should throw label l to signal an error.

The strategy we used to annotate the grammar was to annotate every symbol (terminal or non-terminal) in the right-hand side of a production that should not fail, as a failure would just make the whole parser either fail or not consume the input entirely. For a nearly LL(1) grammar, like the one in our example, that means all symbols in the right-hand side of a production, except the first one. We apply the same strategy when the right-hand side has a choice or a repetition as a subexpression.

We can associate each label with an error message. For example, in rule *whileStmt* the label `rparwwhile` is thrown when we fail to match a closing parenthesis, so we could attach an error message like "`missing ')' in while`" to this label. Dynamically, when the matching of `RPAR` fails and we throw `rparwhile`, we could enhance this message with information related to the input position where this error happened.

Let us consider the example Java program from Fig. 4, which has two syntax errors: a missing '**)**' at line 5, and a missing '**;**' at the end of line 7. For this program, a parser based on the labeled PEG from Fig. 3 would give us a message like:

```
factorial.java:5: syntax error, missing ')' in while
```

The second error will not be reported because the parser did not recover from the first one, since `rparwhile` still has no recovery expression associated with it.

The recovery expression p_r of an label l matches the input from the point where l was thrown. If p_r succeeds then regular parsing is resumed as if the label had not been thrown. Usually p_r should just skip part of the input until is safe to resume parsing. In rule *whileStmt*, we can see that after the '**)**' we expect to match a *stmt*, so the recovery expression of label `rparwhile` could skip the input until it encounters the beginning of a statement.

In order to define a safe input position to resume parsing, we will use the classical $FIRST$ and $FOLLOW$ sets. A more detailed discussion about $FIRST$ and $FOLLOW$ sets in the context of PEGs can be found in other papers [8–10].

With the help of these sets, we can define the following recovery expression for `rparwhile`, where *eatToken* is a rule that matches an input token:

$$(!\texttt{FIRST(stmt)}\ eatToken)*$$

Now, when label `rparwhile` is thrown, its recovery expression matches the input until it finds the beginning of a statement, and then regular parsing resumes.

The parser will now also throw label `semiassign` and report the second error, the missing semicolon at the end of line 7. In case `semiassign` has an associated recovery expression, this expression will be used to try to resume regular parsing again.

Even our toy grammar has 26 distinct labels, each needing a recovery expression to recover from all possible syntactic errors. While most of these expressions are trivial to write, this is still burdensome, and for real grammars the problem is compounded by the fact that they can easily need a small multiple of this number of labels. In the next section, we present an approach to automatically annotate a grammar with labels and recovery expressions in order to provide a better starting point for larger grammars.

3. Automatic insertion of labels and recovery expressions

The use of labeled failures trades better precision in error messages, and the possibility of having error recovery, for an increased annotation burden, as the grammar writer is responsible for annotating the grammar with the appropriate labels. In this section, we show how this process can be partially automated.

To automatically annotate a grammar, we need to determine when it is safe to signal an error: we should only throw a label after expression p fails if that failure $\underline{\text{always}}$ implies that the whole parse will fail or not consume the input entirely, so it is useless to backtrack.

This is easy to determine when we have a nearly $LL(1)$ grammar, as is the case with the PEG from Fig. 2. As we mentioned in Section 2, for an $LL(1)$ grammar the general rule is that we should annotate every symbol (terminal or non-terminal) in the right-hand side of a production after consuming at least one token, which in general leads to annotating every symbol in the right-hand side of a production except the first one.

Although many PEGs are not $LL(1)$, we can use this approach to annotate what would be the $LL(1)$ parts of a non-$LL(1)$ grammar. We will discuss some limitations of this approach in the next section, when we evaluate its application to annotate PEG-based parsers for the programming languages Titan, C, Pascal and Java.

While annotating a PEG with labels we can add an automatically generated recovery expression for each label, based on the tokens that could follow it. We assume the tokens of a grammar are described by the non-terminals $A \in V_{Lex}$. Moreover, we also assume that at most one non-terminal $A \in V_{Lex}$ matches a prefix of the current input, as stated by the following definition:

Definition 2 *(Unique token prefix). An unlabeled PEG $G = (V_{Lex} \cup V_{Syn}, T, P, L, R, \texttt{fail}, p_S)$ has the unique token prefix property iff $G[A]\ R\ axy \overset{PEG}{\rightsquigarrow} y$, where $A \in V_{Lex}$, then $\forall B \in V_{Lex}$, where $B \neq A$, we have that $G[B]\ R\ axy \overset{PEG}{\rightsquigarrow} (\texttt{fail}, x')$, where x' is a suffix of axy.*

In the above definition, we assumed an unlabeled PEG to make sure we would not recover from an error when matching a lexical non-terminal. Alternatively, we could have considered above a labeled PEG with an empty recovery function.

By assuming a grammar with the unique token prefix property we did not have to worry about which lexical non-terminal should come first in a choice (e.g., an alternative that matches "=" can come before one that matches "=="). Such property is useful, for example, when automatically computing a choice with the tokens that a recovery expression should match. The unique token prefix property can be easily achieved with the help of predicates. For example, we could define a non-terminal to match input "=" as $ATRIB \leftarrow$ '=' !'='.

Moreover, when a PEG G has the unique token prefix property the sequence of tokens matched for a given input is unique, as stated below, where we assumed, as previously, an unlabeled PEG to avoid recovering in case of an error:

Lemma 2 *(Unique token sequence). Given an unlabeled PEG $G = (V_{Lex} \cup V_{Syn}, T, P, L, R, \texttt{fail}, p_S)$, with the unique token prefix property, and a subject w, the sequence in which the lexical non-terminals in V_{Lex} match w is unique.*

Proof. By contradiction. Assume the sequence is not unique. This implies that for some suffix ax of w we would have that $G[A]\ R\ ax \overset{\text{PEG}}{\rightsquigarrow} x'$ and $G[B]\ R\ ax \overset{\text{PEG}}{\rightsquigarrow} x''$, where $A, B \in V_{Lex}$, which is not possible given that G has the unique token prefix property. $\square$

Below, we present Algorithm Standard, which automatically adds labels and recovery expressions to a PEG $G = (V_{Lex} \cup V_{Syn}, T, P, L, R, \texttt{fail}, p_S)$. We assume that all occurrences of $FIRST$ and $FOLLOW$ in Algorithm Standard give

Algorithm Standard Adding Labels and Recovery Expressions to a PEG

```
 1: function ANNOTATE($G$)
 2:     $G' \leftarrow G$
 3:     for $A \in V_{Syn}$ do
 4:         $G'(A) \leftarrow labexp(G(A), \textbf{false}, FOLLOW(A))$
 5:     return $G'$
 6:
 7: function LABEXP($p, seq, flw$)
 8:     if $p = A$ and $\varepsilon \notin FIRST(A)$ and $seq$ then
 9:         return $addlab(p, flw)$
10:     else if $p = p_1\ p_2$ then
11:         $p_x \leftarrow labexp(p_1, seq, calck(p_2, flw))$
12:         $p_y \leftarrow labexp(p_2, seq$ or $\varepsilon \notin FIRST(p_1), flw)$
13:         return $p_x\ p_y$
14:     else if $p = p_1\ /\ p_2$ then
15:         $p_x \leftarrow p_1$
16:         if $FIRST(p1) \cap calck(p2, flw) = \emptyset$ then
17:             $p_x \leftarrow labexp(p_1, \textbf{false}, flw)$
18:         $p_y \leftarrow labexp(p_2, \textbf{false}, flw)$
19:         if $seq$ and $\varepsilon \notin FIRST(p_1\ /\ p_2)$ then
20:             return $addlab(p_x\ /\ p_y, flw)$
21:         else
22:             return $p_x\ /\ p_y$
23:     else if $p = p_1*$ and $FIRST(p_1) \cap flw = \emptyset$ then
24:         return $labexp(p_1, \textbf{false}, FIRST(p_1) \cup flw)*$
25:     else
26:         return $p$
27:
28: function CALCK($p, flw$)
29:     if $\varepsilon \in FIRST(p)$ then
30:         return $(FIRST(p) - \{\varepsilon\}) \cup flw$
31:     else
32:         return $FIRST(p)$
33:
34: function ADDLAB($p, flw$)
35:     $l \leftarrow$ newLabel()
36:     $R'(l) \leftarrow (!flw\ eatToken)*$
37:     return $[p]^l$
```

their results regarding to the grammar G passed to function *annotate*. We also assume grammar G' from function *annotate* is available in function *addlab*.

Function `annotate` (lines 1–5) generates a new annotated grammar G' from a grammar G. It uses `labexp` (lines 7–26) to annotate the right-hand side, a parsing expression, of each syntactical rule of grammar G. The auxiliary function `calck` (lines 28–32) is used to update the $FOLLOW$ set associated with a parsing expression. By its turn, the auxiliary function `addlab` (lines 34–37) receives a parsing expression p to annotate and its associated $FOLLOW$ set flw. Function `addlab` associates a label l to p and also builds a recovery expression for l based on flw. The expression $eatToken$, which matches an input token, can be generated from the lexical rules of G. We assume G has the unique prefix property when computing $eatToken$ automatically.

Algorithm Standard annotates every right-hand side, instead of going top-down from the root, to not be overly conservative and fail to annotate non-terminals reachable only from non-LL(1) choices but which themselves might be LL(1). We will see in Section 4 that this has the unfortunate result of sometimes changing the language being parsed, which is the major shortcoming of Algorithm Standard.

Function `labexp` has three parameters. The first one, p, is a parsing expression that we will try to annotate. The second parameter, *seq*, is a boolean value that indicates whether the current concatenation consumes at least one token before p or not. Finally, the parameter flw represents the $FOLLOW$ set associated with p. Let us now discuss how `labexp` tries to annotate p.

When p is a non-terminal expression and it is part of a concatenation that already matched at least one token (lines 8–9), then we associate a new label with p. In case p represents a non-terminal but *seq* is not **true**, we will just return p itself (lines 25–26). In line 8, we also test whether A matches the empty string or not. This avoids polluting the grammar with labels which will never be thrown, since a parsing expression that matches the empty string does not fail.

In case of a concatenation $p_1\ p_2$ (lines 10–13), we try to annotate p_1 and p_2 recursively. To annotate p_1 we use an updated $FOLLOW$ set, and to annotate p_2 we set its parameter *seq* to **true** whenever *seq* is already **true** or p_1 does not match the empty string.

In case of a choice $p_1\ /\ p_2$ (lines 14–22), we annotate p_2 recursively and in case the choice is disjoint we also annotate p_1 recursively. In both cases, we pass the value **false** as the second parameter of `labexp`, since failing to match the first symbol of an alternative should not signal an error. When *seq* is **true**, we associate a label to the whole choice when it does not match the empty string.

In case p is a repetition p_1* (lines 23–24), we can annotate p_1 if we have a disjoint repetition, i.e., if there is no intersection between $FIRST(p_1)$ and flw. When annotating p_1 we pass **false** as the second parameter of `labexp` because failing to match the first symbol of a repetition should not signal an error.

Our concrete implementation of Algorithm Standard also adds labels in case of repetitions of the form p_1+, which should match p_1 at least once, and $p_1?$, which should match p_1 at most once. As these cases are similar to the case of p_1*, we will not discuss them here.

Given the PEG from Fig. 2, function `annotate` would give us the grammar presented in Fig. 3 (as previously, we are not taking rule *prog* into consideration), with the exception of the annotation $[stmt]^{\texttt{elsestmt}}$. Label `elsestmt` was not inserted at this point because token `ELSE` may follow the choice `ELSE` *stmt* / ε, so this choice is not disjoint (the well-known *dangling else* problem). In Fig. 3, we associated the label `elsestmt` to *stmt*. This indicates that an `else` must be associated with the nearby *if* statement.

It is trivial to change the algorithm to leave any existing labels and recovery expressions in place, or to add recovery expressions to any labels that are already present but do not have recovery expressions.

After applying Algorithm Standard to automatically insert labels, a grammar writer can later add (or remove) labels and their associated recovery expressions. We discuss more about this on the next section, where we evaluate the use of Algorithm Standard to add error recovery for the parsers of several programming languages.

4. Evaluating Algorithm Standard

To evaluate Algorithm Standard, we built PEG parsers for the programming languages Titan, C, Pascal and Java. To build such parsers we used `LPegLabel`,[1] a tool that implements the semantics of PEGs with labeled failures, and `pegparser`,[2] which automatically adds labels and recovery expressions to a PEG. When building the parsers, we focused on the syntactical rules, so we have omitted or simplified some lexical rules.

For each language, we first wrote an unlabeled version of the grammar based on some reference grammar. We have tried to follow the reference grammar syntactic structure to avoid a bias that could favor our algorithm. We used a set of syntactically valid and invalid programs to validate each parser.

Given an unlabeled grammar, we used `pegparser` to got an automatically annotated grammar following Algorithm Standard, with a recovery expression associated to each label. We will use the term *generated* when referring to this annotated grammar.

We will compare the generated grammar with a manually annotated grammar obtained from the unlabeled grammar. We used the same set of syntactically valid and invalid programs to validate the generated grammar and the manually annotated one.

In our comparison, we will check the labels of the generated grammar against the labels of the manually annotated grammar. We will discuss mainly the following items:

Equal	When the algorithm correctly inserted a label, as the manual annotation did.
Extra	When the algorithm correctly inserted a new label.
Wrong	When the algorithm incorrectly inserted a label.

Table 1 shows the result of comparing the automatically inserted labels with the manually ones. Below, in Sections 4.1, 4.2, 4.3 and 4.4 we discuss the automatic insertion of labels for each language.

Ideally, we would want a generated grammar with the same labels as the manually annotated one, hopefully with a few new correct labels missed during manual annotation. To a certain extent, we do not consider missing to add some labels a serious flaw of Algorithm Standard, as long as most of the labels are correctly inserted, since failing to add labels does not lead to an incorrect parser. These (hopefully few) labels can still be manually inserted later by an expert.

A discrepancy related to Item Wrong is more problematic, since it can produce a parser that does not recognize some syntactically valid programs. This limitation of our algorithm means that the output needs to be checked by the parser developer to ensure that the algorithm did not insert labels incorrectly.

[1] https://github.com/sqmedeiros/lpeglabel.
[2] https://github.com/sqmedeiros/pegparser.

Table 1
Evaluation of the Labels Inserted by Algorithm Standard.

Approach	Equal	Extra	Wrong
Manual	86	0	0
Standard	76	2	2

(a) Labels Inserted for Titan

Approach	Equal	Extra	Wrong
Manual	87	0	0
Standard	65	9	1

(b) Labels Inserted for C

Approach	Equal	Extra	Wrong
Manual	102	0	0
Standard	100	1	3

(c) Labels Inserted for Pascal

Approach	Equal	Extra	Wrong
Manual	175	0	0
Standard	139	10	32

(d) Labels Inserted for Java

Table 2
Evaluation of Automatic Error Recovery Based on Algorithm Standard.

Approach	Excel.	Good	Poor	Awful
Manual	91%	4%	5%	0%
Standard	81%	3%	15%	1%

(a) Error Recovery for Titan

Approach	Excel.	Good	Poor	Awful
Manual	87%	7%	5%	2%
Standard	67%	5%	27%	2%

(b) Error Recovery for C

Approach	Excel.	Good	Poor	Awful
Manual	80%	11%	9%	0%
Standard	80%	11%	9%	0%

(c) Error Recovery for Pascal

Approach	Excel.	Good	Poor	Awful
Manual	74%	17%	9%	0%
Standard	63%	16%	21%	0%

(d) Error Recovery for Java

This checking can be done either by manual inspection of the grammar or by running the generated parser against test programs. In this latter case, when the parser fails to recognize a valid program, the parsing result will point the label incorrectly added. Once identified, we need to remove the incorrect label from the grammar.

After analyzing how Algorithm Standard annotated the grammar of a given language, we will discuss the error recovering parser generated by it. During this discussion we will assume that we have already removed the labels that Algorithm Standard may have inserted incorrectly.

As we mentioned, Algorithm Standard associates a recovery expression to each label. To recover from a label l we add a recovery rule l to the grammar, where the right-hand side of l is its recovery expression. The generated grammar has a recovery rule associated with each label.

As `pegparser` automatically builds an AST when the match is successful, we will evaluate the error recovering parser got from a generated grammar by comparing the AST built by the parser for a syntactically invalid program with the AST of what would be an equivalent correct program. For the AST leaves associated with a syntax error, we do not require their contents to be the same, just the general type of the node, so we are comparing just the structure of the ASTs.

Based on this strategy, a recovery is *excellent* when it gives us an AST equal to the intended one. A *good* recovery gives us a reasonable AST, i.e., one that captures most information of the original program (e.g., it does not miss a whole block of commands). A *poor* recovery, by its turn, produces an AST that loses too much program information. Finally, a recovery is rated as *awful* whenever it gives us an AST without any information about the program.

Table 2 shows for how many programs of each language the recovery strategy we implemented was considered *excellent*, *good*, *poor*, or *awful*. Sections 4.1, 4.2, 4.3 and 4.4 discuss the results of error recovery for each language. In case of the manually annotated grammars, to evaluate them we added recovery rules based on the way Algorithm Standard generates recovery rules for labels.

To illustrate how we rated a recovery, let us consider the following syntactically invalid Titan program, where the range start of the `for` loop was not given at line 2:

```
1  sum = 0
2  for i = , 10 do
3    print(i)
4    sum = sum + i
5  end
```

A recovery would be *excellent* in case the AST has all the information associated with this program (such AST should have a dummy node to represent the range start). A recovery would be *good* in case the resulting AST misses only the information about the loop range. By its turn, a recovery would be rated as *poor* in case the resulting AST misses the statements inside the `for` (lines 3 and 4). Lastly, we would rate a recovery as `awful` in case it would have produced an AST only with dummy nodes.

Below, based on the approach discussed previously, we evaluate the use of Algorithm Standard to generate error recovering parsers for the programming languages Titan, C, Pascal and Java.

4.1. Titan

Titan [11] is a new statically-typed programming language under development to be used as a sister language to the Lua programming language [12].

After some initial development, the Titan parser was manually annotated with labels to improve its error reporting. The original Titan parser[3] has no error recovery, it stops parsing the input after encountering the first syntax error. Based on it, we wrote our unlabeled grammar for Titan,[4] which has 50 syntactical rules.

The Titan grammar is not $LL(1)$, there are non-$LL(1)$ choices in 7 rules and non-$LL(1)$ repetitions in 3 rules, but it has many $LL(1)$ parts.

The manually annotated Titan grammar[5] we got from our unlabeled grammar is equivalent to the original Titan grammar, we have just adapted the grammar syntax to be able to use the `pegparser` tool.

The manually annotated grammar has 86 expressions that throw labels. Some labels, such as `EndFunc`, are thrown more than once, i.e., they are associated with more than one expression.

We then applied Algorithm Standard to this unlabeled grammar and got an automatically annotated Titan grammar, with a recovery expression associated to each label.[6]

In Section 4.1.1, we compare the labels automatically inserted with the labels in the original Titan grammar. Then, in Section 4.1.2, we will discuss the error recovery mechanism of the generated Titan grammar.

4.1.1. Automatic insertion of labels

Algorithm Standard annotated the Titan grammar with 80 labels, which is close to the 86 labels of the original Titan grammar. A manual inspection revealed that usually the algorithm inserted labels at the same location of the original ones, as Table 1a shows. We could insert automatically around 90% of the labels inserted manually. Below we discuss the main issues related to the generated Titan grammar.

As expected our approach did not annotate parts of the grammar where the alternatives of a choice were not disjoint, on in case of a non-disjoint repetition. This happened in 4 of the 50 grammar rules. One of these rules was *castexp*, which we show below:

$$castexp \leftarrow simpleexp \; \texttt{AS} \; type \; / \; simpleexp$$

As we can see, both alternatives of the choice match a *simpleexp*, so these alternatives are not disjoint. After manual inspection, we can see it is possible to add a label to *type* in the first alternative, since the context where *castexp* appears in the rest of the grammar makes it clear that a failure on *type* is always a syntax error. Left-factoring the right-hand side of *castexp* to $simpleexp \; (\texttt{AS} \; type \; / \; \varepsilon)$, or using the short form $simpleexp \; (\texttt{AS} \; type)?$, would give enough context for Algorithm Standard to correctly annotate *type* with a label, though.

The manually annotated Titan grammar uses an approach known as *error productions* [13]. As an example, the choice associated with rule *statement* has two extra alternatives whose only purpose it to match some usual syntactically invalid statements, in order to provide a better error message. One of these alternatives is as follows:

$$\&(exp \; \texttt{ASSIGN}) \; \Uparrow^{\texttt{ExpAssign}}$$

Before this alternative, the grammar has one that tries to match an assignment statement. That alternative might have failed because the programmer used an expression that is not a valid l-value in the left-hand side of the assignment. This error production guards against this case. Without the error production, the parser would still fail, but we would get an error related to not closing a function, which may be confusing for a user.

The Algorithm Standard does not add error productions, and we think they should only be added by an expert.

In case of Titan, the algorithm inserted two labels incorrectly, a problem related to Item Wrong, which made the parser reject valid inputs. Although these two labels have also been added during the manual annotation, their insertion by Algorithm Standard was undue, as we will see. This issue happened in rules *toplevelvar* and *import*. Fig. 5 shows the definition of these rules, plus some rules that help to add context, in the manually annotated Titan grammar.

Non-terminals *toplevelvar*, *import* and *foreign* are alternatives of a non-$LL(1)$ choice in rule *program*. The parser first tries to recognize *toplevelvar*, then *import*, and finally *foreign*. As a *decl* may consist of only a name, an input like “`local x =`” may be the beginning of any of these rules. In rule *toplevelvar*, the predicate $!(\texttt{IMPORT} \; / \; \texttt{FOREIGN})$ was added by the Titan

[3] http://bit.ly/titan-reference.
[4] http://bit.ly/titan-unlabeled.
[5] http://bit.ly/titan-manual.
[6] http://bit.ly/titan-standard.

$$\begin{aligned}
program &\leftarrow (toplevelfunc \;/\; toplevelvar \;/\; toplevelrecord \;/\; import \;/\; foreign) * \texttt{EOF}\\
toplevelvar &\leftarrow localopt\ decl\ [\texttt{ASSIGN}]^{\texttt{AssignVar}}\ !(\texttt{IMPORT} \;/\; \texttt{FOREIGN})\ [exp]^{\texttt{ExpVarDec}}\\
import &\leftarrow \texttt{LOCAL}\ [\texttt{NAME}]^{\texttt{NameImport}}\ [\texttt{ASSIGN}]^{\texttt{AssignImport}}\\
&\qquad !\texttt{FOREIGN}\ [\texttt{IMPORT}]^{\texttt{ImportImport}}\ (\texttt{LPAR}\ [\texttt{STRING}]^{\texttt{StringLParImport}}\ [\texttt{RPAR}]^{\texttt{RParImport}} \;/\; [\texttt{STRING}]^{\texttt{StringImport}})\\
foreign &\leftarrow \texttt{LOCAL}\ [\texttt{NAME}]^{\texttt{NameImport}}\ [\texttt{ASSIGN}]^{\texttt{AssignImport}}\\
&\qquad \texttt{FOREIGN}\ [\texttt{IMPORT}]^{\texttt{ImportImport}}\ (\texttt{LPAR}\ [\texttt{STRING}]^{\texttt{StringLParImport}}\ [\texttt{RPAR}]^{\texttt{RParImport}} \;/\; [\texttt{STRING}]^{\texttt{StringImport}})\\
decl &\leftarrow \texttt{NAME}\ (\texttt{COLON}\ [type]^{\texttt{TypeDecl}})?\\
localopt &\leftarrow \texttt{LOCAL}?
\end{aligned}$$

Fig. 5. Predicates !(IMPORT / FOREIGN) and !FOREIGN enable adding labels after them.

developers to make sure the input neither matches the *import* nor the *foreign* rule, so it is safe to throw an error after this predicate in case we do not recognize an expression. The predicate !FOREIGN in rule *import* plays a similar role.

As Titan developers inserted these predicates solely to enable the subsequent label annotations, we judged that we would do a fairer evaluation by removing them from our unlabeled grammar.

In rule *program*, although alternatives *toplevelvar*, *import*, and *foreign* have LOCAL in their $FIRST$ sets, the algorithm adds labels to the right-hand side of these non-terminals, because it does not take into consideration the fact these non-terminals appear as alternatives in a non-$LL(1)$ choice.

The outcome is that the algorithm is able to insert the same labels added by manual annotation, but without the syntactic predicates we should not throw label AssignImport in rule *toplevelvar* and label ImportImport in rule *import*. As Algorithm Standard inserted these labels, the resulting parser will wrongfully signal errors in valid inputs such as "`local x = import "foo"`".

After removing these labels, our generated Titan parser successfully passed the Titan tests.

We think this was less work than manually annotating the grammar, given that the parser already needs to have an extensive test suite that will catch these errors, as was the case in our evaluation.

Lastly, Algorithm Standard correctly added two new labels. It annotated RARROW in the first alternative of rule *type*, and FOREIGN in rule *foreign*.

4.1.2. Automatic error recovery

The test suite of Titan has 74 tests related to syntactically invalid programs. For our evaluation of automatic error recovery, we ran the Titan parser against these files and we analyzed the AST built for each of them. Since that our parser will only build an AST for a successful matching, the grammar start rule should not fail. Thus, as a special case, we should annotate the expressions of the grammar start rule which may lead to a failure. In case of Titan, we should annotate EOF and add a recovery rule that consumes the rest of the input. By doing this, we will get an AST whenever we successfully match an input prefix before matching EOF. We will use this same approach for the other languages. It is not difficult to extend the Algorithm Standard with this extra case involving the start rule.

We can see in Table 2a that our recovery mechanism for Titan seems promising, since that more than 80% of the recovery done was considered acceptable, i.e., it was rated at least *good*.

By analyzing the programs for which our parser built a poor AST, we can see that most cases (9 out of 11) are related to missing labels. Instead of throwing such labels and recovering from them using their corresponding recovery expressions, the generated parser will produce a regular failure, which either leads to the failure of a matching or makes the parser backtrack.

As an example, let us see the case of a missing label related to rule *castexp*, which we have shown in Section 4.1.1. In the following input there is a missing type after the keyword "as" at line 1:

```
1  x = foo as
2  return x
```

The manually annotated parser would have thrown an error after "as". However, as we have discussed in Section 4.1.1, Algorithm Standard did not annotate this rule. Thus, the automatically generated parser will produce a regular failure after failing to match *type* after "as".

This leads the first alternative of rule *castexp* to fail, then the second alternative matches just the input "foo". This will lead to another failure when the parser tries to match "as" as the beginning of a statement.

As Algorithm Standard was able to insert most of the labels inserted by manual annotation, usually the generated Titan parser was able to recover from an syntactic error and to build an AST with nearly all the information about a program.

$$\begin{aligned}
translation\_unit &\leftarrow external\_decl+\ !\texttt{EOF} \\
external\_decl &\leftarrow function\_def\ \ /\ \ decl \\
function\_def &\leftarrow declarator\ decl*\ \ compound\_stat\ \ /\ \ decl\_spec\ [function\_def]^{\texttt{ErrFuncDef}} \\
decl\_spec &\leftarrow storage\_class\_spec\ \ /\ \ type\_spec\ \ /\ \ type\_qualifier \\
decl &\leftarrow decl\_spec\ \ init\_declarator\_list?\ \texttt{SEMI}\ \ /\ \ decl\_spec\ [decl]^{\texttt{ErrDecl}}
\end{aligned}$$

Fig. 6. Label *ErrFuncDef* Incorrectly Added in Rule *function_def*.

4.2. C

We have developed a parser for C, without preprocessor directives, based on the reference grammar presented by Kernighan and Ritchie [14], which is essentially a grammar for ANSI C89.

To write our unlabeled grammar for C[7] we needed to remove left-recursion, as `LPegLabel` does not accept grammars with left-recursive rules. After this, we got an unlabeled grammar for C with 50 syntactical rules, from which 17 have non-$LL(1)$ choices and 5 have non-$LL(1)$ repetitions.

Due to the `typedef` feature, to correctly recognize the C syntax we need the help of semantic actions to determine when a name should be considered a *typedef_name*. As we did not implement these semantic actions, we disabled the matching of this rule to not incorrectly recognize an identifier as a *typedef_name*.

The manually annotated C grammar[8] has 87 expressions that throw labels. By its turn, the automatically annotated C grammar[9] we got after applying Algorithm Standard has 75 labels.

In Section 4.2.1, we compare the manually annotated C grammar with the automatically annotated one. After, in Section 4.2.2, we will discuss the error recovering C parser we got from this automatically annotated grammar.

4.2.1. Automatic insertion of labels

As was the case for Titan, often the Algorithm Standard inserted labels at the same location of the original ones, as Table 1b shows. The algorithm was able to insert 75% of the labels inserted manually.

As our C grammar has many rules with non-$LL(1)$ choices (17 out of 50), and some rules with $non - LL(1)$ repetitions too, it was not possible to automatically add some labels in these rules.

Algorithm Standard incorrectly added one new label, in rule *function_def*. Fig. 6 shows the definition of this rule, plus other rules that help to add context, in the generated C grammar.

The cause of the problem related to Item Wrong in the C grammar is similar to the one discussed in Titan grammar in Section 4.1.1. In rule *external_decl*, we have a non-$LL(1)$ choice, since that a *decl_spec* may be the beginning of a *function_def* as also of a *decl*.

When we annotate the right-hand side of the rule associated with non-terminal *function_def*, which appears in the first alternative of the non-$LL(1)$ choice in rule *external_decl*, we may throw a label incorrectly. In this case, given an input like "`int x;`", we would match "`int`" as a *decl_spec* and we would throw label `ErrFuncDef` after failing to recognize "`x;`" as a *function_def*. After removing label `ErrFuncDef`, our generated C parser successfully passed the tests.

Finally, Algorithm Standard added 9 new labels correctly, which is more than the 2 new labels added for the Titan grammar. We think this may be due to the higher rate of non-disjoint expressions in our C grammar, which may have imposed a more conservative behavior during manual annotation.

Nevertheless, the manual annotation is not free of faults. For both grammars some labels were added during manual annotation and later removed when the parser failed to recognize syntactically valid programs.

4.2.2. Automatic error recovery

The test suite we used for our C parser has 59 syntactically invalid programs. As we did for Titan, we ran the generated C parser against these files and we analyzed the AST built for each of them. As we discussed in Section 4.1.2, we manually added labels to the grammar start rule to assure our parser will build an AST when it successfully matches an input prefix. In the case of the C grammar, we added two labels to the right-hand side of the grammar start rule.

In Table 2b we can see that for more than 70% of the syntactically invalid programs in our test set the recovery done was considered acceptable, i.e., it was rated at least *good*.

Similarly to Titan (see Section 4.1.2), in most cases (12 out of 16) we can associate the building of a poor AST by our parser with the absence of a label.

As our C grammar has more non-$LL(1)$ choices, Algorithm Standard missed more labels, which makes a proper recovery more difficult and results in more poor ASTs.

[7] http://bit.ly/c89-unlabeled.
[8] http://bit.ly/c89-manual.
[9] http://bit.ly/c89-standard.

$$\begin{aligned} stat \leftarrow\ & \texttt{IF}\ [\texttt{LPAR}]^{\texttt{BrackIf}}\ [exp]^{\texttt{InvalidExpr}}\ [\texttt{RPAR}]^{\texttt{Brack}}\ [stat]^{\texttt{Stat}}\ \texttt{else}\ [stat]^{\texttt{Stat}} \\ /\ & \texttt{IF}\ [\texttt{LPAR}]^{\texttt{BrackIf}}\ [exp]^{\texttt{InvalidExpr}}\ [\texttt{RPAR}]^{\texttt{Brack}}\ [stat]^{\texttt{Stat}} \end{aligned}$$

Fig. 7. Manually Annotated *if-else* Statement.

$$\begin{aligned} simpleStmt &\leftarrow assignStmt\ /\ procStmt\ /\ gotoStmt \\ assignStmt &\leftarrow var\ [\texttt{ASSIGN}]^{\texttt{AssignErr}}\ [expr]^{\texttt{ExprErr}} \\ var &\leftarrow \texttt{ID}\ (\texttt{LBRA}\ [expr]^{\texttt{ExprErr}}\ (\texttt{COMMA}\ [expr]^{\texttt{ExprErr}})*\ [\texttt{RBRA}]^{\texttt{RBrackErr}}\ /\ \texttt{DOT}\ [\texttt{ID}]^{\texttt{IdErr}}\ /\ \texttt{POINTER})* \\ procStmt &\leftarrow \texttt{ID}\ params? \end{aligned}$$

Fig. 8. Label *AssignErr* Incorrectly Added in Rule *assignStmt*.

As an example, let us see the case of a missing label related to an `if-else` statement. Fig. 7 shows the definition of such statement in rule *stat* of the manually annotated C grammar. Other alternatives of rule *stat* were omitted for simplicity.

As the choice in *stat* is not $LL(1)$, Algorithm Standard will not add the 5 labels to the first alternative of this choice. Given a program as the following one, where there is no statement associated with the `else`:

```
int fat (int x) {
  if (x == 0)
    return 1;
  else
}
```

The generated C parser will try to recognize the first alternative of the choice in rule *stat*. It will fail to recognize *stat* after "`else`", which will produce a regular failure. Thus, the parser backtracks, recognize an *if*-statement without an *else*-part, and then will fail to recognize another statement as we left "`else`" on the input.

As we commented out in Section 4.1.1, we could rewrite this choice to put in evidence the common prefix. After doing this, Algorithm Standard could annotate the *if*-statement and we would get a better recovery in this case.

Although Algorithm Standard will not annotate `LPAR` in the first alternative of the choice above, this will not make error recovery worst in case of a missing "`(`" after "`if`", as long as we annotate `LPAR` in the second alternative. The reason for this is that after failing to match `LPAR` via the first alternative, the parser will backtrack and eventually match `LPAR` via the second alternative. The same rationale applies for the other labels present in the common prefix of both alternatives.

4.3. Pascal

We have developed a parser for Pascal based on the grammar available in the ISO 7185:1990 standard [15]. Our unlabeled Pascal grammar[10] has 67 syntactical rules. Among these rules, 4 of them have non-$LL(1)$ choices, and 6 of them have non-$LL(1)$ repetitions.

The manually annotated Pascal grammar[11] has 102 expressions that throw labels.

By using Algorithm Standard, from the unlabeled Pascal grammar we got a generated grammar[12] with 104 labels. Below, Section 4.3.1 compares the manually annotated grammar with the generated one, and Section 4.3.2 discusses the error recovering Pascal parser we got from this generated grammar.

4.3.1. Automatic insertion of labels

As Table 1c shows, Algorithm Standard annotated the Pascal grammar in a way nearly identical to manual annotation, it inserted 98% of the labels inserted manually. We think the low number of non-$LL(1)$ choices and non-$LL(1)$ repetitions helped the algorithm to achieve this performance.

However, three of the labels inserted by Algorithm Standard were added incorrectly. The incorrect labels were added to rules *subrangeType*, *assignStmt* and *funcCall*. All these rules are referenced (directly or indirectly) in the first alternative of non-$LL(1)$ choices, where an identifier belong to the $FIRST$ set of both choice alternatives. Let us discuss the problem related to *assignStmt*, whose definition is given in Fig. 8.

We can see in this figure that there is a non-$LL(1)$ choice in rule *simpleStmt*, as `ID` belongs to the $FIRST$ set of both *assignStmt* and *procStmt*. Due to this, in rule *assignStmt*, which appears in the first alternative of this choice, we should not annotate `ASSIGN`, otherwise the parser will not recognize a valid *procStmt* such as "`f(x)`", as "`:=`" does not follow the identifier "`f`".

10 http://bit.ly/pascal-unlabeled.

11 http://bit.ly/pascal-manual.

12 http://bit.ly/pascal-standard.

After removing the incorrect labels in rules *subrangeType*, *assignStmt* and *funcCall*, our generated Pascal parser successfully passed the tests.

Lastly, Algorithm Standard also added 2 new labels correctly.

4.3.2. Automatic error recovery

Our test suite for Pascal has 101 syntactically invalid programs. We can see in Table 2c that for more than 90% of the syntactically invalid programs in our test set the recovery done was considered acceptable, i.e., it was rated at least *good*.

Differently from the analysis we did for the Titan and the C error recovering parsers, in case of the Pascal parser we can not associate the poor ASTs with the absence of labels. A manual inspection indicates that most of poor ASTs built were due to synchronizing the input too early (instead of discarding one more token). This issue may be fixed by adjusting the recovery expression used. Our approach allows to do this tuning manually for a given recovery expression.

Overall, a recovery strategy may show a better performance after it is tuned to match features of a given language.

4.4. Java

We have developed a parser for Java 8 following the parser available at the Mouse site.[13]

Our unlabeled Java grammar[14] has 147 syntactical rules, where there are 35 rules with a non-$LL(1)$ choice and 15 rules with a non-$LL(1)$ repetition. A rule may have a non-$LL(1)$ choice and also a non-$LL(1)$ repetition, but this occurs in only 2 rules. Overall, one third of the grammar rules has an $LL(1)$ conflict. The manually annotated Java grammar[15] has 175 expressions that throw labels.

From the unlabeled Java grammar, we used Algorithm Standard to get a generated grammar[16] with 181 labels.

In Section 4.4.1 we compare the manually annotated grammar with the generated one, and in Section 4.4.2 we discuss our error recovering parser for Java.

4.4.1. Automatic insertion of labels

We can see in Table 1d that Algorithm Standard annotated the Java grammar with 181 labels, from which 139 were also inserted during the manual annotation. This seems a good amount, given that many rules of the grammar have an $LL(1)$ conflict.

The $LL(1)$ conflicts also impose a difficult to add labels correctly. As a consequence of this, an important part of the labels added (18%) by Algorithm Standard were inserted incorrectly. The cases where these labels were inserted are similar to the cases of incorrect labels we have already discussed for the other languages, so we will not present them here.

The significant number of incorrect labels added limits somewhat the usefulness of using Algorithm Standard to annotate our unlabeled Java grammar, since that it is necessary to manually remove several labels later. Although this removal is not hard, the usual process requires running the tests once for each incorrect label, and then removing such label after failing to pass the tests.

Finally, Algorithm Standard also correctly added 10 new labels.

4.4.2. Automatic error recovery

Our test suite for Java has 175 syntactically invalid programs. Table 2d shows that for almost 80% of these programs the recovery done was considered acceptable, i.e., it was rated at least *good*.

About half of the cases where our generated parser built a poor AST are related to a missing label. We could get a better result in these cases by rewriting non-disjoint choices, as we have shown for Titan and C, so Algorithm Standard could insert more labels and their corresponding recovery rules.

For also about half of the cases we got a poor AST because of an intersection between the tokens that could follow a symbol in the right-hand side of a rule A and the tokens that could follow A itself. To improve these ASTs we usually need either to manually add labels to the grammar or to manually tune the recovery rules.

5. Conservative insertion of labels

As have discussed previously, Algorithm Standard annotates a grammar with labels, but it may add labels incorrectly, which leads to a parser that rejects some valid inputs. To avoid this shortcoming, we will discuss conservative approaches, which address the problem related to Item Wrong.

[13] http://www.romanredz.se/Mouse/Java.1.8.peg.
[14] http://bit.ly/java8-unlabeled.
[15] http://bit.ly/java8-manual.
[16] http://bit.ly/java8-standard.

$$
\begin{aligned}
ordinalType &\leftarrow newOrdinalType \;/\; \texttt{ID} \\
newOrdinalType &\leftarrow enumType \;/\; subrangeType \\
enumType &\leftarrow \texttt{LPAR}\; [ids]^{\texttt{IdErr}}\; [\texttt{RPAR}]^{\texttt{RParErr}} \\
subrangeType &\leftarrow const\; [\texttt{DOTDOT}]^{\texttt{DotDotErr}}\; [const]^{\texttt{ConstErr}} \\
const &\leftarrow \texttt{SIGN}?\; (\texttt{UNUMBER} \;/\; \texttt{ID}) \;/\; \texttt{STRING}
\end{aligned}
$$

Fig. 9. Label *DotDotErr* Incorrectly Added in Rule *subrangeType*.

5.1. Non-terminals banning

Our first approach to not insert labels incorrectly is based on the idea of banning a non-terminal A that is used in a non-disjoint choice or a non-disjoint repetition. When A is banned, we do not annotate its right-hand side. To properly avoid the wrong insertion of labels, this approach should be recursive, i.e., when banning A we should also ban the non-terminals in the right-hand side of A.

To illustrate this point, let us consider Fig. 9, which shows an excerpt from Pascal grammar. In rule *ordinalType* there is a non-disjoint choice, where `ID` belongs to the $FIRST$ set of both alternatives of the choice. Because of this, we should ban the non-terminal *newOrdinalType*, so we will not annotate its right-hand side.

In case the banning process is not recursive, as in rule *newOrdinalType* there is no conflict, we will not ban the non-terminals in its right-hand side. This approach leads to incorrectly adding label `DotDotErr` in rule *subrangeType*.

We should not throw `DotDotErr` because in rule *ordinalType*, when matching the first alternative of $newOrdinalType$ / `ID`, the parser could recognize an `ID` as the beginning of a *subrangeType*, then fail to recognize `DOTDOT`, backtrack and finally match the second alternative. Thus, we should apply a recursive banning approach to avoid adding labels incorrectly.

The result of applying such approach leads to the insertion of a few labels, or even none. When there are conflicts in the top-level grammar rules, the recursive banning strategy bans almost all non-terminals. For the C and Java grammars, after banning the non-terminals related to a non-disjoint choice or repetition, we could not add a single label. In case of Titan, we could add 12 labels, while for Pascal, which has few non-disjointness conflicts, we had the best result and could add 36 labels, which corresponds to 35% of the labels we have inserted manually.

Although the recursive banning approach have added only correct labels, its usefulness seems quite limited. Therefore we will use this strategy only as a complementary one. Below, we discuss a more effective approach, based on the idea of unique non-terminals, to conservatively insert only correct labels.

5.2. Unique non-terminals

In Section 3 we saw that the main challenge when adding labels is to determine statically when failing to match an expression p indicates that the parser has no other viable option to recognize the input.

In order to identify these safe places where we can insert labels, we will introduce the concept of <u>unique lexical non-terminals</u>. The following definition says that a lexical non-terminal A is <u>unique</u> when it appears in the right-hand side of only one syntactical rule, and just once:

Definition 3 *(Unique lexical non-terminal). Given a PEG* $G = (V_{Lex} \cup V_{Syn}, T, P, L, R, \texttt{fail}, p_S)$*,* $A \in V_{Lex}$ *is unique iff* $\exists B \in V_{Syn}$ *such that* A *is used only once in* $P(B)$ *and* $\forall C \in V_{Syn}$*, where* $C \neq B$*, we have* A *is not used in* $P(C)$*.*

When we have a grammar G with the unique token prefix property, and A is a unique lexical non-terminal of G, once A matched, failing to match the expression the follows A leads to the failure of the whole matching, as the following lemma states:

Lemma 3 *(Unique matching). Let* $G = (V_{Lex} \cup V_{Syn}, T, P, L, R, \texttt{fail}, p_S)$ *be an unlabeled PEG, with the unique token prefix property, and let* w *be a subject* w*. Let* $A\ p_2$ *be a subexpression of* $P(B)$*, where* A *is a unique lexical non-terminal and* $B \in V_{Syn}$*, and let* axy *be a suffix of* w*, if* $G[A]\ R\ axy \overset{PEG}{\rightsquigarrow} y$ *and* $G[p_2]\ R\ y \overset{PEG}{\rightsquigarrow} (\texttt{fail},\ y')$*, then* $G[p_S]\ R\ w \overset{PEG}{\rightsquigarrow} (\texttt{fail},\ w')$*.*

Proof. The proof uses Lemma 2 and the fact that G has the unique token prefix property and A is a unique lexical non-terminal.

When the matching of p_2 fails, either we backtrack to a previous choice and try to match a different alternative, or we do not backtrack.

In the former case, by Lemma 2 we know that after backtracking the grammar will match the same sequence of tokens, thus we will need to match axy again. As G has the unique token prefix property, only A matches axy, and given that A is a unique lexical non-terminal, A is not used anywhere else in G. Therefore, once more A would match prefix ax and p_2 would fail to match y, leading to the failure of the whole matching.

The proof of the last case, when there is no backtracking, is straightforward given the previous discussion. □

As a result of Lemma 3, we know that after matching a unique lexical non-terminal A we start a kind of unique path, and failing to match an expression that follows A indicates that the input is invalid. Therefore, we can safely annotate the expression p_2 that follows A.

Based on this, we present the Algorithm Unique, which automatically annotates a PEG $G = (V, T, P, L, R, \texttt{fail}, p_S)$. In comparison with the Algorithm Standard, function `labexp` now receives an extra parameter, $afterU$, which indicates if we have already matched a unique lexical non-terminal, and function `matchUni`, which determines whether a parsing expression p matches at least one unique lexical non-terminal or not, is new. Below we discuss these functions in more detail. Functions `annotate`, `calck` and `addlab` remain the same and their definitions were omitted.[17] We assume the unique lexical non-terminals have already been computed. Given a non-terminal A, function `isUniLex` returns true in case A is a unique lexical non-terminal, and false otherwise.

Algorithm Unique Inserting Labels and Recovery Expressions in a PEG after Unique Lexical Non-Terminals

```
 1: function LABEXP($p, seq, afterU, flw$)
 2:     if $p = A$ and $\varepsilon \notin FIRST(A)$ and $seq$ and $afterU$ then
 3:         return $addlab(p, flw)$
 4:     else if $p = p_1\ p_2$ then
 5:         $p_x \leftarrow labexp(p_1, seq, afterU, calck(p_2, flw))$
 6:         $p_y \leftarrow labexp(p_2, seq$ or $\varepsilon \notin FIRST(p_1), afterU$ or $matchUni(p1), flw)$
 7:         return $p_x\ p_y$
 8:     else if $p = p_1\ /\ p_2$ then
 9:         $disjoint \leftarrow FIRST(p1) \cap calck(p2, flw) = \emptyset$
10:         $p_x \leftarrow labexp(p_1,$ false$, disjoint$ and $afterU, flw)$
11:         $p_y \leftarrow labexp(p_2,$ false$, afterU, flw)$
12:         if $seq$ and $\varepsilon \notin FIRST(p_1\ /\ p_2)$ and $afterU$ then
13:             return $addlab(p_x\ /\ p_y, flw)$
14:         else
15:             return $p_x\ /\ p_y$
16:     else if $p = p_1*$ then
17:         $disjoint \leftarrow FIRST(p1) \cap calck(p2, flw) = \emptyset$
18:         return $labexp(p_1,$ false$, disjoint$ and $afterU, FIRST(p_1) \cup flw)*$
19:     else
20:         return $p$
21:
22: function MATCHUNI($p$)
23:     if $p = A$ and $islexrule(A)$ then
24:         return $isUniLex(p)$
25:     else if $p = p_1\ p_2$ then
26:         return $matchUni(p_1)$ or $matchUni(p_2)$
27:     else if $p = p_1\ /\ p_2$ then
28:         return $matchUni(p_1)$ and $matchUni(p_2)$
29:     else if $p = p_1+$ then
30:         return $matchUni(p_1)$
31:     else
32:         return false
```

Function `labexp` (lines 1–20) has four parameters: p, a parsing expression; seq, a boolean value indicating whether the current concatenation have already matched at least one token; $afterU$, a boolean value indicating whether the current right-hand side have already matched at least one unique lexical non-terminal; flw, the $FOLLOW$ set associated with p.

When p is a non-terminal that does not match the empty string and both seq and $afterU$ are **true** (lines 2–3), then we associate a new label with p. When p is a non-terminal but these conditions do not hold, we will just return p itself (lines 19–20).

In case of a concatenation $p_1\ p_2$ (lines 4–7), the main difference to Algorithm Standard is the handling of parameter $afterU$ when annotating p_2 (line 6). In this case, we supply a **true** value for $afterU$ when it is already **true** or when p_1 consumes at least one unique lexical non-terminal.

When p is a choice $p_1\,/\,p_2$ (lines 8–15), a main difference to Algorithm Standard is that we call `labexp` recursively even when the choice is not disjoint. In this case, we set $afterU$ to **false** when annotating p_1 (line 10). The rationale is that is not safe to throw a label after failing to match p_1 in such case, since the parser can still backtrack and consume the input via p_2. We will only add labels to p_1 in case an expression of p_1 matches a unique lexical non-terminal. When annotating p_2, we pass the current value of $afterU$, since there is no other alternative and thus it is safe to annotate p_2 in

[17] Actually, now function `annotate` provides a false value to $afterU$ when calling `labexp`.

Table 3
Evaluation of the Labels Inserted by the Different Approaches for each Grammar.

Approach	Equal	Extra	Wrong
Manual	86	0	0
Standard	76	2	2
Unique	60	3	0

(a) Labels Inserted for Titan

Approach	Equal	Extra	Wrong
Manual	87	0	0
Standard	65	9	1
Unique	48	2	0

(b) Labels Inserted for C

Approach	Equal	Extra	Wrong
Manual	102	0	0
Standard	100	1	3
Unique	74	6	0

(c) Labels Inserted for Pascal

Approach	Equal	Extra	Wrong
Manual	175	0	0
Standard	139	10	32
Unique	80	16	0

(d) Labels Inserted for Java

case we have matched a unique lexical non-terminal before. Whether both seq and $afterU$ are **true**, we associate a label to the whole choice when it does not match the empty string.

In case p is a repetition p_1* (lines 16–18), differently from Algorithm Standard and similarly to the case we discussed before, we also call `labexp` recursively when the repetition is not disjoint, providing a false value in this case.

After applying Algorithm Unique, we could see we added, as expected, only correct labels to the grammars we have been discussing so far. In case of Titan, for example, we added 42 labels, while in case of Java we added 51 labels. To increase the number of labels inserted, we will do some extra analysis to determine whether when matching a given expression p we are in a unique path (and thus we can annotate p) or not.

Below, we discuss some analyses we did to compute this unique path. When evaluating Algorithm Unique, in Section 6, we assume this extra analysis was performed:

- **Unique Syntactical Non-Terminal:** When an syntactical non-terminal A is only used after we have already matched a unique lexical non-terminal, then we can also mark A as unique and annotate its right-hand side. Both lexical and syntactical non-terminals can be marked as unique now, the main difference is that in case of a unique syntactical non-terminal this implies providing a true value for parameter $afterU$ when calling `labexp` to annotate the right-hand side of A.
- **Unique Context:** If the lexical non-terminal A is used more than once in grammar G but the set S of tokens that may occur immediately before an usage of A is unique, i.e., $\forall s \in S$ we have that s may not occur immediately before the other usages of A, then we can mark this instance of A preceded by S as unique.

In the next section, we compare the number of labels inserted by Algorithm Unique with the number of labels inserted via manual annotation and by using Algorithm Standard, as also as the resulting error recovering parsers obtained via each approach.

6. Evaluating the conservative insertion of labels

Table 3 shows the amount of labels inserted for the Titan, C, Pascal and Java grammars when we used an automatic approach and when we used manual annotation.

We can see that the manual approach is the one that adds more labels for all the grammars we evaluated, then comes Algorithm Standard, and finally Algorithm Unique, which, as expected, did not insert labels incorrectly.

Overall, the amount of labels added by Algorithm Unique, when compared with manual annotation, ranged from 55%, in case of Java, to 78%, in case of Pascal. By its turn, when compared with Algorithm Standard, the Algorithm Unique was able to insert between 64%, in case of Java, and 81%, in case of Titan, of the labels inserted by it.

In Table 4, we can see that the smaller amount of labels, and thus of recovery rules, inserted by Algorithm Unique leads to a parser that performs a poorer recovery when compared to the error recovering parsers based on manual annotation and on Algorithm Standard. In the best scenario, the Pascal grammar, Algorithm Unique give us a parser that usually (in 84% of the cases) performs an acceptable recovery when the other two approaches do. In the worst scenario, the Java grammar, the parser produced by Algorithm Unique only performs an acceptable recovery in around half of the cases the other approaches do.

Below, in Sections 6.1, 6.2, 6.3 and 6.4, we discuss in more detail the use of Algorithm Unique to annotate the grammar of each language.

Table 4
Evaluating the Error Recovery for Each Approach.

Approach	Excel.	Good	Poor	Awful
Manual	91%	4%	5%	0%
Standard	81%	3%	15%	1%
Unique	64%	11%	20%	5%

(a) Error Recovery for Titan

Approach	Excel.	Good	Poor	Awful
Manual	87%	7%	5%	2%
Standard	67%	5%	27%	2%
Unique	55%	3%	3%	38%

(b) Error Recovery for C

Approach	Excel.	Good	Poor	Awful
Manual	80%	11%	9%	1%
Standard	80%	11%	9%	1%
Unique	61%	15%	24%	11%

(c) Error Recovery for Pascal

Approach	Excel.	Good	Poor	Awful
Manual	74%	17%	9%	0%
Standard	63%	16%	21%	0%
Unique	37%	3%	56%	4%

(d) Error Recovery for Java

$$
\begin{aligned}
\mathit{program} &\leftarrow (\mathit{toplevelfunc} \;/\; \mathit{toplevelvar} \;/\; \mathit{toplevelrecord} \;/\; \mathit{import} \;/\; \mathit{foreign})* \;\; [\texttt{EOF}]^{\texttt{Eof}_2} \\
\mathit{toplevelvar} &\leftarrow \mathit{localopt}\;\mathit{decl}\;[\texttt{ASSIGN}]^{\cancel{\texttt{AssignVar}}}\;[\texttt{exp}]^{\cancel{\texttt{ExpVarDec}}} \\
\mathit{import} &\leftarrow \texttt{LOCAL}\;[\texttt{NAME}]^{\cancel{\texttt{NameImport}}}\;[\texttt{ASSIGN}]^{\cancel{\texttt{AssignImport}}} \\
&\qquad [\texttt{IMPORT}]^{\cancel{\texttt{ImportImport}}}\;(\texttt{LPAR}\;[\texttt{STRING}]^{\texttt{StringLParImport}_3}\;[\texttt{RPAR}]^{\texttt{RParImport}_3}\;/\;[\texttt{STRING}]^{\texttt{StringImport}_3}) \\
\mathit{foreign} &\leftarrow \texttt{LOCAL}\;[\texttt{NAME}]^{\texttt{NameImport}_2}\;[\texttt{ASSIGN}]^{\texttt{AssignImport}_2} \\
&\qquad [\texttt{FOREIGN}]^{\texttt{Foreign}_2}\;[\texttt{IMPORT}]^{\texttt{ImportImport}_1}\;(\texttt{LPAR}\;[\texttt{STRING}]^{\texttt{StringLParImport}_1}\;[\texttt{RPAR}]^{\texttt{RParImport}_1}\;/\;[\texttt{STRING}]^{\texttt{StringImport}_1}) \\
\mathit{decl} &\leftarrow \texttt{NAME}\;(\texttt{COLON}\;[\mathit{type}]^{\cancel{\texttt{TypeDecl}}})? \\
\mathit{localopt} &\leftarrow \texttt{LOCAL}?
\end{aligned}
$$

Fig. 10. Excerpt of Titan Grammar Annotated by Algorithm Unique.

6.1. Titan

As we have mentioned in Section 4.1, the Titan grammar has 7 rules with non-$LL(1)$ choices and 3 rules with non-$LL(1)$ repetitions. After applying Algorithm Unique, we got a generated grammar[18] with 63 labels (around 75% of the labels added by manual annotation).

Algorithm Unique initially identified 44 unique lexical elements in the Titan grammar. Since that Algorithm Unique can annotate the first alternative of a non-disjoint choice when this alternative has a unique non-terminal that consumes input, we could add label `CastMissingType` in the rule below, where `AS` is a unique lexical non-terminal:

$$\mathit{castexp} \leftarrow \mathit{simpleexp}\;\texttt{AS}\;[\texttt{type}]^{\texttt{CastMissingType}} \;/\; \mathit{simpleexp}$$

Fig. 10 shows an excerpt of Titan grammar that we discussed in Section 4.1, without the predicates added by manual annotation. Non-terminal `FOREIGN` is unique, thus we can annotate the symbols that follow it. In Fig. 10, we represented as l_1 the labels added due to the uniqueness of `FOREIGN`.

To get a successful match, the start non-terminal must succeed, so we mark *program* as unique and thus we can annotate its right-hand side. In case of a repetition p_1*, expression p_1 should match at least one token before we can annotate it. As p_1 is a choice, where the same rationale for p_1* applies, we can not add labels to the alternatives in rule *program*.

The syntactical non-terminals *toplevelvar*, *import* and *foreign* are only used in rule *program*, so we could also mark them as unique and annotate their right-hand side. However, we will only annotate the right-hand side of *foreign*, because it is the last alternative of the non-disjoint choice in rule *program* involving these non-terminals. By marking *foreign* as unique, we can add the labels represented as l_2 in Fig. 10.

Finally, we can see in Fig. 10 that the lexical non-terminal `IMPORT` is used twice, so it is not unique. However, each use of `IMPORT` is preceded by a different context. In rule *import*, `ASSIGN` comes before `IMPORT`, while in rule *foreign* it is `FOREIGN` that precedes `IMPORT`. As we have different contexts, in rule *import* we can add the labels represented as l_3.

The labels represented as $\cancel{lab}$ were added by Algorithm Standard but not by Algorithm Unique. As we discussed in Section 4.1, labels `ExpVarDec`, in rule *toplevelvar*, and `ImportImport`, in rule *import*, should have not been added by Algorithm Standard. As expected, Algorithm Unique did not insert these labels incorrectly. We should notice that non-terminal `COLON` is used in other rules of the grammar, which were omitted here, so it is not unique.

[18] http://bit.ly/titan-unique-scp.

$$\begin{aligned} stat \leftarrow\ & \texttt{IF}\ [\texttt{LPAR}]^{\cancel{\texttt{BrackIf}}}\ [exp]^{\cancel{\texttt{InvalidExpr}}}\ [\texttt{RPAR}]^{\cancel{\texttt{Brack}}}\ [stat]^{\cancel{\texttt{Stat}}}\ \texttt{ELSE}\ [stat]^{\texttt{Stat}_1} \\ /\ & \texttt{IF}\ [\texttt{LPAR}]^{\texttt{BrackIf}_2}\ [exp]^{\texttt{InvalidExpr}_2}\ [\texttt{RPAR}]^{\texttt{Brack}_2}\ [stat]^{\texttt{Stat}_2} \end{aligned}$$

Fig. 11. *if-else* Statement Annotated by Algorithm Unique.

The error recovering parser generated by Algorithm Unique did an acceptable recovery for 75% of the test programs, while by manually annotating the grammar we could get an acceptable recovery for 95% of them.

6.2. C

Our unlabeled C grammar has 50 syntactical rules, from which 17 have non-$LL(1)$ choices and 5 have non-$LL(1)$ repetitions. Algorithm Unique was able to generate an error recovering parser[19] with 50 labels.

In case of our C grammar, Algorithm Unique added only 58% of the amount of labels inserted manually, while for Titan it could add 73% of this amount. The higher occurrence of non-disjoint expressions in the C grammar makes more difficult to mark symbols as unique and also to propagate a unique path after we have seen a unique symbol. In such grammars, to insert more labels it seems we need to do a more sophisticated analysis when computing the unique non-terminals.

Below, we revisit the *if-else* statement presented in Fig. 7 and discuss an extra analyses we did to mark an usage of `IF` as unique and helped us to achieve the amount of 50 labels. In Fig. 11, we used $\cancel{lab}$ to represent a label added by manual annotation but not by Algorithm Unique:

Initially, the only unique non-terminal is `ELSE`, which allows us to add just label $\texttt{Stat}_1$. Non-terminal `IF` was not considered unique at first because it is used twice, and both uses are preceded by the same context. To be able to annotate the second alternative of a non-disjoint choice such this, we check if the two usages of a non-terminal A with a context in common occur in the same right-hand side. If it is the case, we mark the last usage as unique. After doing this, we could add the labels represented as l_2 in Fig. 11.

We can see in Table 4b that the parser generated by Algorithm Unique performed an acceptable recovery for 58% of the test files, while by manually annotating the grammar we got a 94% rate of acceptable recovery for the same test files.

6.3. Pascal

As mentioned in Section 4.3, only 10 syntactical rules, out of 67, from the Pascal grammar have either a non-disjoint choice or a non-disjoint repetition. Because of this low number of non-disjoint expressions, the recursive banning approach discussed in Section 5.1 can annotate the Pascal grammar with 36 labels. By its turn, Algorithm Unique was able to add 72 labels.

As there are eight labels which were only added by the banning approach, in case of Pascal we automatically generated an error recovering parser[20] which joins the labels added by these two approaches and thus has 80 labels.

We can see in Table 3c that Algorithm Unique only inserted correct labels and was able to insert around 80% of labels added manually. In Table 4c we can see the resulting error recovering parser performs an acceptable recovery for 76% of the test files, while the parsers based on the other approaches perform such recovery for 91% of the test files.

A manual inspection revealed that the parser generated by Algorithm Unique built an AST with less information than the parser generated by Algorithm Standard for the files related to a label inserted only by Algorithm Standard, which shows we got a poorer recovery in these cases due to the missing labels.

6.4. Java

In case of our unlabeled Java grammar, where there is a non-disjoint expression in one third of the 147 grammar rules, Algorithm Unique generated a grammar[21] with 96 labels.

As was the case in our C grammar (Section 6.2), the higher occurrence of non-disjoint expressions in the Java grammar makes more difficult to annotate it. In case of Algorithm Standard, this leaded to adding 32 labels incorrectly, while in case of the recursive banning approach discussed in Section 5.1 this resulted in not adding a single label to the Java grammar. As Table 3d shows, Algorithm Unique was able to add 55% of the amount of labels added manually, without inserting labels incorrectly.

From Table 4d, we can see that the error recovering parser generated by Algorithm Unique only performed an acceptable recovery for 40% of the test files. This result was somehow expected, since that the algorithm failed to add many labels that were inserted during the manual annotation.

[19] http://bit.ly/c89-uniquescp.

[20] http://bit.ly/pascal-unique-scp.

[21] http://bit.ly/java8-unique-scp.

7. Related work

In this section, we discuss some error reporting and recovery approaches described in the literature or implemented by parser generators. Overall, a distinctive feature of our approach is that our error recovery mechanism is integrated with the recognizing formalism (PEGs, in our case).

Swierstra [16] shows a sophisticated implementation of parser combinators for error recovery. The recovery strategy uses information about the tails of the pending rules in the parser stack. When the parser fails to match a given symbol it may insert this symbol or to remove the current input symbol.

Our approach cannot simulate this recovery strategy, as it relies on the path that the parser dynamically took to reach the point of the error, while our recovery expressions are statically determined from the label. In Swiertra's approach, in case the right-hand side of the rules are not in some normal form, the parser may have a high memory consumption.

A popular error reporting approach applied for bottom-up parsing is based on associating an error message to a parse state and a lookahead token [17]. To determine the error associated to a parse state, it is necessary first to manually provide a sequence of tokens that lead the parser to that failure state. We can simulate this technique with the use of labels. By using labels we do not need to provide a sample invalid program for each label, but we need to annotate the grammar properly.

The error recovery approach for predictive top-down parsers proposed by Wirth [18] was a major influence for several tools. In Wirth's approach, when there is an error during the matching of a non-terminal A, we try to synchronize by using the symbols that can follow A plus the symbols that can follow any non-terminal B that we are currently trying to match (the procedure associated with B is on the stack). Moreover, the tokens which indicate the beginning of a structured element (e.g., `while`, `if`) or the beginning of a declaration (e.g., `var`, `function`) are used to synchronize with the input.

Our approach can simulate this recovery strategy just partially, because similarly to [19] it relies on information that will be available only during the parsing. We can define a recovery expression for a non-terminal A according to Wirth's idea, however, as we do not know statically how will be the stack when trying to match A, the recovery expression of A would use the $FOLLOW$ sets of all non-terminals whose right-hand side have A, and could possibly be on the stack.

Coco/R [20] is a tool that generates predictive $LL(k)$ parsers. As the parsers based on Coco/R do not backtrack, an error is signaled whenever a failure occurs. In case of PEGs, as a failure may not indicate an error, but the need to backtrack, in our approach we need to annotate a grammar with labels, a task we tried to make more automatic.

In Coco/R, in case of an error the parser reports it and continues until reaching a *synchronization point*, which can be specified in the grammar by the user through the use of a keyword `SYNC`. Usually, the beginning of a statement or a semicolon are good synchronization points.

Another complementary mechanism used by Coco/R for error recovery is weak tokens, which can be defined by a user though the `WEAK` keyword. A weak token is one that is often mistyped or missing, as a comma in a parameter list, which is frequently mistyped as a semicolon. When the parser fails to recognize a weak token, it tries to resume parsing based also on tokens that can follow the weak one.

Labeled failures plus recovery expressions can simulate the `SYNC` and `WEAK` keywords of Coco/R. Each use of `SYNC` keyword would correspond to a recovery expression that advances the input to that point, and this recovery expression would be used for all labels in the parsing extent of this synchronization point. A weak token can have a recovery expression that tries also to synchronize on its $FOLLOW$ set.

Coco/R avoids spurious error messages during synchronization by only reporting an error if at least two tokens have been recognized correctly since the last error. This is easily done in labeled PEG parsers through a separate post-processing step.

ANTLR [21,22] is a popular tool for generating top-down parsers. ANTLR automatically generates from a grammar description a parser with error reporting and recovery mechanisms, so the user does not need to annotate the grammar. After an error, ANTLR parses the entire input again to determine the error, which can lead to a poor performance when compared to our approach [4].

As its default recovery strategy, ANTLR attempts single token insertion and deletion to synchronize with the input. In case the remaining input can not be matched by any production of the current non-terminal, the parser consumes the input "*until it finds a token that could reasonably follow the current non-terminal*" [23]. ANTLR allows to modify the default error recovery approach, however, it does not seem to encourage the definition of a recovery strategy for a particular error, the same recovery approach is commonly used for the whole grammar.

A common way to implement error recovery in PEG parsers is to add an alternative to a failing expression, where this new alternative works as a fallback. Semantic actions are used for logging the error. This strategy is mentioned in the manual of Mouse [24] and also by users of LPeg.[22] These fallback expressions with semantic actions for error logging are similar to our recovery expressions and labels, but in an ad-hoc, implementation-specific way.

Several PEG implementations such as Parboiled,[23] Tatsu,[24] and PEGTL[25] provide features that facilitate error recovery.

[22] See http://lua-users.org/lists/lua-l/2008-09/msg00424.html.

[23] https://github.com/sirthias/parboiled/wiki.

[24] https://tatsu.readthedocs.io.

[25] https://github.com/taocpp/PEGTL.

The previous version of Parboiled used an error recovery strategy based on ANTLR's one, and requires parsing the input two or three times in case of an error. Similar to ANTLR, the strategy used by Parboiled was fully automated, and required neither manual intervention nor annotations in the grammar. Unlike ANTLR, it was not possible to modify the default error strategy. The current version of Parboiled[26] does not have an error recovery mechanism.

Tatsu uses the fallback alternative technique for error recovery, with the addition of a *skip expression*, which is a syntactic sugar for defining a pattern that consumes the input until the skip expression succeeds.

PEGTL (version 3) makes a distinction between a local failure (a regular failure), and a global failure, which is equivalent to throwing a label. The PEGTL user can use a function `raise` to produce a global failure, which is similar to annotate a grammar with labels.

Mizushima et al. [25] proposed the use of a cut operator, borrowed from Prolog, to avoid unnecessary backtracking in PEG parsers, and propose an automatic way to insert this operator in a grammar. Differently from the throw operator, which leads to a global failure, in case there is no recovery rule, the cut operator just discards the next alternative of the current choice, which makes difficult the use of cut operators to signal an error as the parser can still backtrack. The algorithm proposed by [25] to insert the cut operator is similar to Algorithm Standard. However, the former algorithm seems to do less insertions, as it does not annotate the second alternative of a choice, since there is no local backtracking to discard in this case.

Rüfenacht [26] proposes a local error handling strategy for PEGs. This strategy uses the farthest failure position and a record of the parser state to identify an error. Based on the information about an error, an appropriate recovery set is used. This set is formed by parsing expressions that match the input at or after the error location, and it is used to determine how to repair the input.

The approach proposed by Rüfenacht is also similar to the use of a recovery expression after an error, but more limited in the kind of recovery that it can do. When testing his approach in the context of a JSON grammar, which is simpler than grammar we analyzed, Rüfenacht noticed long running test cases and mentions the need to improve memory use and other performance issues.

The evaluation of our error recovery technique was based on Pennelo and DeRemmer's [27] strategy, which evaluates the quality of an error recovery approach based on the similarity of the program obtained after recovery with the intended program (without syntax errors). This quality measure was used to evaluate several strategies [28–30], although it is arguably subjective [30].

Differently from Pennelo and DeRemmer's approach, we did not compare programming texts, we compared the AST from an erroneous program after recovery with the AST of what would be an equivalent correct program.

8. Conclusion

We proposed algorithms to automate the process of adding error reporting and error recovery to parsers based on Parsing Expression Grammars. These algorithms annotate a PEG with error labels and associate recovery expressions for these labels.

We evaluated such algorithms on the grammars of four programming languages: Titan, C, Pascal and Java. For all these languages, we build a test suite both for valid and erroneous input.

Algorithm Standard could add to these grammars at least 75% of the labels added manually. The error recovering parser we got produced an acceptable recovery for at least 70% of the syntactically invalid files of each language.

The major limitation of Algorithm Standard is that it can annotate the right-hand side of a non-terminal A that is used either in a non-$LL(1)$ choice or in a non-$LL(1)$ repetition. This may prevent the parser from backtrack and recognize a valid input, thus changing the grammar language.

To address this issue, we proposed Algorithm Unique, which uses a more conservative approach, based on the idea of unique non-terminals. By using it, we inserted only correct labels and got an acceptable recovery rate that ranged from 41% to 76%.

We have also discussed how the rewriting of some grammar rules could lead both algorithms to produce a better result.

The automatic insertion of labels provides a good generic error reporting mechanism. To get more specific error messages, the parser developer just needs to associate an error message with each inserted label.

It is easy to adapt our algorithms to use a different error recovery strategy, which can also be defined after inserting the labels. It is also possible to adapt them to work on grammars that have already been partially annotated, either with just labels or labels and recovery expressions, as well as marking the parts of the grammar the algorithm should ignore and that will be annotated by hand by the parser developer.

To generate a more robust error recovering parsing, the approach based on unique tokens should insert more labels. One way to achieve this is by using the derivative of a PEG [31,32] to automatically generate valid inputs based on a grammar without annotations. After applying Algorithm Unique, we could repeatedly try to insert a label added only by Algorithm Standard and use the valid input generated through derivatives to determine whether the insertion of this label is correct or not.

[26] https://github.com/sirthias/parboiled2/.

As a future work, we should also explore other grammar analysis that may lead Algorithm Unique to insert more correct labels.

Moreover, we may investigate the use of some normal form when writing a PEG grammar to help our algorithms to produce a better result, without imposing too much restrictions for a grammar writer.

Finally, as the use of labeled failures may avoid unnecessary backtracking, we should also analyze the performance of the generated parsers.

References

[1] B. Ford, 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, New York, NY, USA, 2004, pp. 111–122.

[2] A.M. Maidl, F. Mascarenhas, R. Ierusalimschy, Exception handling for error reporting in parsing expression grammars, in: A.R. Du Bois, P. Trinder (Eds.), Programming Languages, in: LNCS, vol. 8129, Springer Berlin Heidelberg, 2013, pp. 1–15.

[3] A.M. Maidl, F. Mascarenhas, S. Medeiros, R. Ierusalimschy, Error reporting in parsing expression grammars, Sci. Comput. Program. 132 (P1) (2016) 129–140, https://doi.org/10.1016/j.scico.2016.08.004.

[4] S. Medeiros, F. Mascarenhas, Syntax error recovery in parsing expression grammars, in: Proceedings of the 33st Annual ACM Symposium on Applied Computing, SAC '18, ACM, New York, NY, USA, 2018, pp. 1195–1202.

[5] S.Q. de Medeiros, F. Mascarenhas, Error recovery in parsing expression grammars through labeled failures and its implementation based on a parsing machine, J. Vis. Lang. Comput. 49 (2018) 17–28, https://doi.org/10.1016/j.jvlc.2018.10.003, http://www.sciencedirect.com/science/article/pii/S1045926X18301897.

[6] S.Q. de Medeiros, F. Mascarenhas, Towards automatic error recovery in parsing expression grammars, in: Proceedings of the XXII Brazilian Symposium on Programming Languages, SBLP '18, ACM, New York, NY, USA, 2018, pp. 3–10.

[7] B. Ford, Packrat Parsing: a Practical Linear-Time Algorithm With Backtracking, Master's thesis, Massachusetts Institute of Technology, September 2002.

[8] R.R. Redziejowski, Applying classical concepts to parsing expression grammar, Fundam. Inform. 93 (2009) 325–336.

[9] R.R. Redziejowski, More about converting BNF to PEG, Fundam. Inform. 133 (2014) 257–270, https://doi.org/10.3233/FI-2014-1075.

[10] F. Mascarenhas, S. Medeiros, R. Ierusalimschy, On the relation between context-free grammars and parsing expression grammars, Sci. Comput. Program. 89 (2014) 235–250, https://doi.org/10.1016/j.scico.2014.01.012, http://www.sciencedirect.com/science/article/pii/S0167642314000276.

[11] T.T. Developers, The Titan programming language, http://titan-lang.org/, 2017 [Visited on September 2019].

[12] R. Ierusalimschy, Programming in Lua, 4th edition, Lua.Org, 2016.

[13] D. Grune, C.J. Jacobs, Parsing Techniques: A Practical Guide, 2nd edition, Springer Publishing Company, Incorporated, 2010.

[14] B.W. Kernighan, D.M. Ritchie, The C Programming Language, Prentice Hall Press, Upper Saddle River, NJ, USA, 1988, the complete C grammar is given in Section A13.

[15] ISO Central Secretary, Information Technology – Programming Languages – Pascal, Standard ISO/IEC 7185:1990(E), International Organization for Standardization, Geneva, CH, the complete Pascal grammar is given in Annex A, 1991, https://www.iso.org/standard/13802.html.

[16] S.D. Swierstra, Combinator parsers: from toys to tools, Electron. Notes Theor. Comput. Sci. 41 (1) (2001) 38–59.

[17] C.L. Jeffery, Generating LR syntax error messages from examples, ACM Trans. Program. Lang. Syst. 25 (5) (2003) 631–640.

[18] N. Wirth, Algorithms + Data Structures = Programs, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1978.

[19] S.D. Swierstra, L. Duponcheel, Deterministic, error-correcting combinator parsers, in: Advanced Functional Programming, in: Lecture Notes in Computer Science, vol. 1129, Springer, 1996, pp. 184–207.

[20] H. Mössenböck, The Compiler Generator Coco/R, 2010, http://www.ssw.uni-linz.ac.at/Coco/Doc/UserManual.pdf.

[21] T. Parr, ANTLR, http://www.antlr.org, 2014 [Visited on September 2019].

[22] T. Par, The Definitive ANTLR 4 Reference, 2nd edition, Pragmatic Bookshelf, 2013.

[23] T. Parr, S. Harwell, K. Fisher, Adaptive ll(*) parsing: the power of dynamic analysis, in: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14, ACM, New York, NY, USA, 2014, pp. 579–598.

[24] R.R. Redziejowski, Mouse: From Parsing Expressions to a Practical Parser, 2017, http://mousepeg.sourceforge.net/Manual.pdf.

[25] K. Mizushima, A. Maeda, Y. Yamaguchi, Packrat parsers can handle practical grammars in mostly constant space, in: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '10, ACM, New York, NY, USA, 2010, pp. 29–36.

[26] M. Rüfenacht, Error Handling in Peg Parsers, Master's thesis, University of Berne, 2016.

[27] T.J. Pennello, F. DeRemer, A forward move algorithm for lr error recovery, in: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '78, ACM, New York, NY, USA, 1978, pp. 241–254.

[28] R. Corchuelo, J.A. Pérez, A. Ruiz, M. Toro, Repairing syntax errors in lr parsers, ACM Trans. Program. Lang. Syst. 24 (6) (2002) 698–710, https://doi.org/10.1145/586088.586092.

[29] P. Degano, C. Priami, Comparison of syntactic error handling in lr parsers, Softw. Pract. Exp. 25 (6) (1995) 657–679, https://doi.org/10.1002/spe.4380250606.

[30] M. de Jonge, L.C.L. Kats, E. Visser, E. Söderberg, Natural and flexible error recovery for generated modular language environments, ACM Trans. Program. Lang. Syst. 34 (4) (2012) 15:1–15:50, https://doi.org/10.1145/2400676.2400678.

[31] A. Moss, Simplified parsing expression derivatives, CoRR, arXiv:1808.08893, http://arxiv.org/abs/1808.08893.

[32] T. Garnock-Jones, M. Eslamimehr, A. Warth, Recognising and generating terms using derivatives of parsing expression grammars, CoRR, arXiv:1801.10490, http://arxiv.org/abs/1801.10490.