Error recovery in parsing expression grammars through labeled failures and its implementation based on a parsing machine

Sérgio Queiroz de Medeiros⁎,¹, Fabio Mascarenhas²

¹ School of Science and Technology, UFRN, Natal, Brazil
² Department of Computer Science, UFRJ, Rio de Janeiro, Brazil

ABSTRACT

Parsing Expression Grammars (PEGs) are a formalism used to describe top-down parsers with backtracking. As PEGs do not provide a good error recovery mechanism, PEG-based parsers usually do not recover from syntax errors in the input, or recover from syntax errors using ad-hoc, implementation-specific features. The lack of proper error recovery makes PEG parsers unsuitable for use with Integrated Development Environments (IDEs), which need to build syntactic trees even for incomplete, syntactically invalid programs.

We discuss a conservative extension, based on PEGs with labeled failures, that adds a syntax error recovery mechanism for PEGs. This extension associates recovery expressions to labels, where a label now not only reports a syntax error but also uses this recovery expression to reach a synchronization point in the input and resume parsing. We give an operational semantics of PEGs with this recovery mechanism, as well as an operational semantics for a parsing machine that we can translate labeled PEGs with error recovery to, and prove the correctness of this translation. We use an implementation of labeled PEGs with error recovery via a parsing machine to build robust parsers, which use different recovery strategies, for the Lua language. We evaluate the effectiveness of these parsers, alone and in comparison with a Lua parser with automatic error recovery generated by ANTLR, a popular parser generator.

1. Introduction

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

More specifically, a PEG can be interpreted as the specification of a recursive descent parser with restricted (or local) backtracking. This means that 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 try the next alternative.

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. While PEGs cannot use error handling techniques that are often applied to predictive top-down parsers, because these techniques assume the parser reads the input without backtracking [2,3], some techniques for correctly reporting syntactic errors in PEG parsers have been proposed, such as tracking the position of the farthest failure [2] and labeled failures [4,5].

While these error reporting techniques improve the quality of error reporting in PEG parsers, they all assume that the parser aborts after reporting the first syntax error. We believe this is acceptable for a large class of parsers, however Integrated Development Environments (IDEs) often require parsers that can recover from syntax errors and build Abstract Syntax Trees (ASTs) even for syntactically invalid programs, in order to conduct further analyses necessary for IDE features such as auto-completion. These parsers should also be fast, as the user of an IDE expects an almost instantaneous feedback.

Some PEG parser generators already provide ad-hoc mechanisms that can be exploited to perform error recovery⁶, but the mechanisms are specific to each implementation, tying the grammar to a specific
implementation. To address this issue, we proposed in a prior work a conservative extension of PEGs, based on labeled failures, that adds a recovery mechanism to the PEG formalism itself [6]. The mechanism attaches recovery expressions to labels so that throwing those labels not only reports syntax errors but also skips the erroneous input until reaching a synchronization point and resuming parsing.

In this paper we present a parsing machine for PEGs that can implement our error recovery approach. Each PEG is translated to a program that is executed by a virtual parsing machine. A formal semantics of such a parsing machine for regular PEGs already exists [7], as well as an extension of this semantics for labeled PEGs [8]. We extend this semantics with farthest failure tracking and error recovery, and prove the correctness of our translation of labeled PEGs with recovery to this extended machine.

We use an implementation of this extended parsing machine to build robust parsers, which use different recovery strategies, for the Lua language. Then we compare the error recovery behavior of these parsers with a Lua parser generated with ANTLR [9,10], a popular parsing tool based on a top-down approach, by evaluating the AST built by each one. This comparison is broader than the comparison done previously [6], since it implements different recovery strategies based on labeled failures, and is more precise too, because it compares ASTs directly instead of comparing error messages.

In this extended version of our previous work [6], we also present a more detailed discussion about other error recovery approaches, and we provide links for our implementations.

The remainder of this paper is organized as follows: the next section (Section 2) revisits the error handling problem in PEG parsers and introduces the semantics of labeled PEGs that supports syntactic error recovery; Section 3 discusses error recovery strategies that PEG-based parsers can implement using our recovery mechanism; Section 4 gives the semantics of the extended parsing machine for labeled PEGs with failure tracking and error recovery; Section 5 evaluates our error recovery approach by comparing PEG-based parsers for the Lua language with an ANTLR-generated parser; Section 6 discusses related work on error recovery for top-down parsers with backtracking; finally, Section 7 gives some concluding remarks.

2. PEGs with error recovery

In this section, we revisit the problem of error handling in PEGs, and show how labeled failures [4,5] combined with the farthest failure heuristic [2] can improve the error messages of a PEG-based parser. Then we show how labeled PEGs can be the basis of an error recovery mechanism for PEGs, and show an extension of previous semantics for labeled PEGs that adds recovery expressions.

2.1. PEGs and error reporting

A PEG $G$ is a tuple $(V, T, P, p_0)$ 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 and $p_0$ is the initial parsing expression. We describe the function $P$ as a set of rules of the form $A \rightarrow p$, where $A \in V$ and $p$ is a parsing expression. A parsing expression, when applied to an input string, either fails or consumes a prefix of the input and returns the remaining suffix. The abstract syntax of parsing expressions is given as follows, where $a$ is a terminal, $A$ is a non-terminal, and $p_1$ and $p_2$ are parsing expressions:

$$ p = \varepsilon | a | A | p_1p_2 | p_1 / p_2 | p^* | p^{|} $$

Intuitively, $\varepsilon$ successfully matches the empty string, not changing the input; $a$ matches and consumes itself or fails otherwise; $A$ tries to match the expression $P(A)$; $p_1p_2$ tries to match $p_1$ followed by $p_2$; $p_1 / p_2$ tries to match $p_1$; if $p_1$ fails, then it tries to match $p_2$; $p^*$ repeatedly matches $p$ until $p$ fails, that is, it consumes as much as it can from the input; the matching of $p^{|}$ succeeds if the input does not match $p$ and fails when the input matches $p$, not consuming any input in either case; we call it the negative predicate or the lookahead predicate.

Fig. 1 shows a PEG for a tiny subset of Java, where lexical rules (shown in uppercase) have been elided. While simple (this PEG is equivalent to an LL(1) CFG), this subset is already rich enough to show the problems of PEG error reporting; a more complex grammar for a larger language just compounds these problems.

Fig. 2 is an example of a Java program with two syntax errors (a missing semicolon at the end of line 7, and an extra semicolon at the end of line 8). A predictive top-down parser will detect the first error when reading the RCUR (;) token at the beginning of line 8, and will know and report to the user that it was expecting a semicolon.

In the case of our PEG, it will still fail when trying to parse the SEMI rule, which should match a ‘;’, while the input has a closing curly bracket, but as a failure does not guarantee the presence of an error the parser cannot report this to the user. Failure during parsing of a PEG usually just means that the PEG should backtrack and try a different alternative in an ordered choice, or end a repetition. For example, three failures will occur while trying to match the BlockStmt rule inside Prog against the $n$ at the beginning of line 3, first against IF in the IfStmt rule, then against WHILE in the WhileStmt rule, and finally against PRINTLN in the PrintStmt rule.

After all the failing and backtracking, the PEG in our example will ultimately fail in the RCUR rule of the initial BlockStmt, after consuming only the first two statements of the body of main. Failing to match the SEMI in AssignStmt against the closing curly bracket in the input will make the PEG backtrack to the beginning of the statement to try the other alternatives in Stmt, which also fail. This marks the end of the repetition inside the BlockStmt that is parsing the body of the while statement. The whole BlockStmt will fail trying to match RCUR against the $n$ in the beginning of line 7, this ultimately makes the whole WhileStmt fail, which makes the PEG backtrack to the beginning of line 7, resulting in the input not matching any of the rules of the initial BlockStmt.
Now the process repeats with the `BlockStmt` that is parsing the body of `main`.

In the end, the PEG will report that it failed and cannot proceed at the beginning of line 5, complaining that the `while` in the input does not match the `RCUR` that it expects, which does not help the programmer in finding and fixing the actual error.

To circumvent this problem, Ford [2] suggested that the furthest position in the input where a failure has occurred should be used for reporting an error. A similar approach for top-down parsers with backtracking was also suggested by Grune and Jacobs [11].

In our previous example, the use of the furthest failure approach reports an error at the beginning of line 8, the same as a predictive parser would. We can even use a map of lexical rules to token names to track expected tokens in the error position, and report that a semicolon was expected.

If the programmer fixes this error, the parser will then fail repeatedly at the extra semicolon at line 8, while trying to match the first term of all the alternatives of `Stmt`. This will end the repetition inside `BlockStmt`, and then another failure will happen when trying to match a `RCUR` token against the semicolon, finally aborting the parse. The parser can use the furthest failure information to report an error at the exact position of the semicolon, and a list of expected tokens that includes `IF`, `WHILE`, `NAME`, `LCUR`, `PRINTLN`, and `RCUR`.

The great advantage of using the furthest failure is that the grammar writer does not need to do anything to get a parser with better error reporting, as the error messages can be generated automatically. However, although this approach gives us error messages with a fine approximation of the error location, these messages may not give a good clue about how to fix the error, and may contain a long list of expected tokens [5].

We can get more precise error messages at the cost of manually annotating the PEG with labeled failures, a conservative extension of the PEG formalism. A labeled PEG `G` is a tuple `(V, T, P, L, fail, p1)` where `L` is a finite set of labels, `fail` is a failure label, and the expressions in `P` have been extended with the `throw` operator, represented by `@`. The parsing expression `@`, where `l ∈ L`, generates a failure with label `l`.

A label `l` is fail thrown by `@` cannot be caught by an ordered choice, so it indicates an actual error during parsing, while `fail` is caught by a choice and indicates that the parser should backtrack. The lookahead operator `!` captures any label and turns it into a success, while turning a success into a `fail` label.

We can map different labels to different error messages, and then annotate our PEG with these labels. Fig. 3 annotates the PEG of Fig. 1 (except for the `Prog` rule). The expression `[p]` is syntactic sugar for `(p / @)`.

The strategy we used to annotate the grammar was the following: on the right-hand side of a production, we annotate every symbol (terminal or non-terminal) that should not fail, that is, making the PEG backtrack on failure of that symbol would be useless, as the whole parse would either fail or not consume the whole input in that case. For an LL(1) grammar like the one in our example, that means all symbols in the right-hand side of a production except the one in the very beginning of the production. We apply a similar rule when the right-hand side has a choice or a repetition as a subexpression.

Using this labeled PEG in our program, the first syntax error now fails directly with a `semia` label, which we can map to a “missing semicolon in assignment” message. If the programmer fixes this, the second error will fail with a `rcblk` label, which we can map to a “missing end of block” message.

Compared with the furthest failure approach, one drawback of labeled failures is the annotation burden. But we can combine both approaches, and still track the position, and set of expected lexical rules, of the furthest simple failure. The parser can fall back on automatically generated error messages whenever parsing fails without producing a more specific error label.

### 2.2. Error recovery

The labeled PEGs with furthest failure tracking we described in the previous section make it easier to report the first syntax error found in a PEG, and we will use them as the first step towards an error recovery mechanism.

Before giving the full formal definition of PEGs with error recovery, let us return to the example program in Fig. 2 and its two syntax errors: a missing semicolon at the end of line 7, and an extra semicolon at the end of line 8. The labeled PEG of Fig. 3 throws the label `semia` when it finds the first error, and finishes parsing.

If every syntactic error is labeled, to recover from them we need to do the following: first, catch the label right after it is thrown, before the parser aborts, then log this error, possibly skip part of the input, and finally resume parsing. In our example, for the first error we just need to log the error, continue as if the semicolon was found, and for the second error we need to log the error, skip until finding the end of a block (taking care with nested blocks on the way), and then resume.

To achieve this, we extend labeled PEGs with a list of recovered errors and a map of labels to `recovery expressions`. These recovery expressions are responsible for skipping tokens in the input until finding a place where parsing can continue.

Fig. 4 presents the semantics of labeled PEGs with error recovery as a set of inference rules for a `PEG` function. The notation `G[p] R xy !` (y, v?, E) represents a successful match 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`. The term `v?` is information for tracking the location of the furthest failure, and denotes either a suffix `v` of the original input or `nil`. `E` is a list of pairs of a label and a suffix of the original input, denoting errors that were logged and recovered. For an unsuccessful match the first element of the resulting triple is `nil`, or `fail`, denoting a label.

The auxiliary function `smallest` that appears on Fig. 4 compares two possible error positions, denoted by a suffix of the input string, or `nil` if no failure has occurred, and returns the furthest: any suffix of the input is a further possible error position than `nil` and a shorter suffix is a further possible error position than a longer suffix.

Most of the rules are conservative extensions of the rules for labeled PEGs [5,8], where the recovery map `R` is simply passed along, and any lists of recovered errors are concatenated. The exceptions are the rules for the syntactic predicate and for throwing labels.

The syntactic predicate turns any failure label into a success, using an empty recovered map to make sure that errors are not recovered inside the predicate. Failure tracking information is also thrown away. In essence, any error that happens inside a syntactic predicate is expected, and not considered a syntax error in the input.

**Rule throw.1** is related to error reporting, while rules **throw.2** and **throw.3** are 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**), we just append the label and current position to `E` and propagate the error upwards, so parsing finishes after reaching the first syntactical error.

If label `l` has a recovery expression `R(l)`, we append the current error to the list `L` of errors and try to match the current input by using `R(l)`. Rule **throw.2** deals with the case where the matching of `R(l)` succeeds, and rule **throw.3** deals with the case where `R(l)` fails. Intuitively, rule **throw.2** applies when the parser recovers from an error and regular parsing is resumed after, while we use rule **throw.3** when another error (or a regular failure) happens during recovery, which may finish the
As a recovery expression, this one does not consume the synchronization symbols, as they should be consumed by the following expression.

The recovery expression above could be automatically computed from FOLLOW(Exp) and associated with labels condi, condw, edec, rval, eprint, and parexp. Another option is to compute a specific FOLLOW set for each use of Exp. For example, the FOLLOW set associated with Exp in rules DecStmt and AssignStmt contains only SEMI, while the FOLLOW set associated with Exp in IfStmt, WhileStmt, AtomExp, and PrintStmt contains only RCUR.

Fig. 5 extends our annotated PEG for a subset of Java with recovery rules computed automatically. For each expression of the form \( \text{p} \), we added a recovery rule of the form \( l \rightarrow ((\text{FOLLOW}(\text{p})).)\) . In Fig. 5, the symbol EOF represents the end of input. The recovery expression associated with labels then, else, body, semid, semia, and semip is the same. For the sake of brevity, Fig. 5 shows the recovery rule for label then and omits the recovery rules for these other labels. In a similar way, this figure shows only the recovery rule for label condi and omits the recovery rules for condw, eprint, and parexp. This use of the FOLLOW set (possibly enhanced by some usual synchronization symbols, such as ‘;’) provides a default error recovery strategy.

Let us consider that the Java program from Fig. 2 has an error on line 5, inside the condition of while loop, as follows:

Our default error recovery strategy will report this error and resume parsing correctly at the following right parenthesis. In the resulting AST, the node for the while loop will have an empty condition, so we lose the node corresponding to the use of the \( n \) variable, and the information that the condition was a \( < \) expression.

Now let us consider the default error recovery strategy for label rclbk. A BlockStmt can be followed by a Statement or by RCUR, or it can be the last statement of a program. Therefore, as we can see in Fig. 5, the default recovery expression for rclbk synchronizes with a token that indicates the beginning of a Statement, an else block, a ‘)’, or the end of input.

Unfortunately, this recovery strategy is not good for rclbk, as our example program from Fig. 2 shows. The recovery expression for rclbk will consume the ‘;’ at the end of line 8, and then stop at the beginning of the next statement on line 9. But the parser just closed the BlockStmt of the main function of the program, and now expects another ‘;’ to close the class body in Prog. This will lead to a spurious error when the parser finds the beginning of the print statement, so a custom SkipToRCUR recovery expression is a better way to deal with rclbk errors.

While SkipToRCUR avoids spurious errors, it does have the potential
Empty

\[ G[x] \xrightarrow{PEG} (x, \text{nil}, []) \]

Non-terminal

\[ G[P(A)] \xrightarrow{PEG} (y, v?, E) \]
\[ G[A] \xrightarrow{PEG} (y, v?, E) \]

Terminal

\[ G[a] \xrightarrow{PEG} (x, \text{nil}, []) \]

Sequence

\[ G[p_1] \xrightarrow{PEG} (yz, v?, E_1) \]
\[ G[p_2] \xrightarrow{PEG} (z, w?, E_2) \]

Repetition

\[ G[p] \xrightarrow{PEG} (l, y, E) \]
\[ l \neq \text{fail} \]

Negative Predicate

\[ G[p] \xrightarrow{PEG} (l, y, E) \]
\[ l \neq \text{fail} \]

Ordered Choice

\[ G[p_1] \xrightarrow{PEG} (y, v?, E) \]
\[ G[p_1 / p_2] \xrightarrow{PEG} (y, v?, E) \]

Recovery

\[ l \notin \text{Dom}(R) \]
\[ G[R(l)] \xrightarrow{PEG} (y, v?, E) \]

Fig. 4. Semantics of PEGs with labels, recovery and farthest failure tracking.

to skip a large portion of the input, leading to a poor AST. We can improve this by noticing that \texttt{Stmt} inside the repetition of \texttt{BlockStmt} is not allowed to fail unless the next token is \texttt{RCUR}, so we can replace \texttt{Stmt} with \texttt{RCUR [Stmt][stmtb]}. Now the second error in our program will make parsing fail with a \texttt{stmtb} label. The recovery expression of this label can synchronize with the beginning of the next statement, or ‘)’. In our example, this will skip the erroneous ‘)’ at the end of line 8 and then continue parsing the rest of the block.

Finally, we have the full power of PEGs inside recovery expressions, and can use it for more elaborate recovery strategies. Going back to the error in the condition of a \texttt{while} loop earlier in this section, we can, instead of blindly skipping tokens until finding the closing ‘)’, try to see if we have a partial relational expression before giving up with the following recovery expression for \texttt{condw}:

\texttt{!!EQ \{RelExp\} / !!LT \{AddExp\} / (!RPAR . )}

The double negation is an and syntactic predicate, and is a way of guarding an expression so it will only be tried if its beginning matches the guard.

In the next section, we discuss how to automatically annotate a PEG with labels, making it possible to automate some of the error recovery strategies that we discussed in this section.
4. A parsing machine for labeled PEGs

This section discusses how we can extend an implementation of PEGs based on a parsing machine to provide error reporting and recovery facilities. Parsing machines are a way to efficiently implement PEG-based parsers that can be constructed dynamically at runtime [7,15]. In parsing machine-based implementations each parsing expression is compiled to instructions of a virtual parsing machine; the program for a compound expression is a combination of the programs for its subexpressions. The result program can be interpreted using the same techniques that virtual machines for programming languages use, and even Just-In-Time compiled to machine code. The performance of a parsing machine interpreter is comparable to the performance of parsers generated by other approaches [7,15,16].

In its abstract definition [7], the parsing machine has two registers and a stack. One register holds the program counter, used to address the next instruction to execute, and one register holds the current subject (an input suffix). The stack holds call and backtrack frames. A call frame is just a return address for the program counter, and a backtrack frame is an address for the program counter and a subject to backtrack to. The semantics of each instruction is given as its effect on the registers and on the stack.

Formally, the program counter register, the subject register, and the stack form a machine state. We represent it as a tuple $\langle e, N \times T^* \times \mathbb{N} \rangle$, where $e$ is the stack. Stacks are lists of $(N \times T^*) \cup N$, where $N \times T^*$ represents a backtrack frame and $N$ represents a call frame.

The behavior of each instruction is straightforward: $Call$ pushes a call frame with the address of the next instruction and jumps to an instruction with a given label$^3$; $Return$ pops a call frame and jumps to

---

Fig. 5. A PEG with labels and recovery rules for a small subset of Java.
its address. Jump is an unconditional jump, Char tries to match a character with the first character of the subject, consuming it if successful and entering a failure state otherwise. When the machine enters a failure state it pops call frames from the stack until reaching a backtrack frame, then pops this frame and resumes execution with the subject and address stored in it. Any consumes the first character of the subject (failing if the subject is ε), Choice pushes a backtrack frame with the current subject and the address associated with a given label, Commit discards the backtrack frame in the top of the stack and jumps to a given label, and Fail unconditionally enters a failure state.

As an example, the following PEG (that matches a sequence of zero or more as, followed by any character other than a, followed by a b) uses all basic instructions of the parsing machine, where ‘.’ is a parsing expression that matches any character:

\[
\begin{align*}
A & \leftarrow !a \cdot B \mid aA \\
B & \leftarrow b
\end{align*}
\]

This PEG compiles to the following program:

\[
\begin{align*}
A & |
\text{Choice} & A1 \\
& |
\text{Choice} & A2 \\
& |
\text{Char} & a \\
& |
\text{Commit} & A3 \\
A3 & |
\text{Fail} & \\
A2 & |
\text{Any} & \\
& |
\text{Call} & B \\
& |
\text{Commit} & A4 \\
A1 & |
\text{Char} & a \\
& |
\text{Jump} & A \\
A4 & |
\text{Return} & \\
B & |
\text{Char} & b \\
& |
\text{Return} &
\end{align*}
\]

We want to extend the parsing machine to support error reporting and recovery. Let us start by adding error reporting features. The machine state becomes \( \mathbb{N} \times \mathbb{T}^* \times \text{Stack} \times (\mathbb{T}^* \cup \{\text{nil}\}) \), where the extra \((\mathbb{T}^* \cup \{\text{nil}\})\) term keeps the information about the farthest failure. We then update instructions Char and Any to track the farthest failure position.

As we also want to keep information about the label associated with an error we will represent a failure state as \( \text{Fail}(l, v, \epsilon) \), where \( l \) is a label, \( v \) represents (possibly nil) suffix of the subject associated with the farthest failure, and \( \epsilon \) is the stack. We also rename the Fail instruction to Throw, where Throw takes a label \( l \) as an operand. With the exception of the Throw instruction, all instructions that can fail will produce a failure state where \( l \) is fail.

Fig. 6 presents the operational semantics of the parsing machine as a relation between machine states. The program \( \mathcal{P} \) that the machine executes is implicit. The relation \( \rightarrow \) relates two states when \( p \) in the first state addresses a instruction matching the name above the \( \rightarrow \), and the guard (if present) is valid.

When the Char instruction fails, in case the subject is not \( \epsilon \) it uses smallest to update the suffix associated with the farthest failure position, otherwise, the \( \epsilon \) itself becomes the suffix associated with the farther failure position. The Any instruction also updates the suffix associated with the farthest failure position accordingly.

The Throw instruction sets the current position as the position of the farthest failure. Although the current position may not represent the farthest failure, it indicates the position where label \( f \) was thrown. It is easy to see that right now the machine can only recover from a failure state if its label is fail and it finds a backtrack frame on the top of the stack.

To add error recovery, we extend the machine state with a map \( \mathcal{R} \)

(just as we would do before the introduction of labels may not be valid anymore. We will not discuss this topic here, as our previous paper on a parsing machine for labeled PEGs (without recovery) already has a detailed discussion [8]).

4.1. Correctness of the translation

The following lemma gives the correctness condition for the transformation II with regards to the semantics of PEGs with labeled failures but without recovery, by proving that a program for the parsing machine obtained using the translation function II always produces the
implies that $(p, x, e, v?)$.

Lemma 4.2 (Correctness of $\mathit{PEG}$) Given a PEG with labels $G$, a map $R$, and a subject $xy$, we have that $brevity$: for the syntactic predicate $\mathit{PEG}$.

Proof. Given in the paper that describes the parsing machine for labeled PEGs [8].

In order to establish the correctness of our extended machine, we need to prove the following equivalent lemma, which takes utmost failure tracking and recovery expressions into account:

Lemma 4.3 (Correctness of $\mathit{PEG}$) Given a PEG with labels $G$, a map $R$, and a subject $xy$, we have that $G[p]\ xy \Rightarrow y$ implies that $(p, x, e, v?)$.

Proof. By induction on the height of the proof trees for the antecedents. Most cases are similar to the ones in the proofs of the previous lemma and similar lemma for the original parsing machine [7]. The ones that deviate are the cases where the expression is a syntactic predicate $\mathit{PEG}$ and for $\mathit{PEG}$, but those are still straightforward and have been elided for brevity: for the syntactic predicate $\mathit{PEG}$.

5. Evaluation

In this section, we evaluate our syntax error recovery approach for PEGs using a complete parser for an existing programming language in two different contexts, first in isolation and then by comparison with a parser generated by ANTLR, a mature parser generator that uses predictive parsing.

5.1. Error recovery in a Lua parser

It seems there is not a consensus about how to evaluate an error recovery strategy. Ripley and Druseks [17] collected a set of syntactic invalid Pascal programs that was used to evaluate some error recovery strategies [18–21]. However, as far as we know, this set of programs is not publicly available.

Another issue related to the evaluation of an error recovery strategy is how to measure its quality. Pennello and DeRemmer [18] proposed a criteria based on the similarity of the program got after recovery with the intended program (without syntax errors). This quality measure was used to evaluate several strategies [21–23], although it is arguably subjective [23].

We will evaluate our strategy following Pennello and DeRemmer’s approach, however instead of comparing programming texts we will...
compare the AST from an erroneous program after recovery with the AST of what would be an equivalent correct program. For the leaves of our AST 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, does not report spurious errors, and does not miss other errors. A poor recovery, by its turn, produces an AST that loses too much information, results in spurious errors, or misses errors. Finally, a recovery is rated as failed whenever it fails to produce an AST at all.

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

A recovery would be excellent in case the AST has all the information associated with this program, where the AST may have a dummy expression node to represent the range start. A recovery would be good in case the resulting AST misses only the loop range end and no spurious error is reported. By its turn, a recovery would be rated as poor either in case the resulting AST misses the statements inside the for (lines 3 and 4), or it produces spurious error messages. Lastly, we would rate a recovery as failed in case it would have not even produced an AST for the previous program.

To evaluate our error recovery strategy, we adapted an existing PEG parser for the Lua programming language [24] using the LPegLabel tool [25]. Our parser was based on the parser available at https://github.com/sqmedeiros/lua-parser, which targets the syntax defined in the Lua 5.3 reference manual4, and builds the AST associated with a given program.

The Lua grammar of our parser based on LPegLabel has 75 different labels, which were added manually, and 80 expressions involving the if operator (most labels are thrown only once). When implementing error recovery for our Lua parser we used three different error recovery strategies, one that required manual intervention and two others based on FOLLOW-set error recovery [26], which could be implemented automatically. The difference between these implementations resides only on the recovery expression each one associates to a label, once that all of them are based on the same Lua grammar annotated with 75 different labels. The implementation of our error recovering Lua parser is available at https://github.com/sqmedeiros/lua-parser/releases/tag/comlan.

Initially, we defined a small set of default recovery expressions, based on what would be good recovery tokens for the Lua grammar, and we associated one of these expressions with each label of our grammar. Then, while testing our recovery strategy we wrote some custom recovery expressions in order to avoid spurious error messages or to build a better AST. We will refer to this version of the Lua parser as custom. This version uses around 25 different recovery expressions.

After we implemented recovery strategies based on the use of FOLLOW sets, by using these sets to build the recovery expressions associated to each label. These recovery expressions try to be synchronized by using the tokens in the FOLLOW set of the current non-terminal the parser was matching when the label was thrown. We name as global the version of the Lua parser that builds recovery expressions based on global FOLLOW sets, and as local to the version based on local FOLLOW sets. A non-terminal A has only one global FOLLOW set associated with it, which takes into consideration all occurrences of A in the right-hand side of the grammar productions. On the other hand, a non-terminal A may have several local FOLLOW sets, where each set only has the symbols that can follow A in the right-hand side of a specific production where A occurs.

We used 180 syntactically invalid Lua programs to test our error recovery mechanism. In a general way, each program should cause the throwing of a specific label, to test whether the associated recovery expression recovers well. We usually wrote more than one erroneous Lua program to test each label.

Table 1 shows for how many programs the recovery strategies we implemented were considered excellent, good, poor, or failed. As we can see, the use of labels plus the recovery operator enabled us to implement PEG parsers for the Lua language with a robust recovery mechanism. In our evaluation approach, for all three parsers more than 90% of the recovery done was considered acceptable, i.e., it was rated at least good. In Section 5.2, we will discuss the row named antlr.

Our parsers were always able to build an AST, given that no recovery expression raised an unrecoverable error, or entered a loop. These properties can be conservatively checked, as indicated by Ford [1].

We have also compared the number of error messages generated by each error recovery approach for the 180 syntactically invalid Lua programs. Table 2 shows that the Lua parsers based on a FOLLOW-set error recovery usually report more errors than the other Lua parser. In the next section we discuss the row named antlr.

A possible explanation for this resides in the fact that our custom recovery expressions use a few tokens to synchronize the parser, while the corresponding FOLLOW set usually has more tokens. The use of a richer set of tokens seems to slightly improve synchronization in general, at the cost of sometimes reporting more spurious error messages. For most programs, all approaches report the same amount of error messages.

5.2. Comparison with ANTLR

ANTLR [27,28] is a popular tool for generating top-down parsers. The repository of ANTLR at GitHub contains the implementation of several parsers, including a parser for Lua 5.3. We used ANTLR 4 for our comparison.

Unlike LPegLabel, 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. The error recovery mechanism of ANTLR is based on early ideas of Niklaus Wirth [29], though it has its own distinctive features.

In general, after an error an ANTLR parser attempts to re-synchronize by trying single token insertion and deletion. 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” [10]. ANTLR allows to modify its default error recovery approach, and also allows to define a recovery strategy for a particular grammar rule, although this does not seem usual in ANTLR 4. A more detailed description of ANTLR 4 recovery mechanism is given in [28]. In our evaluation we used the default ANTLR error recovery strategy.

Based on the available ANTLR parser for Lua, we implemented a visitor pattern in order to get the AST associated with a given Lua program. This implementation is available at https://github.com/sqmedeiros/antlrlua.

In Table 1, we can see that for our set of 180 syntactic invalid Lua programs the error recovery done by the ANTLR parser was acceptable for around 76% of them, while for one quarter of these files the recovery was rated as poor.

An analysis of the test files where the Lua parsers based on LPegLabel built a richer AST seems to indicate that the ANTLR parser does not recover well when there is a missing token or expression in a for or if statement. For example, in the following program the expression after the comma is missing:

For this example the LPegLabel parser builds an AST with a dummy
expression node. By its turn, as ANTLR first tries token deletion, its parser discards token `do` and matches `print(i)` as the missing expression, then ANTLR parser reports a second error when it sees end, since it was expecting the `do` keyword that should follow the second expression.

The default ANTLR recovery strategy also did not produce a good AST for most of the tests related to an invalid statement. The ANTLR Lua grammar defines a block as `stat* reststat?`, i.e., zero or more statements followed by an optional return statement. Let us consider the invalid Lua program below:

```
local i = 0
print(i)
```

According to Lua’s syntax, the first line should be `local i = 0`. The ANTLR parser correctly reports an error when it sees the ;, but discards the remaining of the input when building the AST, it does not match any valid statements after the error.

The ANTLR parsers builds a richer AST when the previous error happens inside the block of another statement, such as below:

```
if i > 2 then
    local j = 1
end
```

In this case, the AST has information related to the statement `local i = 0` but the block related to the `do` statement ends when the parser sees the ;, what causes a spurious error when trying to match `end`.

In Table 2 we saw that ANTLR parser reports a number of error messages similar to the L Peg Label parsers that use a `FOLLOW` set to synchronize with the input. This is according to ANTLR default recovery strategy, which uses a kind of enriched local expressions. The second error message is spurious, a side effect of the parser skipping `then` and using `print("that")` as the condition.

Our PEG-based parser reports a single error with the error message "syntax error, expected a condition after ‘if’" at column 4, which seems more helpful to the programmer, and correctly parses the rest of the `if` statement.

We also compared the performance of the Lua parser generated by ANTLR with the performance of our PEG-based parser. We used the following tools in our comparison:

- ANTLR 4.6 and 4.7, with Java OpenJDK 9
- LPegLabel 1.4, with Lua 5.3 interpreter

The test machine was an Intel i7-4790 CPU with 16G RAM, running Ubuntu 16.04 LTS desktop.

We made two tests. In the first test, we created an invalid Lua program `broke.lua` that was formed by concatenating almost all the 180 erroneous programs that we have used before. This file has around 550 lines, and both parsers report more than 200 syntax errors while parsing it. This file was used to measure the performance of parsers in a syntactic invalid program. The Lua parser generated by ANTLR 4.7 crashed when parsing this file, so for this comparison we used a Lua parser generated by ANTLR 4.6.

We also used both parsers to parse the test files from the Lua 5.3.4 distribution⁶. The test comprises 28 syntactic valid Lua programs, which together have more than 12k source lines. We needed to change the first line of test file `main.lua`, because the Lua parser generated by ANTLR could not recognize it.

We ran both parsers 20 times and collected the time reported by `System.nanoTime` for ANTLR and by `os.time` for Lua. For ANTLR, we measured the time by using `@init` and `@after` actions in the start rule of the grammar. In the case of LPegLabel, we measured the time before and after calling the main function of the parser. Tables 3 and 4 show our results. We can see that the PEG-based parser was significantly faster (by approximately a factor of six) than the ANTLR parser in both tests.

6. Related work

In this section, we discuss some error reporting and recovery approaches described in the literature or implemented by parser generators. Grune and Jacobs [11] also present an overview of some error handling approaches.

Swierstra and Duponcheel [30] show an implementation of parser combinators for error recovery, but is restricted to LL(1) grammars. The recovery strategy is based on a noskip set, computed by taking the `FIRST` set of every symbol in the tails of the pending rules in the parser stack. Associated with each token in this set is a sequence of symbols (including non-terminals) that would have to be inserted to reach that point in the parse, taken from the tails of the pending rules. Tokens are then skipped until reaching a token in this set, and the parser then takes actions as if it found the sequence of inserted symbols for this token.

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. But while their strategy is more resistant to the introduction of spurious errors than just using the `FOLLOW` set it still can introduce those.

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 [31]. 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 [29] was a major influence for several tools, such as ANTLR. In Wirth’s approach, when there is an error during the

---

⁶https://www.lua.org/tests/lua-5.3.4-tests.tar.gz
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 [30] 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 [32] 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 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 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.

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 [33] and also by users of LPEG. These fallback expressions with semantic actions for error logging are similar to our labels and recovery expressions, but in an ad-hoc, implementation-specific way.

Several PEG implementations such as Parboiled10, Tatsu3, and PEGTL10 provide features that facilitate error recovery.

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 Parboiled11 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 allows to define for each rule R a set of terminator tokens T, so when the matching of R fails, the input is consumed until a token t ∈ T is matched. This is also similar to our approach for recovery expressions, but with coarser granularity, and lesser control on what can be done after an error.

Rüfenacht [3] 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, Rüfenacht noticed long running test cases and mentions the need to improve memory use and other performance issues.

7. Conclusions

We have presented a conservative extension of PEGs that is well-suited for implementing parsers with a robust mechanism for recovering from syntax errors in the input. Our extension is based on the use of labels to signal syntax errors, and differentiates them from regular failures, together with the use of recovery expressions associated with those labels.

When signaling an error with a label that has an associated recovery expression, the parser logs the label and the error position, then proceeds with the recovery expression. This recovery expression is a regular parsing expression, with access to all the parsing rules that the grammar provides. We showed how to use the information provided by FIRST and FOLLOW sets when building a recovery expression.

We also gave an extension to the formal semantics of a parsing machine for labeled PEGs that adds both furthest failure tracking and error recovery to the parsing machine. Parsing-machine based semantics form the core of several efficient PEG implementations. We proved that the extended parsing machine is correct with regards to our extended definition of PEGs.

We extended one of these virtual machine-based implementations with our error recovery mechanism, and tested several parsers for the Lua programming language on a suite of 180 programs with syntax errors to assess how close the syntax trees produced by the parsers with error recovery are to the trees we get from manually fixing the syntax errors present in the programs. We concluded that error recovery gives at least good results for 91% of our test programs, and excellent results for 56% of them.

We also compared one of our parsers with a Lua parser with

<table>
<thead>
<tr>
<th>Table 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (in ms) to parse broke.lua.</td>
</tr>
<tr>
<td>LPEGLabel</td>
</tr>
<tr>
<td>ANTLR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (in ms) to parse Lua 5.3 tests.</td>
</tr>
<tr>
<td>LPEGLabel</td>
</tr>
<tr>
<td>ANTLR</td>
</tr>
</tbody>
</table>

---


8 https://github.com/sirthias/parboiled/wiki
9 https://taocpp.com/PEGTL
10 https://github.com/sirthias/parboiled2/
automatic error recovery generated by ANTLR, a popular parser generator tool. The comparison shows that our PEG-based parser has better error recovery, error messages, and better performance than the ANTLR-generated one.

Labeled PEGs with recovery expressions give the grammar writer great control over the error recovery strategy, at the cost of an annotation burden that we judge to not be too onerous. This annotation burden can be greatly reduced when the grammar has mostly LL(1) choices. We are currently working on an algorithm that automatically annotates a PEG with labels.\[34\]

Finally, our evaluation did not try to take into account errors that are more frequent while writing Lua programs from scratch in the context of an IDE or text editor. Such a study can also be explored in future work.

References


