A Semi-Automatic Approach for Syntax Error Reporting and Recovery in Parsing Expression Grammars

Sérgio Queiroz de Medeiros

*School of Science and Technology – UFRN – Natal – Brazil*

Gilney de Azevedo Alvez Junior

*Institute Digital Metropolis – UFRN – Natal – Brazil*

Fabio Mascarenhas

*Department of Computer Science – UFRJ – Rio de Janeiro – Brazil*

**Abstract**

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 algorithms that automatically annotates a PEG with labels, and build their corresponding recovery expressions. We evaluate these algorithms by using them to generate error recovering parsers.

*Email addresses: sergiomedeiros@ect.ufrn.br (Sérgio Queiroz de Medeiros), gilneyjr@gmail.com (Gilney de Azevedo Alvez Junior), mascarenhas@ufrj.br (Fabio Mascarenhas)*
for four programming languages: Titan, Pascal, C and Java. The results show that with a small amount of manual intervention our approach can be used to produce error recovering parsers for PEGs where most choices have alternatives with disjoint \textit{FIRST} sets.

\textit{Keywords:} parsing expression grammars, labeled failures, error reporting, error recovery

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 order to conduct further analyses necessary for IDE features such as automated refactoring and code completion.

Parsing Expression Grammars (PEGs) \cite{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 \textit{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 \textit{ordered choice} operator.

The \textit{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 \cite{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 \texttt{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 \texttt{Standard} to build error recovering parsers for Pascal, C and Java.

As pointed out in \[6\], Algorithm \texttt{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 \texttt{Conservative} which does not annotate the right-hand side of a non-terminal \(A\) in case \(A\) appears either in a non-\(LL(1)\) choice or in a non-\(LL(1)\) repetition. This restriction may avoid the insertion of incorrect labels at the cost of building an AST with less information.

Overall, our experiments show that given a PEG where most choices have alternatives with disjoint \texttt{FIRST}, these algorithms, plus a small amount of manual intervention, can be used to produce an error recovering parser.

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 \texttt{Standard} which automatically annotates a PEG with labels and associates a recovery expression to each label; Section 4 evaluates the use of Algorithm \texttt{Standard} to annotate the grammars of four programming languages.
Figure 1: A PEG for a tiny subset of Java

(Titan, C, Pascal, Java); Section 5 presents Algorithm Conservative which inserts labels in a more restrictive way, and Section 6 evaluates its use 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 a short 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 more detailed presentation of labeled PEGs, which includes its formal semantics, can be found in our previous work [4, 5].

A labeled PEG $G$ is a tuple $(V, T, P, L, R, \text{fail}, p_S)$, 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, 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 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, when applied to an input string, either produces a label, associated with an input position, or consumes a prefix of the input and returns the remaining suffix. If the expression produces fail we say that it failed. The abstract syntax of parsing expressions is given as follows, where $a$ is a terminal, $A$ is a non-terminal, $p$, $p_1$ and $p_2$ are parsing expressions, and $l$ is a failure label:

$$p = \varepsilon | a | A | p_1 p_2 | p_1/p_2 | p^* | !p | \uparrow l$$

Informally, $\varepsilon$ successfully matches while not consuming any input; $a$ matches and consumes itself or fails otherwise; $A$ tries to match the expression $P(A)$; $p_1 p_2$ tries to match $p_1$ followed by $p_2$; $p_1/p_2$ tries to match $p_1$; if $p_1$ fails, i.e., the result of matching $p_1$ is fail, we try to match $p_2$; $p^*$ repeatedly matches $p$ until $p$ fails, that is, it consumes as much as it can from the input; $!p$ succeeds if the input does not match $p$ producing any label, and fails when the input matches $p$, not consuming any input in either case; we call it the negative predicate or the lookahead predicate; $\uparrow l$, where $l \in L$, generates a failure with label $l$, and in case $l$ has an associated recovery expression it will be used to match the input from the point where $l$ was thrown.

A label $l \neq \text{fail}$ thrown by $\uparrow$ cannot be caught by an ordered choice or a repetition, so it indicates an actual error during parsing, while fail 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. The rationale is that errors inside a syntactic predicate are expected and not actually syntactic errors in the input.

Figure 1 shows 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
Figure 2: A PEG with labels for a tiny subset of Java

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.

We need to annotate our original PEG with labels, which indicate the points where we can signal a syntactical error. Figure 2 annotates the PEG of Figure 1 (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
public class Example {
    public static void main(String[] args) {
        int n = 5;
        int f = 1;
        while(0 < n {
            f = f * n;
            n = n - 1
        }
        System.out.println(f);
    }
}

Figure 3: A Java program with syntax errors

(terminal or non-terminal) in the right-hand side of a production that should not fail, as failure would just make the whole parser either fail or not consume the whole input. 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 `rparwhile` is thrown when we fail to match a `')'`, so we could attach an error message like “missing `')' in while” to this label. Dynamically, when the matching of `')' 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 Figure 3 which has two syntax errors: a missing `')' at line 5, and a missing semicolon at the end of line 7. For this program, a parser based on the labeled PEG from Figure 2 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 detailed discussion about `FIRST` and `FOLLOW` sets in the context of PEGs can be found in other papers [8, 9, 10].

With the help of these sets, we can define the following recovery expression for `rparwhile`, where . is a parsing expression that matches any character:

$$(!\text{FIRST}(\text{stmt}).)^*)$$

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. In a concrete implementation, instead of . we should use a parsing expression that consumes a whole token.

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 automated for some classes of parsing expression grammars.

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 always implies that the whole parse will fail or not consume the whole input, 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 Figure [1]. 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.

Algorithm [Standard] automatically adds labels and recovery expressions to a PEG $G = (V, T, P, L, R, \text{fail}, p_S)$. We assume that all occurrences of $FIRST$ and $FOLLOW$ in Algorithm [Conservative] give 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 7–28) to annotate the right-hand side, a
Algorithm Standard Adding Labels and Recovery Expressions to a PEG

1: function annotate(G)
2:   \( G' \leftarrow G \)
3: for \( A \in G \) do
4:   \( G'(A) \leftarrow \text{labexp}(G(A), \text{false}, \text{FOLLOW}(A)) \)
5: return \( G' \)

7: function labexp(p, seq, flw)
8: if \( p = a \) and \( seq \) then
9:   return addlab(p, flw)
10: else if \( p = A \) and \( \epsilon \not\in \text{FIRST}(A) \) and \( seq \) then
11:   return addlab(p, flw)
12: else if \( p = p_1 \cdot p_2 \) then
13:   \( p_x \leftarrow \text{labexp}(p_1, seq, \text{calck}(p_2, flw)) \)
14:   \( p_y \leftarrow \text{labexp}(p_2, seq \ or \ \epsilon \not\in \text{FIRST}(p_1), flw) \)
15: return \( p_x \cdot p_y \)
16: else if \( p = p_1 / p_2 \) then
17:   \( p_x \leftarrow p_1 \)
18: if \( \text{FIRST}(p_1) \cap \text{calck}(p_2, flw) = \emptyset \) then
19:   \( p_x \leftarrow \text{labexp}(p_1, \text{false}, flw) \)
20: \( p_y \leftarrow \text{labexp}(p_2, \text{false}, flw) \)
21: if \( seq \ and \ \epsilon \not\in \text{FIRST}(p_1 / p_2) \) then
22:   return addlab(p_x / p_y, flw)
23: else
24:   return \( p_x / p_y \)
25: else if \( p = p_1* \) and \( \text{FIRST}(p_1) \cap flw = \emptyset \) then
26:   return \( \text{labexp}(p_1, \text{false}, flw)* \)
27: else
28:   return \( p \)
30: function calck(p, flw)
31: if \( \epsilon \in \text{FIRST}(p) \) then
32:   return \( (\text{FIRST}(p) - \{ \epsilon \}) \cup flw \)
33: else
34:   return \( \text{FIRST}(p) \)
35: function addlab(p, flw)
36: \( l \leftarrow \text{newLabel}() \)
37: \( R'(l) \leftarrow \text{(!flw .)}* \)
38: return \([p] \)
parsing expression, of each rule of grammar $G$. The auxiliary function $\text{calc}$ (lines 30–34) is used to update the $\text{FOLLOW}$ set associated with a parsing expression. By its turn, the auxiliary function $\text{addlab}$ (lines 35–38) receives a parsing expression $p$ to annotate and its associated $\text{FOLLOW}$ set $flw$. Function $\text{addlab}$ associates a label $l$ to $p$ and also builds a recovery expression for $l$ based on $flw$.

Algorithm $\text{Standard}$ 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 our algorithm.

Function $\text{labexp}$ has three parameters. The first one, $p$, is a parsing expression that we will try to annotate. The second parameter, $seq$, indicates whether we have already matched a prefix of a concatenation expression that consumes at least one terminal or not. Parameter $seq$ has value $\text{true}$ when $p$ is a suffix of a concatenation $p_1 p_2$ and the prefix of $p_1 p_2$ already consumed at least one input character. Finally, the parameter $flw$ represents the $\text{FOLLOW}$ set associated with $p$. Let us now discuss how $\text{labexp}$ tries to annotate $p$.

When $p$ is an expression that matches a terminal and is part of a concatenation that already matched at least one terminal (lines 8–9), then we associate a new label with $p$. In case $p$ represents a terminal but $seq$ is not $\text{true}$, we will just return $p$ (lines 27–28).

The case when $p$ matches a non-terminal $A$ is similar (lines 10–11), we have just added an extra condition that tests 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 12–15), we try to annotate $p_1$ and $p_2$ recursively. To annotate $p_1$ we use an updated $\text{FOLLOW}$ set, and to annotate $p_2$ we set its parameter $seq$ to $\text{true}$ whenever $seq$ is already $\text{true}$ or $p_1$ does not match the empty string.
In case of a choice \( p_1 / p_2 \) (lines 16–24), 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 25–26), we can annotate \( p_1 \) 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 Figure 1, function `annotate` would give us the grammar presented in Figure 2 (as previously, we are not taking rule `prog` into consideration), with the exception of the annotation `[stmt]elstmt`. Label `elstmt` 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 Figure 2 we associated the label `elstmt` 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 L Peg Label, a tool that implements the semantics of PEGs with labeled failures, and pegparser, 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 get 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 inserted a label as the manual annotation did.

Missing When the algorithm did not insert a label.

ExtraOk When the algorithm correctly inserted a new label.

ExtraWrong When the algorithm incorrectly inserted a new label.
Table 1: Evaluation of the Labels Inserted by Algorithm Standard

| Grammar | Equal | Missing | ExtraOk | ExtraWrong |
|---------|-------|---------|---------|------------|
| Titan   | 76 + 2 (≈ 90% of manual) | 9 (≈ 10% of manual) | 2 | 0 |
| C       | 65 (≈ 75% of manual) | 22 (≈ 25% of manual) | 9 | 1 |
| Pascal  | 100 (≈ 98% of manual) | 2 (≈ 2% of manual) | 1 | 3 |
| Java    | 140 (≈ 80% of manual) | 35 (≈ 20% of manual) | 10 | 31 |

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 Item Missing a serious flaw, 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 ExtraWrong 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.

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 Algo-
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 matching 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.

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

| Parser | Excellent | Good | Poor | Awful |
|--------|-----------|------|------|-------|
| Titan  | 61 ($\approx 81\%$) | 2 ($\approx 3\%$) | 11 ($\approx 15\%$) | 1 ($\approx 1\%$) |
| C      | 40 ($\approx 67\%$) | 3 ($\approx 5\%$) | 16 ($\approx 27\%$) | 1 ($\approx 2\%$) |
| Pascal | 79 ($\approx 78\%$) | 13 ($\approx 13\%$) | 9 ($\approx 9\%$) | 0 (0%) |
| Java   | 111 ($\approx 63\%$) | 24 ($\approx 14\%$) | 40 ($\approx 23\%$) | 0 (0%) |

Table 2: Evaluation of Automatic Error Recovery Based on Algorithm Standard.
given at line 2:

```plaintext
1  sum = 0
2  for i = 1, 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. As the Titan

---

[1] http://bit.ly/titan-reference
[2] http://bit.ly/titan-unlabeled
developers intend to keep using a PEG-based parser for the Titan compiler, the 
Titan grammar seemed a good candidate to evaluate our algorithm.

The manually annotated Titan grammar we got from our unlabeled gram-
mar 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 87 expressions that throw lab els. 
Some labels, such as EndFunc, are thrown more than once, i.e., they are asso-
ciated with more than one expression. From the 50 grammar syntactical rules, 
36 of them may throw some label.

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

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 87 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 I shows. We could insert automatically 90% of the labels 
inserted manually. Below we discuss the main issues related to the generated 
Titan grammar.

About Item Missing, as expected our approach did not annotate parts of the 
grammar where the alternatives of a choice were not disjoint. This happened 
in 4 of the 50 grammar rules. One of these rules was castexp, which we show 
below:

\[
\text{castexp} \leftarrow \text{simpleexp} \text{ `as` type } / \text{ simpleexp}
\]

http://bit.ly/titan-manual
http://bit.ly/titan-standard
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` (`as` `type` / `ε`), or using the short form `simpleexp` (`as` `type`)?, would give enough context for Algorithm Standard to correctly annotate `type` with a label, though.

The original Titan grammar also 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:

\[ &\left\langle exp \text{ as}' \right\rangle \quad \uparrow^{\text{ExpAssign}} \]

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 did not insert new labels incorrectly, but we have faced a problem similar to Item ExtraWrong, which made the parser reject valid inputs. This was caused by the insertion of 2 labels already present in the manually annotated Titan grammar. Because of this, in Table II we put the value 76 + 2 in column Equal.

This issue happened in rules `toplevelvar` and `import`. Figure 4 shows the definition of these rules, plus some rules that help to add context, in the original Titan grammar.
program ← (toplevelfunc / toplevelvar / toplevelrecord / import / foreign)* !.
toplevelvar ← localopt decl ['m']=AssignVar !(import / foreign) [exp] ExpVarDec
import ← localopt NAME NameImport ['m']=AssignImport
    !(foreign) [import] ImportImport ('(' [STRING] StringLParImport [')']) RParImport / [STRING] StringImport
foreign ← localopt NAME NameImport ['m']=AssignImport
    'foreign' [import] ImportImport ('(' [STRING] StringLParImport [')']) RParImport / [STRING] StringImport
decl ← NAME (';:' [type] TypeDecl)?

Figure 4: Predicates !(import / foreign) and !foreign enable adding labels after them

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 !(import/foreign) was added by the Titan developers to make sure the input does not match the import or the foreign rules, 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 as the original grammar, 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 “\texttt{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 \textbf{Standard} correctly add two new labels. It annotated ‘\texttt{->}’ in the first alternative of rule \textit{type}, and ‘\texttt{foreign}’ in rule \textit{foreign}.

4.1.2. Automatic Error Recovery

The test suite of Titan has 75 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.

A first running of our parser showed that for 7 files it failed to build an AST. A brief comparison with the manually annotated grammar revealed that this was due a missing label in the start rule \textit{program}, shown in Figure \ref{fig:program}.

As rule \textit{program} is the grammar start rule, it must not fail if we want a successful matching. As our parser will only build an AST for such matchings, we should annotate the expressions of the grammar start rule which may lead to a failure. In this case, we should annotate ‘!’. and add a recovery rule that consumes the rest of the input.

We will use this same approach for the other languages, in order to always build an AST for syntactically invalid programs. It is not difficult to extend the Algorithm \textbf{Standard} with this extra case involving the start rule. After changing rule \textit{program}, we ran the test set of Titan again and as expected our parser built ASTs for all syntactically invalid programs.

We can see in Table \ref{table:results} 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 \textit{good}.

By analysing 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 (Item \textbf{Missing}). 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 \texttt{castexp}, which we have shown in Section 4.1.1. In the following input there is a missing type after the keyword \texttt{as} at line 1:

\begin{verbatim}
1    x = foo as
2    return x
\end{verbatim}

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

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

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

\section{C}

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

To write our unlabeled grammar for C\footnote{http://bit.ly/c89-unlabeled} we needed to remove left-recursion, as \texttt{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-\textit{LL}(1) choices and 5 have non-\textit{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\(^9\). From the 50 grammar syntactical rules, 30 of them may throw some label.

By its turn, the automatically annotated C grammar\(^10\) 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

Algorithm `Standard` annotated the C grammar with 75 labels, which is not far from the 87 labels of the original C grammar. As was the case for Titan, often the algorithm inserted labels at the same location of the original ones, as we can see in Table \(1\). 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`. Figure \(5\) 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 `ExtraWrong` 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

---

\(^8\)http://bit.ly/c89-manual
\(^9\)Coincidentally, both Titan and C grammars have the same number of syntactical rules, and their manually annotated grammars throw the same amount of labels.
\(^10\)http://bit.ly/c89-standard
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 labels correctly, which is more than the 2 new labels added when considered the Titan grammar. We think the algorithm added more new labels for C for two reasons: first, as the original Titan parser already used labels, more time was devoted to manually annotate the grammar; second, as the C grammar has many non-LL(1) choices, this 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 60 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 always build an AST. In the case of the C grammar, we added two labels to the grammar start rule.

In Table 2 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 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 (Item Missing).

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.

As an example, let us see the case of a missing label related to an if-else statement. Figure 6 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:

```c
1 int fat (int x) {
2   if (x == 0)
3     return 1;
4 else
5   }
```

Figure 6: Manually Annotated if-else Statement

```
stat ← 'if' ['(']BrackIf [exp]InvalidExpr ['(']Brack [stat]Stat 'else' [stat]Stat
/ 'if' ['(']BrackIf [exp]InvalidExpr ['(']Brack [stat]Stat
```
The generated C parser will try to recognize the first alternative of the choice in rule \textit{stat}. It will fail to recognize \textit{stat} after \texttt{“else”}, which will produce a regular failure. Thus, the parser backtracks, recognize an \textit{if}-statement without an \textit{else}-part, and then will fail to recognize another statement as we left \texttt{“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 \textit{if}-statement and we would get a better recovery in this case.

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 has 67 syntactical rules. Among these rules, 4 of them have non-\textit{LL}(1) choices, and 6 of them have non-\textit{LL}(1) repetitions.

The manually annotated Pascal grammar[12] has 102 expressions that throw labels. From the 67 grammar syntactical rules, 48 of them may throw some label.

By using Algorithm Standard from the unlabeled Pascal grammar we got a generated grammar[13] 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[1] shows, Algorithm Standard annotated the Pascal grammar with 104 labels, in a way nearly identical to manual annotation, it inserted 98% of the labels inserted manually. We think the low number of non-\textit{LL}(1) choices and non-\textit{LL}(1) repetitions helped the algorithm to achieve this performance.

\footnotesize
1. http://bit.ly/pascal-unlabeled
2. http://bit.ly/pascal-manual
3. http://bit.ly/pascal-standard
simpleStmt ← assignStmt / procStmt / gotoStmt

assignStmt ← var ['='][AssignErr] [expr][ExprErr]

\[
\text{var} \leftarrow \text{Id} \left( [\text{expr}]^\text{ExprErr} \left( \text{Comma} [\text{expr}]^\text{ExprErr} \right)^* \right)^\text{RBrackErr} / \text{Dot} [\text{Id}]^\text{IdErr} / \text{Dot}^* \\
\]

procStmt ← Id params?

Figure 7: Label AssignErr Incorrectly Added in Rule assignStmt

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 Figure 7.

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 that, in rule assignStmt, which appears in the first alternative of this choice, we should not annotate ‘=’, otherwise the parser will not recognize a valid procStmt such as “f(x)”, as ‘=’ does not follow the identifier “f”.

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 2 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 C error recovering parses, in case of the Pascal parser we can not associate the poor ASTs with
the absence of labels (Item Missing). 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 in the Mouse site\footnote{http://www.romanredz.se/Mouse/Java.1.8.peg}.

Our unlabeled Java grammar\footnote{http://bit.ly/java8-unlabeled} 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\footnote{http://bit.ly/java8-manual} has 175 expressions that throw labels. More than half of the syntactical rules, 77 out of 147, may throw some label.

From the unlabeled Java grammar, we used Algorithm Standard to get a generated grammar\footnote{http://bit.ly/java8-standard} 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\footnote{http://bit.ly/java8-standard} that Algorithm Standard annotated the Java grammar with 181 labels, from which 140 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 this, an important part of the labels added (17%) 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 2 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 (Item **Missing**). We could get a better result in these cases by rewriting non-$LL(1)$ 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.

Algorithm Conservative tries to address this problem related to Item ExtraWrong by not annotating the right-hand side of a non-terminal A when it appears either in a non-LL(1) choice or in a non-LL(1) repetition. This algorithm annotates a grammar more cautiously, although it may still insert labels incorrectly. Algorithm Conservative essentially modifies function annotate from Algorithm Standard as we discuss below. Functions labexp, calck and addlab remain the same, they were omitted.

Now, in function annotate (lines 1–9), before annotating the right-hand side of a non-terminal A, we check whether A was banned or not. In case A have not been banned (lines 5–6), we try to annotate its right-hand side. Otherwise (lines 7–8), we do not.

Function ban (lines 11–15) builds a set banned with all the non-terminals that appear either in a non-LL(1) choice or in a non-LL(1) repetition. To build such set, it uses function notlabel.

Function notlabel (lines 17–29) receives an expression p, a flag that indicates whether p is a subexpression of either a non-LL(1) choice or a non-LL(1) repetition, and the FOLLOW set of p. As a result, function notlabel gives a set with the non-terminals whose right-hand sides we should not annotate.

When p is a non-terminal A and flag holds a true value (lines 18–19), then we create a set with A indicating we should not annotate its right-hand side. In case p is a concatenation $p_1p_2$ (lines 20–21), we return the union of the sets obtained by calling notlabel recursively for $p_1$ and $p_2$.

In function notlabel, the value of flag can change from false to true in case either of a non-LL(1) choice (line 23) or of a non-LL(1) repetition (line 26). In case an expression p has none of these non-LL(1) expressions, function notlabel will return an empty set.
Algorithm Conservative

Annotating Only the Right-Hand Side of Non-Terminals Neither Used in a Non-LL(1) Choice Nor in a Non-LL(1) Repetition

1: function annotate(G)
2:   banned ← ban(G)
3:   G′ ← G
4: for A ∈ G do
5:   if A /∈ banned then
6:       G′(A) ← labexp(G(A), false, FOLLOW(A))
7:   else
8:       G′(A) ← G(A)
9: return G′
10:
11: function ban(G)
12:   banned ← ∅
13: for A ∈ G do
14:       banned ← banned ∪ notlabel(G(A), false, FOLLOW(A))
15: return banned
16:
17: function notlabel(p, flag, flw)
18:   if p = A and flag then
19:       return {A}
20:   else if p = p₁ p₂ then
21:       return notlabel(p₁, flag, calck(p₂, flw)) ∪ notlabel(p₂, flag, flw)
22:   else if p = p₁ / p₂ then
23:       flag ← flag or FIRST(p₁) ∩ calck(p₂, flw) ≠ ∅
24:       return notlabel(p₁, flag, flw) ∪ notlabel(p₂, flag, flw)
25:   else if p = p₁* or p = p₁+ or p = p₂? then
26:       flag ← flag or FIRST(p₁) ∩ flw ≠ ∅
27:       return notlabel(p₁, flag, flw)
28:   else
29:       return ∅
Overall, Algorithm Conservative will insert a subset of the labels inserted by Algorithm Standard. In case a non-terminal \( A \) is not banned, Algorithm Conservative will annotate \( G(A) \) in the same way as Algorithm Standard.

Given the PEG from Figure 1, which has only the non-LL(1) choice ‘else stmt / \( \varepsilon \)’, the non-terminal stmt, which appears inside the choice, will be banned. Thus, the result of function ban will be a set whose only member is stmt, so function annotate will not add labels to the right-hand side of stmt.

For the grammar from Figure 1, Algorithm Conservative will give the same result of Algorithm Standard since that Algorithm Standard also did not add labels to the right-hand side of stmt.

Next section evaluates the use of Algorithm Conservative to annotate the Titan, C, Pascal and Java grammars.

### 6. Evaluating Algorithm Conservative

To evaluate Algorithm Conservative we will use the same approach we used in Section 3 to evaluate Algorithm Standard.

Table 3 shows a comparison between the labels inserted automatically and the labels inserted manually for the Titan, C, Pascal and Java grammars. When we analyze also the results from Table 1, we can see that, with the exception of the Pascal grammar, the Algorithm Conservative inserted considerably less labels than Algorithm Standard. On the other hand, it also diminished the number of labels inserted incorrectly, specially for the Java grammar.

By its turn, Table 4, which we should compare with Table 2, evaluates...
Table 4: Evaluation of Automatic Error Recovery Based on Algorithm Conservative

| Parser  | Excellent | Good | Poor | Awful |
|---------|-----------|------|------|-------|
| Titan   | 33 (≈ 44%)| 3 (≈ 4%) | 0 (0%) | 39 (≈ 52%) |
| C       | 12 (≈ 20%)| 1 (≈ 2%) | 0 (0%) | 47 (≈ 78%) |
| Pascal  | 79 (≈ 78%)| 13 (≈ 13%) | 9 (≈ 9%) | 0 (0%) |
| Java    | 41 (≈ 23%)| 8 (≈ 5%)  | 107 (≈ 61%) | 19 (≈ 11%) |

the ASTs built by the different recovering parsers obtained through Algorithm Conservative. For all parses, the quality of the error recovery diminished, as the grammar has less labels and recovery rules.

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

6.1. Titan

As we have mentioned in Section 4.1, the Titan grammar has 7 rules with non-LL(1) choices (program, rettype, type, statement, castexp, simpleexp and var) and 3 rules with non-LL(1) repetitions (suffixedexp, fieldlist and field). By using Algorithm Conservative, we will ban the non-terminals that are used in the right-hand side of these rules.

This leads to a set with 17 non-terminals (from the 50 non-terminals of Titan grammar) whose right-hand side we will not try to annotate. This, of course, results that the generated grammar we got through Algorithm Conservative has less labels than that we got by using Algorithm Standard.

The Algorithm Standard added 76 labels correctly, and 2 other ones incorrectly. By its turn, Algorithm Conservative added 43 labels correctly and none incorrectly.

Let us revisit the case presented in Figure 4 where Algorithm Standard incorrectly added the label AssignImport in rule toplevelvar and the label...
Import in rule import.

As there is a non-LL(1) choice in rule program, Algorithm Conservative will ban the non-terminals used in this choice. Thus, it will not add any labels to the right-hand side of rules import and toplevelvar.

When there is non-LL(1) choice, Algorithm Conservative bans all the non-terminals that appear after the conflict is detected, so after detecting the non-LL(1) choice in rule program the non-terminal toplevelrecord will be banned too, although there is no conflict related to it.

We could circumvent this by making toplevelrecord the first alternative of the choice. With this change, Algorithm Conservative will not ban this non-terminal, therefore we can try to annotate its right-hand side.

Since that the grammar generated by Algorithm Conservative has less labels, it also has less recovery rules. Because of this, the corresponding parser gives an acceptable recovery for around 50% of our test programs, while the parser generated by Algorithm Standard built acceptable ASTs for more than 80% of these programs.

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. The use of Algorithm Conservative will ban 29 rules.

The banning of these rules gives us a generated grammar with only 23 labels (around one third of the labels correctly inserted by Algorithm Standard), although no label was inserted incorrectly.

In case of our C grammar, we can notice that a great amount of the labels inserted by Algorithm Standard, 33 to be exact, were in the right-hand side of rule stat. However, this non-terminal was banned by Algorithm Conservative so its right-hand side was not annotated in this case.

\[http://bit.ly/c89-conservative\]
One reason that lead to the banning of \textit{stat} was related to the \textit{if-else} statement we have shown in Figure \ref{fig:example}. As there is a non-LL(1) choice, we will ban the non-terminals used in the choice, where one of these non-terminals is \textit{stat}.

In order to not ban \textit{stat}, we could rewrite the fragment of \textit{stat} that appears in Figure \ref{fig:example} as:

\[
\text{\textit{stat} ← 'if' '(' \textit{exp} ')' 'else' \textit{stat} / \varepsilon}
\]

Unfortunately, this does not solve the issue because, as the grammar from Figure \ref{fig:example} our C grammar also has the dangling \textit{else} problem. Given that the choice \textit{'else' stat / \varepsilon} is not LL(1), we would still ban \textit{stat}.

To fix this issue, we would need to use a new rule \textit{statElse} whose right-hand side is just \textit{stat}. The resulting \textit{stat} rule would be:

\[
\text{\textit{stat} ← 'if' '(' \textit{exp} ')' 'else' \textit{statElse} / \varepsilon}
\]

Now, we will ban \textit{statElse}, but not the \textit{stat} non-terminal. By being able to annotate rule \textit{stat}, we would increase the amount of labels inserted by Algorithm \textbf{Conservative} remarkably.

We can see in Table \ref{tab:results} that the C grammar was the one for which Algorithm \textbf{Conservative} inserted less labels in comparison with the amount of labels inserted manually. Accordingly, as we can see in Table \ref{tab:results} the corresponding C error recovering parser presented the worst results when compared to the parsers of the other languages.

6.3. Pascal

As mentioned in Section 4.3 our unlabeled Pascal grammar has few rules, among its 67 syntactical rules, with an LL(1) conflict: 4 of them have a non-LL(1) choice, and 6 of them have a non-LL(1) repetition.

Algorithm \textbf{Conservative} banned 15 rules of Pascal grammar and generated a grammar \cite{http://bit.ly/pascal-conservative} with 84 labels, where one of these labels was inserted incorrectly.
ordinalType ← newOrdinalType / Id
newOrdinalType ← enumType / subrangeType
enumType ← ‘(\[ids\]|IdErr)[\]IdErr
subrangeType ← const [..]DotDotErr [const]ConstErr
const ← Sign? (UNumber / Id) / String

Figure 8: Label DotDotErr Incorrectly Added in Rule subrangeType

To compare, Algorithm [Standard] inserted 104 labels, 3 of them incorrectly.

Algorithm [Conservative] did not insert the label AssignErr, which we have discussed in Figure [7] because non-terminal assignStmt appears in a non-LL(1) choice in rule simpleStmt. Thus, non-terminal assignStmt was banned and the right-hand side of its rule was not annotated.

In case of Pascal, the only label incorrectly added by Algorithm [Conservative] was in rule subrangeType, whose definition, plus the definition of other rules that help to add context, we can see in Figure [8].

In rule ordinalType there is a non-LL(1) choice, since that Id belongs to the FIRST set of both alternatives of the choice. Because of this, Algorithm [Conservative] will ban the non-terminal newOrdinalType, so we will not annotate its right-hand side.

In rule newOrdinalType there is no LL(1) conflict, therefore we will not ban other non-terminals.

However, in rule subrangeType we can not throw label DotDotErr. When trying to match an ordinalType, the parser could recognize an Id as the beginning of a subrangeType, then fail to recognize a ‘..’, backtrack and finally match the second alternative of the choice newOrdinalType / Id.

In this example, Algorithm [Conservative] did not ban subrangeType because it was not used in any non-LL(1) choice. To fix this issue, in rule newOrdinalType,
we would need to replace `subrangeType` with its right-hand side, as below:

\[
\text{newOrdinalType} \leftarrow \text{enumType} / \text{const} \ [\ldots] ^{\text{DotDotErr}} \ [\text{const}] ^{\text{ConstErr}}
\]

As Algorithm `Conservative` bans `newOrdinalType`, the labels `DotDotErr` and `ConstErr` will not be added to the previous rule.

In Table 4, we can see that the error recovering parser generated by Algorithm `Conservative` which inserted 18 correct labels less than Algorithm `Standard` built an acceptable AST for 80% of the syntactically invalid programs in our test. A manual inspection revealed that for 18 incorrect programs this parser built an AST with less information than the parser generated by Algorithm `Standard`, which shows that we got a poorer recovery due to the missing labels.

### 6.4. Java

In case of our unlabeled Java grammar, where there is an `LL(1)` conflict in one third of its 147 rules, Algorithm `Conservative` banned 77 non-terminals. In Table 3 we can see this algorithm generated a grammar with 59 labels, where five of them (≈ 9%) were inserted incorrectly. To compare, Algorithm `Standard` inserted 181 labels, 31 (≈ 17%) of them incorrectly.

As the unlabeled Java grammar has many `LL(1)` conflicts, Algorithm `Conservative` could not annotate more than half of the grammar syntactical rules, which diminished considerably the amount of labels inserted. On the other hand, the use of this algorithm reduced significantly the number of labels inserted incorrectly.

As was the case in our C grammar (Section 6.2), the Algorithm `Conservative` did not annotate the rule `statement` of Java grammar, where we also have the `dangling else` problem. By rewriting this rule as we did for C would enable Algorithm `Conservative` to add more than 20 labels.

The cases where a label was inserted incorrectly are similar to the case we have discussed to Pascal: a non-terminal `A` is used in the first alternative of a
non-LL(1) choice, so Algorithm \texttt{Conservative} bans \texttt{A}; in the right-hand side of rule \texttt{A} there is no LL(1) conflict and we try to match a non-terminal \texttt{B}; as we did not ban \texttt{B}, it may incorrectly throw labels.

From Table \textbf{4} we can see that the error recovering parser generated by Algorithm \texttt{Conservative} did not perform well. This result was somehow expected, since that the algorithm failed to add many labels that were inserted during the manual annotation.

7. Related Work

In this section, we discuss some error reporting and recovery approaches described in the literature or implemented by parser generators.

Swierstra and Duponcheel \cite{Swierstra2014} show an implementation of parser combiners for error recovery, but it is restricted to LL(1) grammars. The recovery strategy is based on a \texttt{noskip} set, computed by taking the \texttt{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 have 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 \texttt{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 \cite{Liu1995}. 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 [16] 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 [19] 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 through 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 \textit{SYNC} and \textit{WEAK} keywords of Coco/R. Each use of \textit{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 \textit{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 \cite{20, 21} 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 \cite{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 “\textit{until it finds a token that could reasonably follow the current non-terminal}” \cite{22}. 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 \cite{23} and also by users of LPeg \cite{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\textsuperscript{23}, Tatsu\textsuperscript{24}, and PEGTL\textsuperscript{25} 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 Parboiled\textsuperscript{26} does not have an error recovery mechanism.

Tatsu uses the fallback alternative technique for error recovery, with the addition of a \textit{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 $t \in T$ 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.

R"ufenacht\textsuperscript{24} 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 R"ufenacht 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, R"ufenacht noticed long running test cases

\footnotesize{\begin{itemize}
\item \textsuperscript{1}https://github.com/sirthias/parboiled/wiki
\item \textsuperscript{2}https://tatsu.readthedocs.io
\item \textsuperscript{3}https://github.com/taocpp/PEGTL
\item \textsuperscript{4}https://github.com/sirthias/parboiled2/
\end{itemize}}
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 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, although it is arguably subjective.

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 have presented a mechanism for partially automating the process of adding error reporting and error recovery to parsers based on Parsing Expression Grammars. To achieve this, we proposed algorithms that automatically annotate the $LL(1)$ parts of a PEG with error labels and associates recovery expressions for these labels.

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

The major limitation of this algorithm 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.

Algorithm tried to address this issue, although it did not solve it completely. By using it, we can add only a subset of the labels inserted by Algorithm. In our evaluation, the size of this subset varied from 79%
(Pascal) to 31% (C and Java) of the original set size.

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

By using these algorithms to automatically insert labels, we can provide a good generic error reporting mechanism. In case generic error messages, which only indicate which term was expected and what was found in the input, are not enough, the parser developer also needs to associate specific error messages 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.

As a future work, we should investigate other variations of Algorithm \texttt{Standard} that can avoid the introduction of spurious annotations, while not decreasing the number of useful annotations.

We should also 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.

References

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, Vol. 8129 of LNCS, Springer Berlin Heidelberg, 2013, pp. 1–15. 
doi:10.1007/978-3-642-40922-6_1
URL http://dx.doi.org/10.1007/978-3-642-40922-6_1

[3] A. M. Maidl, F. Mascarenhas, S. Medeiros, R. Ierusalimschy, Error reporting in parsing expression grammars, Sci. Comput. Prog. 132 (P1) (2016) 129–140. doi:10.1016/j.scico.2016.08.004
URL 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, Journal of Visual Languages & Computing 49 (2018) 17 – 28. doi:https://doi.org/10.1016/j.jvlc.2018.10.003
URL 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. doi:10.1145/3264637.3264638
URL http://doi.acm.org/10.1145/3264637.3264638

[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, Fundamenta Informaticae 93 (2009) 325–336.

[9] R. R. Redziejowski, More about converting BNF to PEG, Fundamenta Informaticae 133 (2014) 257–270. doi:10.3233/FI-2014-1075
URL http://dx.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, Science of Computer Programming 89 (2014) 235 – 250.

doi:https://doi.org/10.1016/j.scico.2014.01.012
URL http://www.sciencedirect.com/science/article/pii/S0167642314000276

[11] T. T. Developers, The Titan programming language, http://titan-lang.org/ [Visited on May 2018] (2017).

[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).
URL https://www.iso.org/standard/13802.html

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

[17] C. L. Jeffery, Generating LR syntax error messages from examples, ACM Transactions on Programming Languages and Systems (TOPLAS) 25 (5) (2003) 631–640.

[18] N. Wirth, Algorithms + Data Structures = Programs, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1978.
[19] H. Mössenböck, The Compiler Generator Coco/R (2010).
URL http://www.ssw.uni-linz.ac.at/Coco/Doc/UserManual.pdf

[20] T. Parr, ANTLR, http://www.antlr.org [Visited on May 2018] (2014).

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

[22] 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.
doi:10.1145/2660193.2660202
URL http://doi.acm.org/10.1145/2660193.2660202

[23] R. R. Redziejowski, Mouse: from Parsing Expressions to a Practial Parser (2017).
URL http://mousepeg.sourceforge.net/Manual.pdf

[24] M. Rüfenacht, Error handling in peg parsers, Master’s thesis, University of Berne (2016).

[25] 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. doi:10.1145/512760.512786
URL http://doi.acm.org/10.1145/512760.512786

[26] 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. doi:10.1145/586088.586092
URL http://doi.acm.org/10.1145/586088.586092

[27] P. Degano, C. Priami, Comparison of syntactic error handling in lr parsers Softw. Pract. Exper. 25 (6) (1995) 657–679.
[28] 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.
doi:10.1145/2400676.2400678
URL http://doi.acm.org/10.1145/2400676.2400678