International Journal of Research in Engineering, Science and Management
Volume-2, Issue-5, May-2019
www.ijresm.com | ISSN (Online): 2581-5792

Error Reporting and Recovery in Parsing Expression Grammars

N. K. Theertha Krishnan[1], Shraddha Sahinath Rane[2], Sinni Anna Alex[3]
[1,2]*Student, Dept. of Computer Science and Engineering, Ramaiah Institute of Technology, Bangalore, India*
[3]*Assistant Professor, Dept. of Computer Science and Engg., Ramaiah Institute of Technology, Bangalore, India*

***Abstract*: In Integrated Development Environments (IDEs), error recovery is an important feature for parsers, which must build Abstract Syntax Trees (ASTs) even for syntactically incorrect programs in order to offer some features. Parsing Expression Grammars are formalisms that describe a top down parser with backtracking. In this paper we discuss some of the error recovery techniques and error reporting method that can be applied to PEGs.**

***Keywords*: Parsing Expression Grammars, Error Recovery, Error Reporting, Parsing.**

1. Introduction

Parsing Expression Grammars [16] are formalisms that are used to describe top down parsers with backtracking. But these don't provide a good error recovery mechanisms and thus are not suitable to be used in Integrated Development Environments (IDEs). Over the years few error recovery techniques and error reporting methods have been proposed and this paper provides a summary of those techniques. Error reporting techniques used for top down parsers cannot be applied to parsers which are based on PEGs, so an error reporting technique for these parsers was introduced. Labeled error recovery is a conservative extension of PEGs where an error recovery expression is mapped to each label. These expressions can use the full expressivity of PEGs to recover from syntactic errors. To avoid the burden of manually annotating a grammar with recovery expressions and labels, automatic error recovery technique was introduced where it automatically annotates a PEG with labels, and builds their corresponding recovery expressions was proposed. The rest of the paper is divided as follows: The following section (2) gives a brief history about PEGs, section (3) discusses about the error reporting technique and section (4) discusses about the two error recovery methods and then a final conclusion is given.

2. Parsing expression grammars

Since several decades we have been using context-free grammars (CFGs) and regular expressions (REs), in order to express the syntax of programming languages. The context free grammars are able to express ambiguous syntax for natural languages but this ability gradually reduces when we use context free grammars for machine oriented languages. Ambiguity in CFGs is difficult to avoid even when we want to, and it makes general CFG parsing an inherently super-linear-time problem [1,2]. Alternatively, Parsing Expression Grammars or PEGs are used. PEGs are stylistically similar to CFGs with RE-like features added, much like Extended Backus-Naur Form (EBNF) notation [3], [4]. One major difference is that instead of unordered choice operator '|' Parsing Expression Grammars use prioritized choice operators'/'which using first correct match lists alternative patterns to be tested in order unconditionally. The EBNF rules '$X \rightarrow x\ y|\ x$' and '$X\rightarrow x\ |x\ y$' are both equivalent in a CFG, but the PEG rules '$X \leftarrow x\ y/\ x$' and '$X \leftarrow x\ /\ x\ y$' are different.

A PEG can be visualized as a formal description of a top-down parser. Two closely related prior systems upon which this work is based, were developed primarily, for the purpose of studying top-down parsers [5, 6]. PEGs have far more syntactic expressiveness than the LL(k) language class typically associated with top-down parsers, however, and can express all deterministic LR(k) languages and many others, including some non-context-free languages. Despite their considerable expressive power, all PEGs can be parsed in linear time using a tabular or memorizing parser [7]. These features determine that both context free grammar and Parsing expression grammars define incomparable language classes. Also PEG can be considered as recursive descent parser with limited backtracking which means that when an input prefix is being recognized by an alternative no other of similar choice will be tried but when input prefix fails to be recognized by an alternative then parser will back track to the next alternative. Although PEGs are considered as a class of top down parsers, the error handling techniques applied to top down parsers cannot be applied to PEGs as these error handling techniques assume that the parser reads the input without backtracking. Also it is more difficult to identify the position and cause of error in PEGs. Ford [3] has already identified this limitation of error reporting in PEGs, and, in his parser generators for PEGs, included a heuristic for better error reporting. This heuristic simulates the error reporting technique that is implemented in top-down parsers without backtracking. The idea is to track the position in the input where the farthest failure occurred, as well as what the parser was expecting at that point, and report this to the user in case of errors. Tracking the farthest failure position and context gives us PEGs that produce error messages similar to the automatically produced error messages of other top-down parsers; they tell the user the position where the error was encountered, what was found in the input at that position, and what the parser was expecting to find. 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 [11].So labeled PEGs were introduced.

3. Error reporting in PEG

A PEG G is a tuple (V,T, P, pS) where V is a finite set of nonterminals, T is a finite set of terminals, P is a total function from non-terminals to parsing expressions and pS is the initial parsing expression and the function P as a set of rules of the form A ← p, where A ∈ V and p is a parsing expression[9].When a parsing expression is applied to an input string it can either consumes a prefix of the input string and return the remaining suffix or it can fail.

Prog ← PUBLIC CLASS NAME LCUR PUBLIC STATIC VOID MAIN LPAR STRING LBRA RBRA NAME RPAR *BlockStmt* RCUR
BlockStmt ← LCUR (*Stmt*)∗ RCUR
Stmt ← *IfStmt* / *WhileStmt* / *PrintStmt* / *DecStmt* / *AssignStmt/ BlockStmt*
IfStmt ← IF LPAR *Exp* RPAR *Stmt* (ELSE *Stmt* /ε)
WhileStmt ← WHILE LPAR *Exp* RPAR *Stmt*
DecStmt ← INT NAME (ASSIGN *Exp* / ε) SEMI
AssignStmt ← NAME ASSIGN *Exp* SEMI
PrintStmt ← PRINTLN LPAR *Exp* RPAR SEMI
Exp ← *RelExp* (EQ *RelExp*)∗
RelExp ← *AddExp* (LT *AddExp*)∗
AddExp ← *MulExp* ((PLUS / MINUS) *MulExp*)∗
MulExp ← *AtomExp* ((TIMES / DIV) *AtomExp*)∗
AtomExp ←LPAR *Exp* RPAR / NUMBER / NAME

Fig. 1. A PEG for a tiny subset of Java [9]

Figure 1 [9] shows a PEG for a tiny subset of Java, where lexical rules (shown in uppercase) have been elided.

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

Fig. 2. A Java program with a syntax error

Fig. 2 depicts a java program with syntax errors. First error is the missing semicolon at the end of line 7 and the second error an extra semicolon at the end of line 8. The top down predictive parser will detect the first error when it reads the RCUR({) token at line 8 and provides a report to the user for missing semicolon. But PEG when trying to parse the SEMI rule which should match a ‘; ‘when the input has’ } ‘ will fail and does not report the error to the user. During the parsing of PEGs if any failure occurs, it means it has to backtrack and try another alternative in an orderly manner or terminate the repetition. 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, while trying to match the BlockStmt rule inside Prog against the ‘a’ at the beginning of line 3, three failures will occur. First against IF in the IfStmt rule second for WHILE in the WhileStmt rule and lastly against PRINTLN in the PrintStmt rule.

After all the failing and backtracking, the PEG in this example, after consuming only the first two statements of the body of main will fail in the RCUR rule of the initial BlockStmt. Failure to match the SEMI in AssignStmt against the closing ‘{‘ in the input will cause PEG to backtrack and try the ordered next alternative in Stmt which will also fail. The repetition inside the BlockStmt that is parsing the body of the while statement will end. The whole BlockStmt trying to match RCUR against the ‘a’ in the beginning of line 7 will fail, thereby causing the whole WhileStmt to fail and makes PEG to backtrack at the beginning of the line5. The process now repeats with the BlockStmt that is parsing the body of main.

At the end the PEG reports that it failed complaining that while in the input does not match the RCUR that it expects and hence it cannot proceed to the beginning of line 5. This does not help the programmer in fixing and finding the actual error. To deal with this problem, Ford [8] 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 [10]. Using the farthest failure approach in our above example it will report an error at the beginning of line 8. The same would have been done by a predictive parser.

To track expected tokens in the error position we can even use a map of lexical rules to token names to report for a expecting semicolon. If the programmer fixes this error , the parser fails repeatedly at the extra semicolon at line 8, while it is trying to match the first term of all the alternatives of the Stmt which will end the repetition inside BlockStmt, thereby another failure will happen when trying to match a RCUR token against the semicolon and finally aborting the parser. 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 parser can use the furthest failure information.

Major advantage of using the farthest failure is that the error messages can be generated automatically and the grammar writer does not need to do anything to get a parser with better error reporting.

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 o fix the error, and may contain a long list of expected tokens [11]. Using PEG with labeled failures, which is a conservative extension of the PEG formalism we can get more precise error messages. A labeled PEG G is a tuple (V,T, P, L, fail,pS) where L is a finite set of labels, fail ∉ L is a failure label, and the expressions in P have been extended with the throw operator, represented by ⇑[9]. The parsing expression ⇑l where l ∈ L will generate a failure with label l. A label l ≠ fail thrown by ⇑ indicates an actual error during parsing as it cannot be caught by an ordered choice while fail indicates that the parser should backtrack as it is caught by a choice. The lookahead operator! captures any label and turns it into a success. And meanwhile turning a success into a fail label.

Prog ← PUBLIC CLASS NAME LCUR PUBLIC STATIC VOID MAIN LPAR STRING LBRA RBRA NAME RPAR *BlockStmt* RCUR

BlockStmt ← LCUR (*Stmt*)∗ $[\text{RCUR}]^{rcblk}$

Stmt ← *IfStmt* / *WhileStmt* / *PrintStmt* / *DecStmt* / *AssignStmt* / *BlockStmt*

IfStmt ← IF $[\text{LPAR}]^{lpif}$ $[Exp]^{condi}$ $[\text{RPAR}]^{rpif}$ $[Stmt]^{then}$(ELSE $[Stmt]^{else}$ / ε)

WhileStmt ← WHILE $[\text{LPAR}]^{lpw}$ $[\text{Exp}]^{condw}$ $[\text{RPAR}]^{rpw}$ $[Stmt]^{body}$

DecStmt ← INT $[\text{NAME}]^{ndec}$ (ASSIGN $[Exp]^{edec}$ / ε) $[\text{SEMI}]^{semid}$

AssignStmt ← NAME $[\text{ASSIGN}]^{assign}$ $[Exp]^{rval}$ $[\text{SEMI}]^{semia}$

PrintStmt ← PRINT $[\text{LPAR}]^{lpp}$ $[Exp]^{eprint}$ $[\text{RPAR}]^{rpp}$ $[\text{SEMI}]^{semip}$

Exp ← *RelExp* (EQ $[RelExp]^{relexp}$)∗

RelExp ← *AddExp* (LT $[AddExp]^{addexp}$)∗

AddExp ← *MulExp* ((PLUS / MINUS) $[MulExp]^{mulexp}$)∗

MulExp ← *AtomExp* ((TIMES / DIV) $[AtomExp]^{atomexp}$)∗

AtomExp ← LPAR $[Exp]^{parexp}$ $[\text{RPAR}]^{rpe}$ /NUMBER / NAME

Fig. 3. A PEG with labels for a small subset of Java [9]

Different labels can be mapped to different error messages and then annotate our PEG with these labels. Fig. 3 show annotation of the PEG of Fig. 1 [except for the Prog rule]. The expression [p]l is syntactic sugar for (p / ⇑l) [9]. The strategy used is as follows: we annotate every symbol that is either terminal or non-terminal that should not fail on the right side of the production. Making the PEG to backtrack on the failure of that symbol would be futile as the whole parse would either consume the whole input or either fail. When we use this labeled PEG in our program the first occurring syntax error will fail with a semia label. This can be mapped to a “missing semicolon in assignment" message. When the programmer fixes this error, the second occurring error will fail with a rcblk label. This can be mapped to a “missing end of block" message. When we compare the farthest failure approach with the labeled failure approach, one disadvantage of the latter is the burden of annotation. If we combine both approaches we can still track even in case of furthest failure the position of the failure and the expected set of lexical rules.

4. Error recovery in PEGs

In this section we discuss two of the error recovery techniques. The first error recovery technique is based on the labeled failures which is based on the error reporting method discussed in the previous section and the second error recovery technique is the automatic error recovery technique.

A. Error recovery through labeled failures

This error recovery uses the labeled error reporting method discussed in section 3 as its first step. Let us consider the example from Fig. 2, which has syntax errors: a missing ‘)’ at line 5, and a missing semicolon at the end of line 7. For this, a parser based on the fig. 3 labeled PEG would give us a message as:

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

As the parser did not recover from the first error, the second error will not be reported, since rpw has no recovery expression mapped with it. The recovery expression pr of a label l matches the input from the point where l was thrown. Regular parsing is resumed if pr is successful and it seems as if the label had not been thrown. Usually pr should skip part of the input till its is fine to continue parsing. In rule WhileStmt, we see that after the ‘)’ we expect to match a Stmt, so the recovery expression of label rpw 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 [12]-[14]. With the help of these sets, we can define the following recovery expression for rpw, where ‘.’ is a parsing expression that matches any character: [15] (!FIRST(Stmt) .)∗ [15]

When label rpw is thrown now, its recovery expression matches the input till it finds the start of a statement, and then regular parsing continues. The parser will now also throw label semia and report the second error, the missing semicolon at the end of line 7. The above example shows how the error recovery using labeled failures works. The disadvantage of this method is that it is quite burdensome to manually annotate the grammar with labels. Even the small example discussed here has 26 labels with recovery expressions.

B. Automatic error recovery

To avoid the burden caused by the above discussed error recovery method, the following algorithm was proposed to automatically annotate the grammars with labels and recovery expressions [15].

Algorithm 1: Automatically Inserting Labels and Recovery Expressions in a PEG [15].

```
1: function annotate(G)
```

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

A grammar writer can either add or remove labels and their respective recovery expressions after applying the Algorithm 1.

5. Conclusion

In this paper we have discussed the error recovery and error reporting methods that were proposed by various authors, used for parsers based on Parsing Expression Grammars. The error reporting method is based on using PEGs with labelled failures. This method reports the error messages better and in a precise way. The two error recovery methods used are also based on labelled PEGs. Every label is associated with a recovery expression which is used for recovery and by using the Algorithm 1, labels and recovery expressions can automatically be annotated with a grammar.

References

[1] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. Journal of the ACM, 49(1):1– 15, 2002.

[2] Amir Shpilka. Lower bounds for matrix product. In IEEE Symposium on Foundations of Computer Science, pages 358– 367, 2001.

[3] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic descriptions. Communications of the ACM, 20(11):822–823, November 1977.

[4] International Standards Organization. Syntactic metalanguage — Extended BNF, 1996. ISO/IEC 14977.

[5] Alexander Birman. The TMG Recognition Schema. PhD thesis, Princeton University, February 1970.

[6] Alexander Birman and Jeffrey D. Ullman. Parsing algorithms with backtrack. Information and Control, 23(1):1–34, August 1973.

[7] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In Proceedings of the 2002 International Conference on Functional Programming, Oct 2002.

[8] B. Ford, Packrat parsing: a practical linear-time algorithm with backtracking, Master's thesis, Massachusetts Institute of Technology, September 2002.

[9] Sérgio Medeiros, Fabio Mascarenhas: Syntax error recovery in parsing expression grammars. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing.

[10] D. Grune, C.J. Jacobs, Parsing techniques: A Practical guide, 2nd, Springer, Berlin, Heidelberg, Germany, 2010.

[11] André Murbach Maidl, Fabio Mascarenhas, Sérgio Medeiros, and Roberto Ierusalimschy. 2016. Error Reporting in Parsing Expression Grammars. Sci. Comput. Program. 132, P1 (Dec. 2016), 129–140.

[12] Fabio Mascarenhas, Sérgio Medeiros, and Roberto Ierusalimschy. 2014. On the relation between context-free grammars and parsing expression grammars. Science of Computer Programming 89 (2014), 235 – 250.

[13] Roman R. Redziejowski. 2009. Applying Classical Concepts to Parsing Expression Grammar. Fundamenta Informaticae 93 (January 2009), 325–336. Issue 1-3.

[14] Roman R. Redziejowski. 2014. More About Converting BNF to PEG. Fundamenta Informaticae 133 (2014), 257–270. Issue 2-3.

[15] Sérgio Queiroz de Medeiros and Fabio Mascarenhas. 2018. Towards Automatic Error Recovery in Parsing Expression Grammars. In Proceedings of Brazilian Symposium on Programming Languages (SBLP'18).

[16] 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, NewYork, NY, USA, 2004, pp.111-122.