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

Recognition based Syntactic Foundation, Relation with CFG's and Syntax Error Recovery in Parsing Expression Grammars

Sini Anna Alex[1], Aditya Raghu[2], K. N. Ajay Shastry[3], H. R. Chetan[4]
[1]*Assistant Professor, Dept. of Computer Science and Engg., Ramaiah Institute of Technology, Bangalore, India*
[2,3,4]*Student, Dept. of Computer Science and Engg., Ramaiah Institute of Technology, Bangalore, India*

***Abstract*: For many years, Chomsky's generative system of grammar has been in regular use, to express syntax of programming languages and protocols. This hereby induces unneeded difficulty to parse machine-oriented languages. PEG (Parsing Expression grammar) provide an alternate recognition based foundation which solves ambiguity problem. PEG's uses prioritized choice instead of alternatives. Parsing Expression Grammars (PEGs) are nothing but formalisms used to outline top-down parsers with backtracking. As PEGs don't provide a good error recovery mechanism they usually don't recover from syntax errors in the input. PEG parsers hence become unfit for use with Integrated Development Environments (IDEs), which need to build syntactic trees even for incomplete, syntactically invalid programs. We propose an extension for PEGs with labelled failures, which introduces a syntax error recovery mechanism for it. Here a label now not only reports a syntax error but also uses this recovery expression to reach a synchronization point in the input and resume parsing. We show a technique removing non-determinism from a formalism yields a formalism with the semantics of PEGs. Based on these new formalisms, we also prove how LL (1) grammars define the same language whether interpreted as CFGs or as PEGs.**

***Keywords*: syntactic foundation, parsing expression grammars**

1. Introduction

CFG formalisms are opted by choice for outlining the syntax of programming languages. A CFG narrates a language by a set of strings generated from the grammar's initial symbol by a sequence of steps that have been rewritten. However, it does not specify how to parse a language which is extremely crucial for working with the specified language in a compiler. We also have the problem of ambiguity which must and should be eliminated. A PEG in contrast to the one explained above defines a language in terms of predicates that infer whether or not a given string is in the language. Simple languages can be expressed easily in both kinds of grammars.

PEGs are similar in style to that of CFGs with features similar to that of Regular Expressions and also like EBNF (Extended Backus-Naur Form) notation. An important difference between both is that instead of the unordered choice operator `|' used to indicate alternative expansions for a non-terminal, PEGs use a prioritized choice operator `/'. PEGs always avoid ambiguities in the definition of their grammar language due to the use of an ordered choice operator.

PEG can be understood as being similar to a recursive descent parser with restricted (or local) backtracking. All the errors in a PEG should not be considered a failure, but it should be considered as an indication to backtrack and try another alternative.

2. PEG (parsing expression grammar)

Definition: A *parsing expression grammar* (PEG) is a 4-tuple $G = (V_N, V_T, R, e_S)$, where V_N is a finite set of non-terminal symbols, V_T is a finite set of terminal symbols, R is a finite set of rules, e_S is a parsing expression termed the *start expression*, and $V_N \cap V_T = \phi$. Each rule $r \in R$ is a pair (N, e), which we write $N \leftarrow e$, where $N \in V_N$ and e is a parsing expression. For any nonterminal N, there is exactly one e such that $N \leftarrow e \in R$. R is, therefore, a function from non-terminals to expressions, and we write $R(N)$ to denote the unique expression e such that $N \leftarrow e \in R$.

We define *parsing expressions* inductively as follows. If e, e_1, and e_2 are parsing expressions, then so is:

1. ε, the empty string
2. a, any terminal, where $a \in V_T$.
3. A, any nonterminal, where $A \in V_N$.
4. $e_1\ e_2$, a sequence.
5. $e_1\ / e_2$, prioritized choice.
6. e^* , zero-or-more repetitions.
7. $!e$, a not-predicate.

Single or double quotes are used as delimiters for string literals. Literals and character classes usually contain C-like escape codes. To match a single character '.' is used.

The sequence expression `$e_1\ e_2$' looks for a match of the expression $e_1$ immediately followed by a match of the expression $e_2$, backtracks to the starting point if a failure situation occurs. The choice expression `$e_1 = e_2$' first attempts pattern e_1, then attempts e_2 from the same starting point if e_1 fails.

Operators '&' and '!' are syntactic predicates, which provide practical expressive power of PEGs. Conversely, the expression `!*e*' fails if *e* succeeds, but succeeds if *e* fails.

3. From CFG to PEG

A tuple (*V*, *T*, *P*, *S*) of a finite set *V* of non-terminals symbols, a finite set *T* of terminal symbols, a finite relation *P* between non-terminals and strings of terminals and non-terminals, and an initial non-terminal S. We say that $N \rightarrow \alpha$ is a production of G if and only if $(N, \alpha) \in P$.

A grammar G defines a relation $\Rightarrow_G$ where $\alpha A \gamma \Rightarrow_G \alpha \beta \gamma$ if and only if $A \rightarrow \beta$ is a production of G. The language of G is the set of all strings of terminal symbols that relate to S by the reflexive–transitive closure of $\Rightarrow_G$.

We now define a PE-CFG (short for CFG using parsing expressions) G as a tuple (V, T, P, p_S), where V and T are still the sets of non-terminals and terminals, but P is now a function from non-terminals to parsing expressions, and p_S is the initial parsing expression of the grammar. As P is a function, we will use the standard notation for function application, P(A), to refer to the parsing expression associated with a non-terminal A in G.

Instead of the relation $\Rightarrow_G$, we define a new relation, CFG, among a grammar G, a string of terminal symbols v, and CFG another string of terminal symbols w. We will use the notation Gv CFG w to say that $(G, v, w) \in$ CFG. The intuition for the CFG relation is that the first string is the input, and the second string is a suffix of the input that is left after G matches a CFG prefix of this input. We will usually say G xy CFG y to mean that G matches a prefix x of input string xy.

As an example, we have a CFG G with the productions:

$P = \{X \dashrightarrow YZ, Y \rightarrow x, Y \rightarrow y, Z \rightarrow z, Z \rightarrow d, Z \rightarrow e\}$
Its corresponding PE-CFG T (G) = G has the definition for the function P:
P (X) = YZ P (Y) = x | yP (Z) = z | d | e

Commutativity and associativity of the choice operator allow any order that would yield a grammar with the same language as above, so we could use this definition instead:

$$P(X) = YZ \qquad P(Y) = x / y \qquad P(Z) = e / z / d$$

A. *Unified language definition*

1) Conventionally syntax descriptions are split into two parts: Context-free grammar to show hierarchical portion.
2) Regexes to define lexical elements to serve as terminals.

CFGs' are actually unsuitable for lexical syntax as they can't directly express many of the common idioms or negative syntaxes. Regular expressions can't describe the recursive syntax.

Associating whitespace with each immediately preceding token is a convenient convention for PEGs, but whitespace could just as easily be associated with the *following* token by referring to Spacing at the *beginning* of each token definition.

B. *New syntax design choices*

Consider that a unified PEG describes a language, however, it 's extremely easy to characterize the language to allow a `>>' sequence to be thought of as either one token or two tokens depending on its context.

String literals permit escape sequences in most programming languages for the sake of expressing dynamic string distributions. A unified PEG that describes a language, can permit the use of arbitrary expressions in escapes, by exploiting the complete power of the expression syntax of the language.

C. *Priorities, Not Ambiguities*

Constructs that are inherently ambiguous when expressed as a CFG, usually lead designers of language to relinquish syntactic formality and depend on informal meta-rules. The all-pervasive "dangling ELSE" is a classic example of this, traditionally requiring an informal meta rule. The prioritized choice operator in a PEG easily expresses this.

D. *Quirks and Limitations*

CFG’s permit both *left* and *right recursion.* Since PEG’s represent regenerate loop, left recursion is unavailable with top-down parsing. For example, the CFG rules ‘$A \rightarrow a\ A \mid a$’ and ‘$A \rightarrow A\ a \mid a$’ represents a series of ‘a’s in a CFG, but the PEG rule ‘$A \leftarrow A\ a / a$’ is degenerate because it indicates that in order to recognize nonterminal A, a parser must first recognize nonterminal A.

Left and right recursion in a CFG represent only repetition. However, repetition is easier to express in a PEG using repetition operators.

E. *Desugaring the Concrete Syntax*

The abstract syntax does not include the following which appear in the concrete syntax.

- character classes
- the “any character” constant ‘.’
- the option operator ‘?’
- the one-or-more- repetitions operator ‘+’
- the and-predicate operator ‘&’

We treat these concrete syntax features as “syntactic sugar,” reducing them to abstract parsing expressions using local substitutions as follows and e_d denotes the desugaring of e.

- We consider the ‘.’ expression in the concrete syntax as a character class containing all of the terminals in V_T.
- If a_1, a_2,..., a_n are all of the terminals listed in a character class expression in the concrete syntax, then we desugar this character class expression to the abstract syntax expression $a_1/a_2/.../a_n$.
- We desugar an option expression *e*? to e_d/ε.
- We desugar a one-or-more-repetitions expression *e*+ to $e_d e^*_d$.
- We desugar an and-predicate &*e* to $!(!e_d)$.

F. Language Properties

Definition: A language L over an alphabet V_T is a parsing expression language (PEL) if there exists a parsing expression grammar G whose language is L.

Theorem: The class of parsing expression languages is closed under union, intersection, and complement.

Theorem: The class of PELs includes non-context-free languages.

Proof: The classic example language $a^n\ b^n\ c^n$ is not context-free, but we can recognize it with a PEG G = $(\{A,B,D\},\{a,b,c\},R,D)$, where R contains the following definitions:

$A \leftarrow aAb/\varepsilon$

$B \leftarrow bBc/\varepsilon$

$D \leftarrow \&(A!b)a^*\ B!.$

4. Reduction to PFGs

A. Eliminating repetition operators

As in CFGs, repetition expressions can be eliminated from a PEG by converting them into recursive non-terminals. Unlike in CFGs, the nonterminal to be substituted in a PEG must be right-recursive.

Theorem: Any repetition expression e^* can be eliminated by replacing it with a new nonterminal A with the definition $A \leftarrow eA/\varepsilon$.

Proof: By induction on the length of the input string.

Theorem: For any PEG G, an equivalent repetition-free grammar G' can be created.

Proof: Simply eliminate all repetition expressions throughout G's nonterminal definitions and start expression.

B. Eliminating predicates

For any well-formed, repetition-free grammar $G = (V_N, V_T, R, e_S)$ where $\varepsilon \notin L(G)$, we will create an equivalent well-formed, repetition-free, and predicate- free grammar $G' = (V_N{}' ,V_T ,R' ,e'_S)$. This process occurs in three normalization stages. In the first stage, we rewrite the grammar so that sequence and predicate expressions only contain non-terminals and choice expressions are disjoint. In the second stage, we rewrite the grammar in a way where non-terminals never succeed without consuming any input. In the third stage, we finally eliminate predicates.

1) Stage 1

We define a function f recursively as follows, to convert expressions in our original grammar G into our first normal form:

1. $f(e) = e$ if $e \in\{\varepsilon\} \cup V_N \cup V_T$.
2. $f(e_1e_2) = AB$, adding $A \leftarrow f(e_1)$ and $B \leftarrow f(e_2)$ to R_1.
3. $f(e_1/e_2) = A/!A\ f(e_2)$, adding $A \leftarrow f(e_1)$ to R_1.
4. $f(!e) =!A$, adding $A \leftarrow f(e)$ to R_1.

Definition: The *stage 1 grammar* G_1 of G is $(V_N{}', \mathrm{VT}, R_1, eS_1)$, where $e_{S1} = f(e_S)$, $R_1 = \{A\leftarrow f(e)|A\leftarrow e \in R\} \cup$\{new definitions resulting from application of f \}, and $V_N{}' = V_N \cup$ \{new non-terminals resulting from application of f \}.

2) Stage 2

We use two functions g_0 and g_1, to "split" expressions into ε*-only* and ε*-free* parts, respectively. The ε*-only part* $g_0(e)$ of an expression e is an expression that yields the same result as e on all input strings for which e succeeds without consuming any input, and fails otherwise. The ε*-free part* $g_1(e)$ of e likewise yields the same result as e on all inputs for which e succeeds and consumes at least one terminal, and fails otherwise.

We first define g_0 recursively as follows:

1. $g_0(\varepsilon)=\varepsilon$.
2. $g_0(a)=F$.
3. $g_0\ (A) = g_0\ (R_G\ (A))$.
4. $g_0(AB) = g_0(A)g_0(B)$ if $A \rightharpoonup 0$, otherwise $g_0(AB) = F$.
5. $g_0(e_1/e_2)=g_0(e_1)/g_0(e_2)$.
6. $g0(!A)=!(A/g0(A))$.

Lemma: If G is well-formed, then function $g0$ terminates.

Proof: By structural induction over the $W\ FG$ relation. Termination relies on $g0\ (AB)$ not recursively invoking $g0\ (B)$ if $A \not\rightharpoonup 0$.

4.2.3 Stage 3

Finally we rewrite $G2$ into the final grammar $G' = (VN' ,VT ,R',e'). S$

Definition: We define a function d, such that $d(A,\ e)$ "distributes" a nonterminal A into an \ε-only expression e resulting from the stage 2 function $g0$:

1. $d(A, e) = e$, if $e \in \{\varepsilon, F\}$.
2. $d(A, e1e2) = d(A, e1)\ d(A, e2)$.
3. $d(A, e1/e2) = d(A, e1) / d(A, e2)$.
4. d(A, !e)=!(Ae).

3) The empty string limitation

We prove that any predicate-free grammar cannot accept the empty input string without accepting all input strings to show that we have no hope of avoiding the restriction that the original grammar cannot accept the empty input string,

Lemma: Assume that G is a predicate-free grammar and that for any expression e and input x of length n or less, $(e, \varepsilon) \Rightarrow^+$ if $(e, x) \Rightarrow^+ \varepsilon$. Then the same holds for input strings of length $n + 1$.

Proof: By induction over step counts in $\Rightarrow_G$.

Theorem: In a repetition-free grammar G, an expression e matches the empty string if it matches *all* input strings and produces only ε results. In consequence, $\in L(G)$ implies $L(G) = V_T{}^*$.

Proof: By induction over string length.

5. PEG Error Recovery

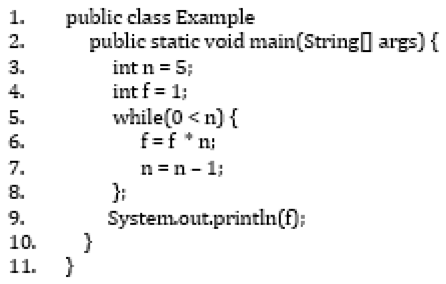
In this section, we revisit the problem of error handling in PEGs and show how labelled failures [1], [2] combined with the farthest failure heuristic [3] can improve the error messages of a PEG-based parser. Then we show how labelled PEGs can be the basis of an error recovery mechanism for PEGs, and show an extension of previous semantics for labelled PEGs that adds recovery expressions.

A. Error recovery

Fig. 1 is an example of a Java program with two syntax errors (a missing semicolon at the end of line 7, and an extra semicolon at the end of line 8). A predictive top-down parser will detect the first error when reading the RCUR (}) token at the beginning of line 8 and will know and report to the user that it was expecting a semicolon.

Fig. 2 shows a PEG for a tiny subset of Java, where lexical rules (shown in uppercase) have been elided. While simple (this PEG is equivalent to an LL(1) CFG), this subset is already rich enough to show the problems of PEG error reporting; a more complex grammar for a larger language just compounds these problems.

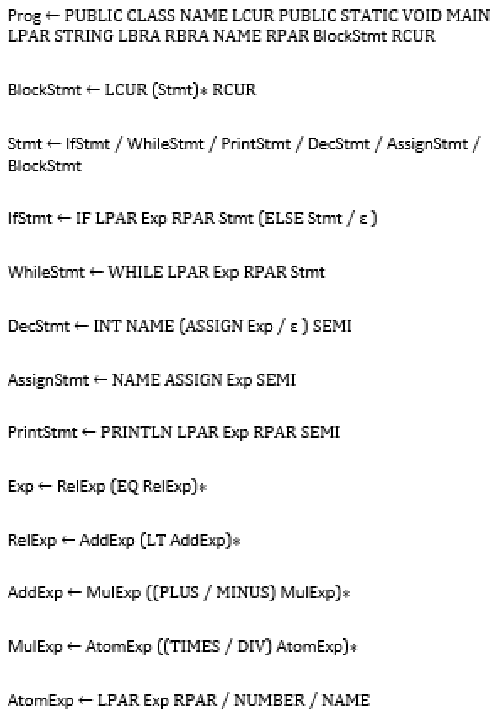
In the case of our PEG, it will still fail when trying to parse the SEMI rule, which should match a ';', while the input has a closing curly bracket, but as a failure does not guarantee the presence of an error the parser cannot report this to the user. Failure during parsing of a PEG usually just means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



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

Fig. 1. A Java program with a syntax error

After all the failing and backtracking, the PEG in our example will ultimately fail in the RCUR rule of the initial BlockStmt, after consuming only the first two statements of the body of main. Failing to match the SEMI in AssignStmt against the closing curly bracket in the input will make the PEG backtrack to the beginning of the statement to try the other alternatives in Stmt, which also fail. This marks the end of the repetition inside the BlockStmt that is parsing the body of the while statement. The whole BlockStmt will fail trying to match RCUR against the 'n' in the beginning of line 7, this ultimately makes the whole WhileStmt fail, which makes the PEG backtrack to the beginning of line 5. In the end, the PEG will report that it failed and cannot proceed at the beginning of line 5, complaining that the while in the input does not match the RCUR that it expects, which does not help the programmer in finding and fixing the actual error. To circumvent this problem, Ford [3] 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 [4]. In our previous example, the use of the farthest failure approach reports an error at the beginning of line 8, the same as a predictive parser would. We can even use a map of lexical rules to token.



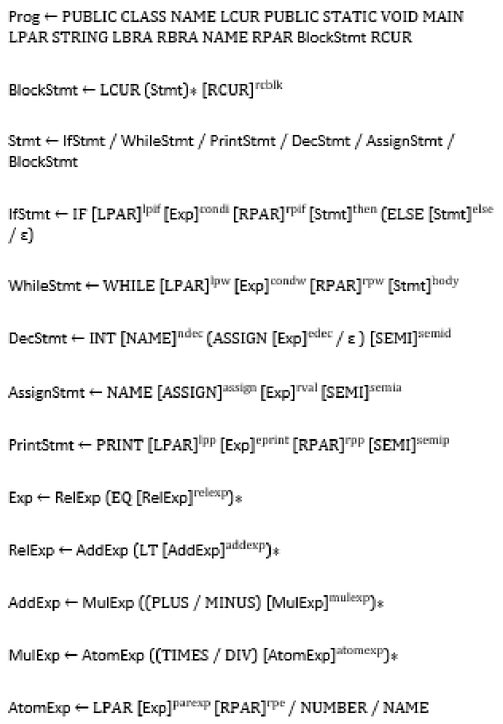
```
Prog ← PUBLIC CLASS NAME LCUR PUBLIC STATIC VOID MAIN
LPAR STRING LBRA RBRA NAME RPAR BlockStmt RCUR

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. 2. A PEG for a tiny subset of Java

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

The great advantage of using the farthest failure is that the grammar writer does not need to do anything to get a parser with better error reporting, as the error messages can be generated automatically. However, although this approach gives us error messages with a fine approximation of the error location, these messages may not give a good clue about how to fix the error and may contain a long list of expected tokens [1].

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



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

BlockStmt ← LCUR (Stmt)∗ [RCUR]^{rcblk}

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

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

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

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

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

PrintStmt ← PRINT [LPAR]^{lpp} [Exp]^{eprint} [RPAR]^{rpp} [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} [RPAR]^{rpe} / NUMBER / NAME

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

B. *Error recovery strategies for PEGs*

A parser with a good recovery mechanism is essential for use in an IDE, where we want an AST that captures as much information as possible about the program even in the presence of syntax errors due to an unfinished program.

We can improve the error recovery quality of a PEG parser by using the FIRST and FOLLOW sets of parsing expressions when throwing labels or recovering from an error. A detailed discussion about FIRST and FOLLOW sets in the context of PEGs can be found in other papers [2, 5, 6].

In our grammar for a subset of Java, we can see that whenever rule Exp is used it should be followed by either a right parenthesis or a semicolon, so based on the FOLLOW set of Exp, we could define (!(RPAR / SEMI) .)∗ as a recovery expression. Differently, from the rcblk recovery expression, this one does not consume the synchronization symbols, as they should be consumed by the following expression.

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

The use of the FOLLOW set (probably enhanced by a synchronization symbol such as ‘;’) provides a default error recovery strategy. Let us apply this strategy for our annotated Java grammar and consider that the Java program from Figure 2 has an error on line 5, inside the condition of while loop, as follows:

5 while (< n) {

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

6. Evaluation

A. *Error recovery in a Lua parser*

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

We will evaluate our strategy following Pennelo and DeRemmer’s approach, however, we will compare the AST got from an erroneous program after recovery with the AST of what would be the equivalent correct program, instead of comparing program texts.

Based on this strategy, a recovery is excellent when it gives us an AST equal to the intended one. A good recovery gives us a reasonable AST, i.e., one that captures most information of the original program does not report spurious errors, and does not miss other errors. A poor recovery, by its turn, produces an AST that loses too much information, results in spurious errors, or misses errors. Finally, recovery is rated as failed whenever it fails to produce an AST at all.

To evaluate our error recovery strategy, we built a PEG parser for the Lua programming language [7] using the LPegLabel tool, in which support for associating labels with recovery expressions has been added to its current version [13]. Our parser is based on the syntax defined in the Lua 5.3 reference manual 3 (https://www.lua.org/manual/5.3/) and builds the AST associated with a given program.

We used 75 different labels to annotate Lua grammar. The process of annotating the Lua grammar with labels was done manually, as well as the process of writing the recovery expressions for each label. Our parser was always able to build an AST, given that no recovery expression raised an unrecoverable error, or entered a loop.

Table 1
Evaluation of our Recovery Strategy Applied to a Lua Parser

Excellent	Good	Poor	Failed	Total
100 (≈ 56%)	63 (≈ 35%)	17 (≈ 9%)	0	180

7. Conclusion

We presented a new formalism for context-free grammars that is based on recognizing (parts of) strings instead of generating them. We adopted a subset of the syntax of parsing expression grammars, and the notion of letting a grammar recognize just part of an input string, to purposefully get a definition for CFGs that is closer to PEGs, yet defines the same class of languages as traditional CFGs. These PE-CFGs define the same class of language as traditional CFGs, and simple transformations let us get a PE-CFG from a CFG and vice-versa.

Parsing expression grammars provide a powerful, formally rigorous, and efficiently implementable foundation for expressing the syntax of machine-oriented languages that are designed to be unambiguous. Because of their implicit longest-match recognition capability coupled with explicit predicates, PEGs allow both the lexical and hierarchical syntax of a language to be described in one concise grammar. The expressiveness of PEGs also introduces new syntax design choices for future languages.

We have presented a conservative extension of PEGs that is well- suited for implementing parsers with a robust mechanism for recovering from syntax errors in the input. Our extension is based on the use of labels to signal syntax errors, and differentiates them from regular failures, together with the use of recovery expressions associated with those labels. When signalling an error with a label that has an associated recovery expression, the parser logs the label and the error position, then proceed with the recovery expression. This recovery expression is a regular parsing expression, with access to all the parsing rules that the grammar provides.

References

[1] Andre´ Murbach Maidl, Fabio Mascarenhas, Se´rgio Medeiros, and Roberto Ierusalimschy. 2016. Error Reporting in Parsing Expression Grammars. Sci. Comput. Program. 132, P1 (Dec. 2016), 129–140.

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

[3] Bryan Ford. 2002. Packrat Parsing: A Practical Linear-Time Algorithm with Backtracking. Master's thesis. Massachusetts Institute of Technology.

[4] Dick Grune and Ceriel J.H. Jacobs. 2010. Parsing Techniques: A Practical Guide (2nd ed.). Springer, Berlin, Heidelberg, Germany.

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

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

[7] Roberto Ierusalimschy. 2016. Programming in Lua (4th ed.). Lua.Org, Rio de Janeiro, Brazil.