Exception Handling for Error Reporting in Parsing Expression Grammars

André Murbach Maidl[1], Fabio Mascarenhas[2], Roberto Ierusalimschy[1]

[1] Department of Computer Science – PUC-Rio – Rio de Janeiro – Brazil
{amaidl,roberto}@inf.puc-rio.br
[2] Department of Computer Science – UFRJ – Rio de Janeiro – Brazil
fabiom@dcc.ufrj.br

Abstract. Parsing Expression Grammars (PEGs) are a new formalism to describe a top-down parser of a language. However, error handling techniques that are often applied to top-down parsers are not directly applicable to PEGs. This problem is usually solved in PEGs using a heuristic that helps to simulate the error reporting technique from top-down parsers, but the error messages are generic. We propose the introduction of labeled failures to PEGs for error reporting, as labels help to produce more meaningful error messages. The labeled failures approach is close to that of generating and handling exceptions often used in programming languages, being useful to annotate and label grammar pieces that should not fail. Moreover, our approach is an extension to the PEGs formalism that is expressive enough to implement some previous work on parser combinators. Finally, labeled failures are also useful to compose grammars preserving the error messages of each separate grammar.

Keywords: parsing, error reporting, parsing expression grammars, packrat parsing, parser combinators

1 Introduction

When a parser receives an erroneous input, it should indicate the existence of syntax errors. However, a generic error message (e.g. `syntax error`) does not help the programmer to find and fix the errors that the input may have. Therefore, the least that is expected from a parser is that it should produce an error message indicating the position of an error in the input and some information about the context of this error. The LL and LR methods detect syntax errors very efficiently because they have the *viable prefix* property, that is, these methods detect a syntax error as soon as a token is read and cannot be used to form a viable prefix of the language [1].

Usually, there are two ways to handle errors: error reporting and error recovery. In error reporting, the parser aborts with an informative message when the first error is found. In error recovery, the parser is adapted to not abort on the first error, but to try processing the rest of the input, informing all errors that it found. Such error handling techniques are described in more detail in [1] and

[5]. In this paper we focus on error reporting because error recovery can produce cascading errors.

Parsing Expression Grammars (PEGs) [4] are a new formalism for describing the syntax of programming languages. We can view a PEG as a formal description of a top-down parser for the language it describes. The syntax of PEGs has similarities to Extended Backus-Naur Form (EBNF), but, unlike EBNF, PEGs avoid ambiguities in the definition of the grammar's language due to the use of an ordered choice operator. More specifically, a parser implemented by a PEG is a recursive descent parser with restricted backtracking. This means that the alternatives of a non-terminal are tried in order; when the first alternative recognizes an input prefix, no other alternative of this non-terminal is tried, but when an alternative fails to recognize an input prefix, the parser backtracks on the input to try the next alternative.

On the one hand, PEGs are an expressive formalism for describing top-down parsers [4]; on the other hand, PEGs cannot use error handling techniques that are often applied to top-down parsers, because these techniques assume the parser reads the input without backtracking [2]. In top-down parsers without backtracking, it is possible to signal a syntax error when there is no alternative to continue reading. In PEGs, it is more complicated to identify the cause of an error and the position where it happened because failures during parsing are not necessarily errors, but just an indication that the parser should backtrack and try a different alternative.

Ford [2] provided a heuristic to the problem of error handling in PEGs. His heuristic simulates the error reporting technique that is implemented in top-down parsers without backtracking. However, the error messages produced by both regular top-down parsers and parsers that use this heuristic are still generic. The best the parsers can do is to tell the user the position where the error happened, what was found in the input and what they were expecting.

In this paper we present a new approach for error reporting in PEGs, based on the concept of *labeled failures*. In our approach, each label may be tied to a specific error message and resembles the concept of exceptions from programming languages. Our approach is not tied to a specific implementation of PEGs, being an extension to the PEGs formalism itself. We show how to use labeled failures to implement error reporting. We also show that our extension is expressive enough to implement alternative error reporting techniques from top-down parsers with backtracking.

The rest of this paper is organized as follows: in Section 2 we contextualize the problem of error handling in PEGs and we also explain in detail the heuristic that Ford used to implement error reporting. In Section 3 we discuss alternative work on error reporting for top-down parsers with backtracking. In Section 4 we introduce the concept of labeled failures, show how to use it for error reporting, and show how labeled failures can encode some of the techniques of Section 3. Finally, we draw our conclusions in Section 5.

2 Error Reporting in PEGs

In this section we use examples to present in more detail how a PEG behaves badly on the presence of syntax errors. After that, we present a heuristic proposed by Ford [2] to implement error reporting in PEGs. Rather than using the original definition of PEGs by Ford [4], our examples use the equivalent and more concise definition proposed by Medeiros et al. [12, 13]. We will extend this definition in Section 4 to present a semantics for PEGs with labeled failures.

A PEG G is a tuple (V, T, P, 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* and p_S is the initial parsing expression. 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 fails or consumes a prefix of the input resulting in the remaining suffix. The abstract syntax of parsing expressions is given as follows:

$$p = \varepsilon \mid a \mid A \mid p_1 p_2 \mid p_1 / p_2 \mid p* \mid !p$$

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

Hereafter, we present the fragment of a PEG for the Tiny language [11] to show how error reporting differs between top-down parsers without backtracking and PEGs. Tiny is a simple programming language with a syntax that resembles Pascal's.

$$\begin{aligned}
\mathit{Tiny} &\leftarrow \mathit{CmdSeq} \\
\mathit{CmdSeq} &\leftarrow (\mathit{Cmd}\ \texttt{SEMICOLON})\ (\mathit{Cmd}\ \texttt{SEMICOLON})* \\
\mathit{Cmd} &\leftarrow \mathit{IfCmd}\ /\ \mathit{RepeatCmd}\ /\ \mathit{AssignCmd}\ /\ \mathit{ReadCmd}\ /\ \mathit{WriteCmd} \\
\mathit{IfCmd} &\leftarrow \texttt{IF}\ \mathit{Exp}\ \texttt{THEN}\ \mathit{CmdSeq}\ (\texttt{ELSE}\ \mathit{CmdSeq}\ /\ \varepsilon)\ \texttt{END} \\
\mathit{RepeatCmd} &\leftarrow \texttt{REPEAT}\ \mathit{CmdSeq}\ \texttt{UNTIL}\ \mathit{Exp} \\
\mathit{AssignCmd} &\leftarrow \mathit{Name}\ \texttt{ASSIGNMENT}\ \mathit{Exp} \\
\mathit{ReadCmd} &\leftarrow \texttt{READ}\ \mathit{Name} \\
\mathit{WriteCmd} &\leftarrow \texttt{WRITE}\ \mathit{Exp}
\end{aligned}$$

PEGs usually express the language syntax down to the character level, without the need of a separate lexer. For instance, we can write the lexical rule `IF` as follows:

$$\texttt{IF} \leftarrow \mathit{if}\ !\mathit{IDRest}\ \mathit{Skip}$$

That is, the rule matches the keyword *if* provided that it is not a prefix of an identifier and then the rule skips surrounding white spaces and comments.

The non-terminal *IDRest* recognizes any character that may be present on a proper suffix of an identifier while the non-terminal *Skip* recognizes white spaces and comments. In the presented fragment, we omitted the lexical rules and the definitions of *Exp* and *Name* for brevity.

Now, we present an example of erroneous Tiny code to compare approaches for error reporting. The program has a missing semicolon (`;`) in the assignment in line `5`:

```
1 n := 5;
2 f := 1;
3 repeat
4   f := f * n;
5   n := n - 1
6 until (n < 1);
7 write f;
```

A hand-written top-down parser without backtracking that aborts on the first error presents an error message like this:

```
factorial.tiny:6:1: syntax error, unexpected 'until', expecting ';'
```

The error is reported in line `6` because the parser cannot complete a valid prefix of the language, since it unexpectedly finds the token `until` when it was expecting a command terminator (`;`).

In PEGs, we can try to report errors using the remaining suffix, but this approach usually does not help the PEG to produce an error message like the one shown above. In general, when a PEG finishes parsing the input, a remaining suffix that is not the empty string means that parsing did not reach the end of file due to a syntax error. However, this remaining suffix usually does not indicate the position where the longest parse ends. This problem happens because the failure of a parsing expression does not necessarily mean an error. Actually, the failure usually means that the PEG should backtrack the input to try a different alternative. For this reason, the remaining suffix probably indicates a position far away from the real position where the first error happened when parsing finishes without consuming all the input.

In our example, the problem happens when the PEG tries to recognize the sequence of commands inside the `repeat` command. Even though the program has a missing semicolon (`;`) in the assignment in line `5`, making the PEG fail to recognize the sequence of commands inside the `repeat` command, this failure is not treated as an error. Instead, this failure makes the recognition of the `repeat` command also fail. For this reason, the PEG backtracks the input to line `3` to try other command alternatives that exist in the language. Since it is not possible to recognize a command other than `repeat` in line `3`, the parsing finishes without consuming all the input. Hence, if the PEG uses the remaining suffix to produce an error message, the PEG shows a wrong position where the error happened.

We can also make the PEG fail whenever it does not consume all the input, instead of checking whether the remaining suffix is the empty string. To do that,

we change the starting symbol to fail when it does not reach the end of file. Even though the failure of the PEG indicates the presence of syntax errors, it does not indicate a possible position where the first error happened.

According to Ford [2], although there is no perfect method to identify which information is the most relevant to report an error, using the information of the farthest position that the PEG reached in the input is a heuristic that provides good results. PEGs implement top-down parsers and try to recognize the input from left to right, so the position farthest to the right in the input that a PEG reaches during parsing usually is close to the real error [2].

Ford used this heuristic to add error reporting to his packrat parsers [2]. A packrat parser generated by Pappy [3], Ford's PEG parser generator, tracks the farthest position and uses this position to report an error when parsing fails because it finished without consuming all the input. In other words, this heuristic helps packrat parsers to simulate the error reporting technique that is implemented in top-down parsers without backtracking.

During our research, we realized that we can use the farthest position heuristic to add error reporting to any implementation of PEGs that provides semantic actions. The idea is to annotate the grammar with semantic actions that track the farthest failure position. For instance, in Leg [16], a PEG parser generator with Yacc-style semantic actions, we can annotate the rule *CmdSeq* as follows:

```
CmdSeq = Cmd (";" Skip | &{ updateffp() })
         (Cmd (";" Skip | &{ updateffp() }))*
```

The parser calls the function `updateffp` when the matching of a semicolon fails. The function `updateffp` is a semantic action that updates the farthest failure position in a global variable if the current parsing position is greater than the position that is stored in this global. After the update, the semantic action forces another failure to not interrupt backtracking.

Since this semantic action propagates failures and runs only when a parsing expression fails, we could annotate all terminals and non-terminals in the grammar without changing the behavior of the PEG. In practice, we just need to annotate terminals to implement error reporting.

However, storing just the farthest failure position does not give the parser all the information it needs to produce an informative error message. That is, the parser has the information about the position where the error happened, but it lacks the information about what terminals failed at that position. Thus, we should include the name of the terminals in the annotations so the parser can also track these names to compute the set of expected terminals at a certain position.

Basically, we give an extra argument to each semantic action. This extra argument is a hard-coded name for the terminal that we want to keep track along with the farthest failure position. For instance, now we annotate the *CmdSeq* rule in Leg as follows:

```
CmdSeq = Cmd (";" Skip | &{ updateffp(";") })
         (Cmd (";" Skip | &{ updateffp(";") }))*
```

We then extend the implementation of `updateffp` to also update the set of expected terminals; the update of the farthest failure position continues the same. If the current position is greater than the farthest failure, the set contains only the given name. If the current position equals the farthest failure, the given name is added to the set.

Parsers generated by Pappy also track the set of expected terminals, but with limitations. The error messages include only symbols and keywords that were defined in the grammar as literal strings. That is, the error messages do not include terminals that were defined through character classes.

The approach of naming terminals in the semantic actions avoids the kind of limitation found in Pappy, though it increases the annotation burden because who is implementing the PEG is also responsible for adding one semantic action for each terminal and its respective name.

The annotation burden can be lessened in implementations of PEGs that treat parsing expressions as first-class objects, as we are able to define functions to annotate the lexical parts of the grammar to track errors, record information about the expected terminals to produce good error messages, and enforce lexical conventions such as the presence of surrounding white spaces. For instance, in LPeg [7, 8], a PEG library for Lua that defines patterns as first-class objects, we can annotate the rule *CmdSeq* as follows:

```
CmdSeq = V"Cmd" * symb(";") * (V"Cmd" * symb(";"))^0;
```

The function `symb` works like a parser combinator [6]. It receives a string as its only argument and returns a pattern that is equivalent to the parsing expression that we used in the Leg example. That is, `symb(";")` is equivalent to `";" Skip | &{ updateffp(";") }`.

We implemented error tracking and reporting using semantic actions as a set of parsing combinators on top of LPeg and used these combinators to implement the PEG for Tiny. It produces the following error message for the example we have been using in this section:

```
factorial.tiny:6:1: syntax error, unexpected 'until',
                    expecting ';', '=', '<', '-', '+', '/', '*'
```

We tested the PEG for Tiny with other erroneous inputs and in all cases the PEG identified an error in the same place as a top-down parser without backtracking. In addition, the PEG for Tiny produced error messages that are similar to the error messages produced by packrat parsers generated by Pappy. We annotated other grammars too and successfully obtained similar results. However, the error messages are still generic.

3 Error Reporting in Top-Down Parsers with Backtracking

In this section we discuss alternative approaches for error reporting in top-down parsers with backtracking other than the heuristic explained in Section 2.

Mizushima et al. [14] proposed a cut operator (↑) to reduce the space consumption of packrat parsers; the authors claimed that the cut operator can also be used to implement error reporting in packrat parsers, but the authors did not give any details on how the cut operator could be used for this purpose. The cut operator is borrowed from Prolog to annotate pieces of a PEG where backtracking should be avoided. PEGs' ordered choice works in a similar way to Prolog's green cuts, that is, they limit backtracking to discard unnecessary solutions. The cut proposed to PEGs is a way to implement Prolog's white cuts, that is, they prevent backtracking to rules that will certainly fail.

The semantics of cut is similar to the semantics of an `if-then-else` control structure and can be simulated through predicates. For instance, the PEG (with cut) $A \leftarrow B \uparrow C/D$ is functionally equivalent to the PEG (without cut) $A \leftarrow BC/!BD$ that is also functionally equivalent to the rule $A \leftarrow B[C, D]$ on Generalized Top-Down Parsing Language (GTDPL), one of the parsing techniques that influenced the creation of PEGs [2–4]. On the three cases, the expression D is tried only if the expression B fails. Nevertheless, this translated PEG still backtracks the input whenever B successfully matches and C fails. Thus, it is not trivial to use this translation to implement error reporting in PEGs.

Even though error handling is an important task for parsers, we did not find any other research about error handling in PEGs, beyond the heuristic proposed by Ford and the cut operator proposed by Mizushima et al. However, parser combinators [6] present some similarities with PEGs so we will briefly discuss them for the rest of this section.

In functional programming it is common to implement recursive descent parsers using parser combinators [6]. A parser is a function that we use to model symbols of the grammar. A parser combinator is a higher-order function that we use to implement grammar constructions such as sequencing and choice. Usually, we use parser combinators to implement parsers that return a list of results. That is, we use non-deterministic parser combinators that return a list of results to implement recursive descent parsers with full backtracking. We get parser combinators that have the same semantics as PEGs by changing the return type from list of results to `Maybe`. That is, we use deterministic parser combinators that return `Maybe` to implement recursive descent parsers with limited backtracking. In this paper we are referring to deterministic parser combinators.

Like PEGs, parser combinators also use ordered choice and try to accept input prefixes. More precisely, parsers implemented using parser combinators also backtrack the input in case of failure. For this reason, when the input string contains syntax errors, the longest parse usually indicates a position far away from the position where the error really happened.

Hutton [6] introduced the `nofail` combinator to implement error reporting in a quite simple way: we just need to distinguish between failure and error during parsing. More specifically, we can use the `nofail` combinator to annotate the grammar's terminals and non-terminals that should not fail; when they fail, the failure should be transformed into an error that aborts parsing. This technique

p_1	p_2	$p_1 p_2$	$p_1 \mid p_2$
`Error`	`Error`	`Error`	`Error`
`Error`	`Fail`	`Error`	`Error`
`Error`	`Epsn`	`Error`	`Error`
`Error`	`OK` (x)	`Error`	`Error`
`Fail`	`Error`	`Fail`	`Error`
`Fail`	`Fail`	`Fail`	`Fail`
`Fail`	`Epsn`	`Fail`	`Epsn`
`Fail`	`OK` (x)	`Fail`	`OK` (x)
`Epsn`	`Error`	`Error`	`Error`
`Epsn`	`Fail`	`Fail`	`Epsn`
`Epsn`	`Epsn`	`Epsn`	`Epsn`
`Epsn`	`OK` (x)	`OK` (x)	`OK` (x)
`OK` (x)	`Error`	`Error`	`OK` (x)
`OK` (x)	`Fail`	`Error`	`OK` (x)
`OK` (x)	`Epsn`	`OK` (x)	`OK` (x)
`OK` (x)	`OK` (y)	`OK` (y)	`OK` (x)

Table 1. Behavior of sequence and choice in the four-values technique

is also called the *three-values* technique because the parser finishes with one of the following values: `OK`, `Fail` or `Error`.

Röjemo [17] presented a `cut` combinator that we can also use to annotate the grammar pieces where parsing should be aborted on failure, on behalf of efficiency and error reporting. The `cut` combinator is different from the cut operator (↑) for PEGs because the combinator is abortive and unary while the operator is not abortive and nullary. The `cut` combinator introduced by Röjemo has the same semantics as the `nofail` combinator introduced by Hutton. However, the `cut` implementation uses an approach based on continuations while the `nofail` implementation uses an approach based on constructs.

Partridge and Wright [15] showed that error detection can be automated in parser combinators when we assume that the grammar is LL(1). Their main idea is: if one alternative successfully consumes at least one symbol, no other alternative can successfully consume any symbols. Their technique is also known as the *four-values* technique because the parser finishes with one of the following values: `Epsn`, when the parser finishes with success without consuming any input; `OK`, when the parser finishes with success consuming some input; `Fail`, when the parser fails without consuming any input; and `Error`, when the parser fails consuming some input. Three values were inspired by Hutton's work [6], but with new meanings.

In the four-values technique, we do not need to annotate the grammar because the authors changed the semantics of the sequence and choice combinators to automatically generate the `Error` value according to the table 1. In summary, the sequence combinator propagates an error when the second parse fails after consuming some input while the choice combinator does not try further alternatives if the current one consumed at least one symbol from the input. In case of error, the four-values technique detects the first symbol following the longest parse of the input and uses this symbol to report an error.

The four-values technique assumes that the input is composed by tokens which are provided by a separate lexer. However, being restricted to LL(1) gram-

mars can be a limitation because parser combinators, like PEGs, usually operate on strings of characters to implement both lexer and parser together. For instance, a parser for Tiny that is implemented with Parsec [10] does not parse the following program: `read x;`. That is, the matching of `read` against `repeat` generates an error. Such behavior is confirmed in table 1 by the third line from the bottom.

Parsec is a parser combinator library for Haskell that employs a technique equivalent to the four-values technique for implementing LL(1) predictive parsers that automatically report errors [10], so in this paper we refer to Parsec using the four-values technique. A predictive parser is a recursive descent parser without backtracking. Parsec inspired Ford on his heuristic that tracks the longest parse of the input to implement error reporting in packrat parsers and on the creation of a parser combinator library for Haskell to implement packrat parsers.

The authors of Parsec introduced the `try` combinator to avoid the LL(1) limitation found in the four-values technique. More precisely, we use `try` to annotate parts of the grammar where arbitrary lookahead is needed, though Parsec is a library for implementing LL(1) predictive parsers. Dual to the `nofail` combinator, the `try` combinator transforms an error into a failure. That is, the `try` combinator pretends that a parser `p` did not consume any input when `p` fails. For this reason, it should be used carefully because it breaks Parsec's automatic error detection system when it is overused.

Parsec's restriction to LL(1) grammars made it possible to implement in the library an error reporting technique similar to the one applied to top-down parsers. Parsec produces error messages that include the error position, the character at this position and the `FIRST` set of the productions that were expected at this position. Parsec also implements the error injection combinator (`<?>`) for naming productions. This combinator gets two arguments: a parser `p` and a string `exp`. The string `exp` replaces the `FIRST` set of a parser `p` when all the alternatives of `p` failed. This combinator is useful to name terminals and non-terminals to get better information about the context of a syntax error.

Swierstra and Duponcheel [18] showed an implementation of parser combinators for error recovery, although most libraries and parser generators that are based on parser combinators implement only error reporting. Their work shows an implementation of parser combinators that repair the input in case of error, produce an appropriated message, and continue parsing the rest of the input.

4 Labeled Failures for Error Reporting

Exceptions are a common mechanism for signaling and handling errors in programming languages. Exceptions let programmers classify the different errors their programs may signal by using distinct types for distinct errors, and decouple error handling from regular program logic.

In this section we add *labeled failures* to PEGs, a mechanism akin to exceptions and exception handling, with the goal of improving error reporting preserving PEGs composability. We also discuss how to use PEGs with labels to

implement some of the techniques that we have discussed in the previous section: the `nofail` combinator [6], the `cut` combinator [17], the four-values technique [15] and the `try` combinator [10].

A labeled PEG G is a tuple $(V, T, P, L, \texttt{fail}, p_S)$ where L is a finite set of labels and $\texttt{fail} \in L$. The other parts use the same definitions from Section 2. The abstract syntax of labeled parsing expressions adds the *throw* operator $\Uparrow^l$, which generates a failure with label l, and adds an extra argument S to the ordered choice operator, which is the set of labels that the ordered choice should catch. S must be a subset of L.

$$p = \varepsilon \mid a \mid A \mid p_1\, p_2 \mid p_1 /^S p_2 \mid p* \mid !p \mid \Uparrow^l$$

The semantics of PEGs with labels is defined by the relation $\overset{\text{PEG}}{\rightsquigarrow}$ among a parsing expression, an input string and a result. The result is either a string or a label. The notation $G[p]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ means that the expression p matches the input xy, consumes the prefix x and leaves the suffix y as the output. The notation $G[p]\ xy \overset{\text{PEG}}{\rightsquigarrow} l$ indicates that the matching of p fails with label l on the input xy.

Figure 1 presents the semantics of PEGs with labels using natural semantics [19]. Intuitively, ε successfully matches the empty string, not changing the input; a matches and consumes itself and fails with label `fail` otherwise; A tries to match the expression $P(A)$; p_1p_2 tries to match p_1, if p_1 matches an input prefix, then it tries to match p_2 with the suffix left by p_1, the label l is propagated otherwise; $p_1/^S p_2$ tries to match p_1 in the input and tries to match p_2 in the same input only if p_1 fails with a label $l \in S$, the label l is propagated otherwise; $p*$ repeatedly matches p until the matching of p silently fails with label `fail`, and propagates a label l when p fails with this label; $!p$ successfully matches if the input does not match p with the label `fail`, fails producing the label `fail` when the input matches p, and propagates a label l when p fails with this label, not consuming the input in all cases; $\Uparrow^l$ produces the label l.

We faced some design decisions in our formulation that are worth discussing.

We use `fail` as a label to maintain compatibility with the original semantics of PEGs. For the same reason, we define the expression p_1/p_2 as syntactic sugar for $p_1/^{\{\texttt{fail}\}}p_2$.

We use a set of labels in the ordered choice as a convenience. We could have each ordered choice handling a single label, and it would just lead to duplication: an expression $p_1\ /^{\{l_1,l_2,\ldots,l_n\}}\ p_2$ would become $(\ \ldots\ ((p_1\ /^{l_1}\ p_2)\ /^{l_2}\ p_2)\ \ldots\ /^{l_n}\ p_2)$.

The repetition stops silently only on the `fail` label to maintain the following identity: the expression $p*$ is equivalent to a fresh non-terminal A plus the rule $A \leftarrow p\ A\ /\ \varepsilon$.

The negative predicate succeeds only on the `fail` label to allow the implementation of the positive predicate: the expression $\&p$ that implements the positive predicate in the original semantics of PEGs [2–4] is equivalent to the expression $!!p$. Both expressions successfully match if the input matches p, fail producing the label `fail` when the input does not match p, and propagate a label l when p fails with this label, not consuming the input in all cases.

Empty $\dfrac{}{G[\varepsilon]\ x \overset{\text{PEG}}{\rightsquigarrow} x}\ (\textbf{empty.1})$

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

Non-terminal $\dfrac{G[P(A)]\ x \overset{\text{PEG}}{\rightsquigarrow} X}{G[A]\ x \overset{\text{PEG}}{\rightsquigarrow} X}\ (\textbf{var.1})$

Concatenation $\dfrac{G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y \quad G[p_2]\ y \overset{\text{PEG}}{\rightsquigarrow} X}{G[p_1\ p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} X}\ (\textbf{con.1})$ $\quad \dfrac{G[p_1]\ x \overset{\text{PEG}}{\rightsquigarrow} l}{G[p_1\ p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} l}\ (\textbf{con.2})$

Ordered Choice $\dfrac{G[p_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}{G[p_1\ /^S\ p_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}\ (\textbf{ord.1})$ $\quad \dfrac{G[p_1]\ x \overset{\text{PEG}}{\rightsquigarrow} l}{G[p_1\ /^S\ p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} l}, l \notin S\ (\textbf{ord.2})$

$$\frac{G[p_1]\ x \overset{\text{PEG}}{\rightsquigarrow} l \quad G[p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} X}{G[p_1\ /^S\ p_2]\ x \overset{\text{PEG}}{\rightsquigarrow} X}, l \in S\ (\textbf{ord.3})$$

Repetition $\dfrac{G[p]\ x \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}}{G[p*]\ x \overset{\text{PEG}}{\rightsquigarrow} x}\ (\textbf{rep.1})$ $\quad \dfrac{G[p]\ xyz \overset{\text{PEG}}{\rightsquigarrow} yz \quad G[p*]\ yz \overset{\text{PEG}}{\rightsquigarrow} z}{G[p*]\ xyz \overset{\text{PEG}}{\rightsquigarrow} z}\ (\textbf{rep.2})$

$$\frac{G[p]\ x \overset{\text{PEG}}{\rightsquigarrow} l}{G[p*]\ x \overset{\text{PEG}}{\rightsquigarrow} l}, l \neq \texttt{fail}\ (\textbf{rep.3})$$

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

$$\frac{G[p]\ x \overset{\text{PEG}}{\rightsquigarrow} l}{G[!p]\ x \overset{\text{PEG}}{\rightsquigarrow} l}, l \neq \texttt{fail}\ (\textbf{not.3})$$

Throw $\dfrac{}{G[\Uparrow^l]\ \overset{\text{PEG}}{\rightsquigarrow} l}\ (\textbf{throw.1})$

Fig. 1. Natural Semantics of PEGs with labels

Now, we use labeled failures to implement error reporting in the fragment of the Tiny grammar that we presented in Section 2. In the following example, the expression $[p]^l$ is syntactic sugar for $(p\ /\ \Uparrow^l)$. We use the expression $[p]^l$ to annotate the pieces of the PEG that should not fail and that should generate a label l to name the error and interrupt backtracking when they fail, saving the error position. That is, we use the `fail` label only for backtracking and other labels for tagging errors.

$$
\begin{aligned}
\mathit{Tiny} &\leftarrow \mathit{CmdSeq} \\
\mathit{CmdSeq} &\leftarrow (\mathit{Cmd}\ [\texttt{SEMICOLON}]^{\texttt{sc}})\ (\mathit{Cmd}\ [\texttt{SEMICOLON}]^{\texttt{sc}})* \\
\mathit{Cmd} &\leftarrow \mathit{IfCmd}\ /\ \mathit{RepeatCmd}\ /\ \mathit{AssignCmd}\ /\ \mathit{ReadCmd}\ /\ \mathit{WriteCmd} \\
\mathit{IfCmd} &\leftarrow \texttt{IF}\ [\mathit{Exp}]^{\texttt{eif}}\ [\texttt{THEN}]^{\texttt{then}}\ [\mathit{CmdSeq}]^{\texttt{cs1}}(\texttt{ELSE}\ [\mathit{CmdSeq}]^{\texttt{cs2}}/\varepsilon)\ [\texttt{END}]^{\texttt{end}} \\
\mathit{RepeatCmd} &\leftarrow \texttt{REPEAT}\ [\mathit{CmdSeq}]^{\texttt{csr}}\ [\texttt{UNTIL}]^{\texttt{until}}\ [\mathit{Exp}]^{\texttt{erep}} \\
\mathit{AssignCmd} &\leftarrow \mathit{Name}\ [\texttt{ASSIGNMENT}]^{\texttt{bind}}\ [\mathit{Exp}]^{\texttt{ebind}} \\
\mathit{ReadCmd} &\leftarrow \texttt{READ}\ [\mathit{Name}]^{\texttt{read}} \\
\mathit{WriteCmd} &\leftarrow \texttt{WRITE}\ [\mathit{Exp}]^{\texttt{write}}
\end{aligned}
$$

We use labeled failures to mark only the pieces of the PEG that should not fail. The PEG detects an error situation when parsing finishes with a certain label that was not caught, so it can identifies the error information that is tied to that certain label to report a more meaningful error message. For instance, if we use this PEG for Tiny to parse the example from Section 2, parsing finishes with the `sc` label and the PEG can use it to produce an error message like below:

```
factorial.tiny:6:1: syntax error, there is a missing ';'
```

Note how the semantics of the repetition works with the rule *CmdSeq*. Inside the repetition, the `fail` label means that there is no more commands to be matched and the repetition should stop while the `sc` label means that a semicolon (`;`) failed to match. It would not be possible to write the rule *CmdSeq* using repetition if we had chosen to stop the repetition with any label, instead of stopping only with the `fail` label, because the repetition would accept the `sc` label as the end of the repetition when it should propagate this label.

Like PEGs, parsers written using parser combinators also finish with success or failure and usually backtrack in case of failure, making it difficult to implement error reporting. In Section 3 we have briefly discussed some related work [6, 17, 15, 10] that solve this problem. Now, we will discuss how these techniques can be expressed using PEGs with labels.

In Hutton's deterministic parser combinators, the `nofail` combinator is used to distinguish between failure and error. We can express the `nofail` combinators using PEGs with labels as follows:

$$\texttt{nofail}\ p \ \equiv\ p\ /\ \Uparrow^{\texttt{error}}$$

That is, `nofail` is an expression that transforms the failure of p into an error to abort backtracking. Note that the `error` label should not be caught by any ordered choice. Instead, the ordered choice propagates this label and catches solely the `fail` label. The idea is that parsing should finish with one of the following values: success, `fail` or `error`.

The annotation of the Tiny grammar to use `nofail` is similar to the annotation we have done using labeled failures. Basically, we just need to change

the grammar to use `nofail` instead of $[p]^l$. For instance, we can write the rule *CmdSeq* as follows:

$$\mathit{CmdSeq} \leftarrow (\mathit{Cmd}\ (\texttt{nofail SEMICOLON}))\ (\mathit{Cmd}\ (\texttt{nofail SEMICOLON}))*$$

If we are writing a grammar from scratch, there is no advantage to use `nofail` instead of more specific labels, as the annotation burden is the same and with `nofail` we lose more specific error messages.

The `cut` combinator was introduced to reduce the space inefficiency of `nofail`, which is space inefficient when implemented in a lazy language due to the error propagation. The semantics of PEGs abstracts the implementation details that differentiate `cut` and `nofail`, thus, in PEGs they are expressed in the same way.

The four-values technique changed the semantics of parser combinators to implement predictive parsers for LL(1) grammars that automatically identify the longest input prefix in case of error, without needing annotations in the grammar. We can express this technique using labeled failures by transforming the original PEG with the following rules:

$$[\![\varepsilon]\!] \equiv \Uparrow^{\texttt{epsn}} \tag{1}$$

$$[\![a]\!] \equiv\ a \tag{2}$$

$$[\![A]\!] \equiv\ A \tag{3}$$

$$[\![p_1 p_2]\!] \equiv\ [\![p_1]\!]\ ([\![p_2]\!] \,/\, \Uparrow^{\texttt{error}}\ /^{\{\texttt{epsn}\}}\ \varepsilon)\ /^{\{\texttt{epsn}\}}\ [\![p_2]\!] \tag{4}$$

$$[\![p_1 / p_2]\!] \equiv\ [\![p_1]\!]\ /^{\{\texttt{epsn}\}}\ ([\![p_2]\!] \,/\, \Uparrow^{\texttt{epsn}})\ /\ [\![p_2]\!] \tag{5}$$

This translation is based on three labels, `epsn` means that the expression successfully finished without consuming any input, `fail` means that the expression failed without consuming any input, and `error` means that the expression failed after consuming some input. In our translation we do not have an `ok` label because a resulting suffix means that the expression successfully finished after consuming some input. It is straightforward to check that the translated expressions behave according to the table 1 from Section 3.

Parsec introduced the `try` combinator to annotate parts of the grammar where arbitrary lookahead is needed. We need arbitrary lookahead because PEGs and parser combinators usually operate on the character level. The authors of Parsec also showed a correspondence between the semantics of Parsec as implemented in their library and Partridge and Wright's four-valued combinators, so we can emulate the behavior of Parsec using labeled failures by building on the five rules above and adding the following rule for `try`:

$$[\![\texttt{try}\ p]\!] \equiv [\![p]\!]\ /^{\{\texttt{error}\}}\ \Uparrow^{\texttt{fail}} \tag{6}$$

If we take the Tiny grammar from Section 2, insert `try` in the necessary places, and pass this new grammar through the transformation $[\![\ ]\!]$, then we get a PEG that automatically identifies errors in the input with the `error` label. For instance, we can write the rule *RepeatCmd* as follows:

$$\mathit{RepeatCmd} \leftarrow (\texttt{try REPEAT})\ \mathit{CmdSeq}\ \texttt{UNTIL}\ \mathit{Exp}$$

5 Conclusions

In this paper we discussed error reporting in PEGs. Unfortunately, PEGs behave badly on the presence of syntax errors because backtracking usually makes the PEG report a position far away from the position where the error happened. Ford [2] showed how he changed his implementation of PEGs to add his farthest position heuristic to have error reporting in packrat parsers. We showed that we can use this heuristic without changing the implementation of PEGs, when it provides mechanisms to produce semantic actions. Although the farthest position heuristic helps PEGs to produce error messages that are close to the ones produced by predictive top-down parsers, these error messages are still generic.

The main contribution of this paper is the introduction of labeled failures to PEGs. The new approach closely resembles the technique of generating and handling exceptions. In this approach, the *throw* operator $\Uparrow^l$ throws labeled failures and the ordered choice catches these failures.

We introduced labeled failures to PEGs as a way to annotate error points in the grammar and tie them to more meaningful error messages. We showed that PEGs with labels report an error when parsing finishes with a label that was not caught. In practice, if we use labeled failures along with the heuristic proposed by Ford, PEGs give specific error messages that report the right place of the error. Furthermore, these error messages can be customized according to the labels that are being used. We also showed that our approach can express several techniques for error reporting on parser combinators as presented in related work [6, 17, 15, 10].

The grammar annotation demands care: if we mistakenly annotate expressions that should be able to fail, this actually modifies the behavior of the parser beyond error reporting. In any case, labeled PEGs introduce an annotation burden that is lesser than the annotation burden introduced by error productions in LR parsers, because error productions usually introduce *reduce-reduce* conflicts to the parser [9].

We implemented the semantics of PEGs with labels using Haskell as a prototype to help us testing our approach. We tested the annotation of Tiny and Lua grammars using this prototype. The tests succeeded in our goal of reporting errors in the correct places and with specific error messages. We also tested in our prototype the translations that we have presented in the previous section, and successfully obtained the expected results.

Finally, labeled failures also help to compose PEGs preserving specific error messages of each separate PEG. For instance, we can compose an annotated PEG that parses HTML with an annotated PEG that parses JavaScript, having specific error messages for each PEG. Composing two different PEGs is an interesting case study to be implemented. It would be also interesting to investigate other cases where exception handling may be useful in PEGs beyond error reporting.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2006)
2. Ford, B.: Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology (September 2002)
3. Ford, B.: Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In: ICFP 2002: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM (2002) 36–47
4. Ford, B.: Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In: POPL 2004: Proceedings of the 31st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, New York, USA, ACM (2004) 111–122
5. Grune, D., Jacobs, C.J.: Parsing Techniques: A Practical Guide. 2nd edn. Springer Publishing Company, Incorporated (2010)
6. Hutton, G.: Higher-Order Functions for Parsing. Journal of Functional Programming **2**(3) (jul 1992) 323–343
7. Ierusalimschy, R.: LPeg - Parsing Expression Grammars for Lua. `http://www.inf.puc-rio.br/~roberto/lpeg/lpeg.html` (2008) [Visited on March 2013].
8. Ierusalimschy, R.: A Text Pattern-Matching Tool based on Parsing Expression Grammars. Software - Practice & Experience **39**(3) (2009) 221–258
9. Jeffery, C.L.: Generating LR Syntax Error Messages from Examples. ACM Transactions on Programming Languages and Systems **25**(5) (2003) 631–640
10. Leijen, D., Meijer, E.: Parsec: Direct Style Monadic Parser Combinators For The Real World. Technical Report UU-CS-2001-35, Department of Computer Science, Utrecht University (2001)
11. Louden, K.C.: Compiler Construction: Principles and Practice. PWS Publishing Co., Boston, MA, USA (1997)
12. Medeiros, S., Mascarenhas, F., Ierusalimschy, R.: From Regular Expressions to Parsing Expression Grammars. In: Brazilian Symposium on Programming Languages. (2011)
13. Medeiros, S., Mascarenhas, F., Ierusalimschy, R.: Left Recursion in Parsing Expression Grammars. In: Brazilian Symposium on Programming Languages. (2012)
14. Mizushima, K., Maeda, A., Yamaguchi, Y.: Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space. In: PASTE 2010: Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, New York, NY, USA, ACM (2010) 29–36
15. Partridge, A., Wright, D.: Predictive parser combinators need four values to report errors. Journal of Functional Programming **6**(2) (1996) 355–364
16. Piumarta, I.: peg/leg — recursive-descent parser generators for C. `http://piumarta.com/software/peg/` (2007) [Visited on March 2013].
17. Röjemo, N.: Efficient Parsing Combinators. Technical report, Department of Computer Science, Chalmers University of Technology (1995)
18. Swierstra, S.D., Duponcheel, L.: Deterministic, Error-Correcting Combinator Parsers. In: Advanced Functional Programming. Volume 1129 of Lecture Notes in Computer Science., Springer (1996) 184–207
19. Winskel, G.: The Formal Semantics of Programming Languages: An Introduction. MIT Press, Cambridge, USA (1993)