Recognising and Generating Terms using Derivatives of Parsing Expression Grammars

Tony Garnock-Jones,[1] Alessandro Warth,[2] and Mahdi Eslamimehr [2]

[1]Northeastern University, Boston, Massachusetts, USA
[2]Communications Design Group, Los Angeles, California, USA

Abstract. Grammar-based sentence generation has been thoroughly explored for Context-Free Grammars (CFGs), but remains unsolved for recognition-based approaches such as Parsing Expression Grammars (PEGs). Lacking tool support, language designers using PEGs have difficulty predicting the behaviour of their parsers. In this paper, we extend the idea of *derivatives*, originally formulated for regular expressions, to PEGs. We then present a novel technique for sentence generation based on derivatives, applicable to any grammatical formalism for which the derivative can be defined—now including PEGs. Finally, we propose applying derivatives more generally to other problems facing language designers and implementers.

1 Introduction

Writing a grammar that accurately captures the syntax of a new language is challenging. No matter which formalism is used, seemingly correct grammars often have bugs, accepting some inputs they should not and rejecting others that they should. We would like to equip language designers and implementers with tools that enable them to build a precise understanding of the language that is accepted by a grammar. Techniques for generating example sentences from a grammar can provide a foundation for such a tool.

Parsing Expression Grammars (PEGs) [1] are recognition-based formal descriptions of language syntax. They go beyond the expressiveness of Context-Free Grammars (CFGs) by permitting unbounded lookahead, both positive and negative, and offering a prioritized choice construct corresponding to the recursive descent approach to alternation. These features enable PEGs to recognise languages—such as $a^n b^n c^n$—that CFGs cannot.

However, there is a price to pay: the interactions among these features can be subtle. This often yields surprising results. A method for generating example sentences following a PEG would help the programmer understand its implications by providing concrete examples. For example, in Ford's original paper on PEGs, he presents a grammar for the language $a^n b^n c^n$, demonstrating the power of the formalism. It was not until we used this grammar as a test case for our generation tool, and it produced the sentence *aaa*, that we realised Ford's grammar is not quite correct: it recognises sentences in a^+ in addition to the

intended $a^n b^n c^n$. Without the tool, it would not have occurred to us to try such sentences with the grammar. We were surprised by the flaw, showing that reading, implementing and using the grammar was not enough to build sufficient understanding of it. Note that this bug also eluded the inventor of the formalism, and the POPL reviewers.[1]

Generating sentences for CFGs has been thoroughly explored [2,3,4]; in fact, CFGs were originally proposed with sentence generation in mind. PEGs, however, correspond directly to *recursive descent* parsing techniques [5], and the potential for sentence generation using PEGs has not been investigated until now. Support for prioritized choice as well as positive and negative lookahead in PEGs introduces constraints on sentence generation that make it a more challenging task than the equivalent problem for CFGs. After failing to approach the problem directly from a recursive-descent perspective, we hit upon a technique based on the *method of derivatives*, used previously primarily for recognition [6,7,8]. The work presented in this paper builds on this idea, and makes the following technical contributions:

- A definition of derivatives for PEGs.
- A novel sentence generation technique based on derivatives that is applicable to any grammatical formalism for which the derivative can be defined.

Our general technique for sentence generation gives programmers much-needed help in debugging their regular, context-free, and parsing expression language specifications.

The rest of this paper is organized as follows. In section 2, we review the definition of PEGs and the method of recognising strings using derivatives of grammars. Section 3 describes our method of using derivatives for term generation. Section 4 presents and justifies our method of computing derivatives of PEGs. We touch on relevant implementation issues in section 5, and section 6 argues for the correctness of our definitions. We discuss related works in section 7, outline the potential for using derivatives in other ways to support language designers in section 8, and conclude the paper in section 9.

2 Background

2.1 Parsing Expression Grammars

A PEG is defined as a tuple $G = (V_N, V_T, R, e_S)$ of nonterminals V_N, terminals V_T, rules $R : V_N \mapsto e$, and a start expression e_S. Parsing expressions e are defined inductively as follows:

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

[1] See Ford 2004 section 3.4 for his $a^n b^n c^n$ grammar [1].

5. e_1/e_2, an alternation
6. e^*, zero-or-more repetitions
7. $!e$, a not-predicate
8. $\_$, a wildcard
9. $\emptyset$, a *failing* expression

To simplify our presentation of derivatives, we extend the core PEG expression variants (items 1 through 7 above) with the wildcard expression $\_$ and the failing expression $\emptyset$ (items 8 and 9). We omit positive lookahead $\&e$, since it can be defined as $\&e = !!e$ [1, §3.2].

A parsing expression conditionally matches a *prefix* of an input sequence drawn from $V_T^\star$. We define the semantics of PEGs with the function

$$(e, x) \Rightarrow o$$

where $x \in V_T^*$ and $o \in V_T^* \cup \{f\}$. (Note that $f \notin V_T$.) The "output" o of a successful match is the *suffix* of the input stream that was not consumed (recognized) by e, while an output of f indicates failure. In our definition of $\Rightarrow$ below, and in the remainder of the paper, we write a, b, c for terminals in V_T, A, B, C for nonterminals in V_N and x, y, z for (possibly-empty) strings in $V_T^\star$.

1. **Empty:** $(\epsilon, x) \Rightarrow x$
2. **Terminal (success case):** $(a, ax) \Rightarrow x$
3. **Terminal (failure case):** $(a, bx) \Rightarrow f$ if $a \neq b$
4. **Terminal (empty case):** $(a, \epsilon) \Rightarrow f$
5. **Nonterminal:** $(A, x) \Rightarrow o$ if $(R(A), x) \Rightarrow o$
6. **Sequence (progress case):** $(e_1\ e_2, xy) \Rightarrow o$ if $(e_1, xy) \Rightarrow y$ and $(e_2, y) \Rightarrow o$
7. **Sequence (failure case):** $(e_1\ e_2, x) \Rightarrow f$ if $(e_1, x) \Rightarrow f$
8. **Alternation (success case):** $(e_1/e_2, xy) \Rightarrow y$ if $(e_1, xy) \Rightarrow y$
9. **Alternation (fallback case):** $(e_1/e_2, x) \Rightarrow o$ if $(e_1, x) \Rightarrow f$ and $(e_2, x) \Rightarrow o$
10. **Zero-or-more repetitions (repetition case):** $(e^*, xy) \Rightarrow o$ if $(e, xy) \Rightarrow y$ and $(e^*, y) \Rightarrow o$
11. **Zero-or-more repetitions (termination case):** $(e^*, x) \Rightarrow x$ if $(e, x) \Rightarrow f$
12. **Not-predicate (success case):** $(!e, x) \Rightarrow x$ if $(e, x) \Rightarrow f$
13. **Not-predicate (failure case):** $(!e, x) \Rightarrow f$ if $(e, x) \Rightarrow o$ and $o \neq f$
14. **Wildcard (success case):** $(\_, ax) \Rightarrow x$
15. **Wildcard (empty case):** $(\_, \epsilon) \Rightarrow f$
16. **Failing:** $(\emptyset, x) \Rightarrow f$

Note that the expression e_1/e_2 denotes a *prioritized* choice: it is only when e_1 fails to match a given input that e_2 is given a chance. This is a key feature of PEGs, making them unambiguous, but is also one of the chief difficulties in reasoning about PEG behaviour while developing a grammar.

2.2 Recognising with derivatives

The method of derivatives, introduced for regular expressions (REs) in 1964 by Brzozowski [6], revisited in 2009 by Owens et al. [7], and extended to CFGs in

2011 by Might et al. [8], is a powerful, easily-understood and easily-implemented recognising technique.

The key idea is to define a *derivative* function that maps a grammar G and a terminal a to a derived grammar that should match the "rest of the input" after a has been consumed. Deciding whether a given input sequence is in the language of G is done by iterating over successive input tokens until the end of input is reached. At that point, the final derived grammar is examined. If its language includes the empty string, then the input is in the language of G; otherwise, it is not.

Formally, the derivative function D has type $V_T \times e \to e$. For REs, V_T is the input alphabet, and e is a regular expression; for CFGs, e is extended to permit *recursion* [8] and an environment mapping nonterminals to expressions is assumed; and for PEGs, V_T and e are to be understood as relating to a particular grammar G as defined in section 2.1.

We write $D_a e$ (the derivative of e with respect to a) for $a \in V_T$, and extend D inductively to sequences $x \in V_T^\star$ by

$$D_\epsilon e = e$$
$$D_{ax} e = D_x(D_a e)$$

In order to recognise using derivatives, we must be able to reliably determine whether the empty sequence ϵ is in the language of a particular expression, $L(e)$. To do this, we use a *nullability predicate*,[2]

$$\nu(e) \iff \epsilon \in L(e)$$

Deciding whether a non-empty sequence ax is in $L(e)$ is done by deciding whether x is in $L(D_a e)$. Summing up, and using the extension of D to sequences given above,

$$x \in L(e) \iff \nu(D_x e)$$

Both $\nu(e)$ and $D_a e$ are defined for PEGs below. The definition of $\nu(e)$ for PEGs includes an additional constraint not relevant to REs or CFGs: that, at the point of the nullability check, no part of e needs to perform any further lookahead in order to accept.

3 Generating sentences using derivatives

Equipped with a definition of the derivative of a grammar, we can now turn to its application in generating sentences within that grammar. Our generation method, the function *gen* in figure 1, is generic in the sense that it is valid for any grammatical formalism for which a nullability predicate and derivative function can be defined.

The only additional requirement is that some overapproximation to the *first set* [9] of a grammar can be computed. Any definition of the function *firsts* will

[2] Nullability is defined for REs by Brzozowski [6] and for CFGs by Might et al. [8]

$$
\begin{aligned}
\mathit{gen} &: e \to V_T^\star \cup \{f\} \\
\mathit{gen}\ e &= \begin{cases} \mathit{gen}'\ e\ \emptyset & \text{when enough output has been generated} \\ \mathit{gen}'\ e\ (\mathit{firsts}\ e) & \text{otherwise} \end{cases} \\
\\
\mathit{gen}' &: e \times \mathcal{P}(V_T) \to V_T^\star \cup \{f\} \\
\mathit{gen}'\ e\ (\{a\} \uplus t) &= \begin{cases} ax & \text{when } \mathit{gen}\ D_a e = x \\ \mathit{gen}'\ e\ t & \text{when } \mathit{gen}\ D_a e = f \end{cases} \\
\mathit{gen}'\ e\ \emptyset &= \begin{cases} \epsilon & \text{when } \nu(e) \\ f & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 1. Generic algorithm for generating sentences using derivative, nullability, and first sets.

work, so long as it really is an overapproximation of the actual first set of its argument. Even setting $\mathit{firsts}\ e = V_T$ would generate correct output, since gen' explores other alternatives if a selected terminal leads to failure of generation.

The algorithm *gen* operates by selecting some terminal a from the first set of a parsing expression e, and then recursing with the corresponding derivative $D_a e$, prepending a to the result. The algorithm can continue to randomly produce terminals as long as elements from the first set of e at each stage remain unexplored, and may yield the empty output sequence any time that e is nullable. Any criterion for placing an upper bound on the length of the output sequence may be used.

Our definition of *firsts* for PEGs (figure 2) avoids much inefficiency by overapproximating only in the case of negative lookahead, $!e$. Consider the expression $!(abc)\ (a/b/c)^\star$. The first set of that expression must include a, even though a is in the first set of the expression abc, because acb is accepted by the overall expression, even though abc is rejected. In general, we cannot use $\mathit{firsts}\ e$ when computing $\mathit{firsts}'\ !e\ t$ because e may examine more than a single token's worth of its input.

Because we overapproximate the first set of $!e$, and we treat *positive* lookahead $\&e$ as if it were $!!e$, we overapproximate the first set of $\&e$ as well. A practical implementation may choose to include $\&e$ as first-class syntax, defining $\mathit{firsts}'\ \&e\ t = \mathit{firsts}\ e \cap t$ in order to avoid unnecessary work in *gen*.

4 Derivatives of PEGs

Our definition of the derivative function for PEGs (figure 3) is closely modelled on Brzozowski's original definition for REs. The chief differences arise from the complications of *lookahead*, both positive and negative, and *prioritized choice*.

$$
\begin{aligned}
\mathit{firsts} &: e \to \mathcal{P}(V_T) \\
\mathit{firsts}\ e &= \mathit{firsts}'\ e\ \emptyset
\end{aligned}
$$

$$
\begin{aligned}
\mathit{firsts}' &: e \times \mathcal{P}(V_T) \to \mathcal{P}(V_T) \\
\mathit{firsts}'\ \epsilon\ t &= t \\
\mathit{firsts}'\ a\ t &= \{a\} \\
\mathit{firsts}'\ A\ t &= \mathit{firsts}'\ R(A)\ t \\
\mathit{firsts}'\ e_1\ e_2\ t &= \mathit{firsts}'\ e_1\ (\mathit{firsts}'\ e_2\ t) \\
\mathit{firsts}'\ e_1/e_2\ t &= \mathit{firsts}'\ e_1\ t \cup \mathit{firsts}'\ e_2\ t \\
\mathit{firsts}'\ e^\star\ t &= t \cup \mathit{firsts}'\ e\ t \\
\mathit{firsts}'\ !e\ t &= t \\
\mathit{firsts}'\ \_\ t &= V_T \\
\mathit{firsts}'\ \emptyset\ t &= \emptyset
\end{aligned}
$$

Fig. 2. Algorithm for (over)approximating the first set of a PEG.

4.1 Lookahead

In most implementations of PEGs, the expression $\&e_1\ e_2$ is evaluated by matching the input against e_1 and then, if the match succeeds, rewinding the input and matching it with e_2. Negative lookahead $!e_1\ e_2$ is evaluated similarly, rewinding and turning to e_2 only when matching against e_1 fails. This sequential processing is reflected in the use of the sequencing operator to compose a lookahead with another expression.

Derivatives, by contrast, have no notion of backtracking, rewinding, or sequential processing. Each input token is presented to the derivative function only once, and all possible alternative matches proceed essentially in *parallel*. The lookahead problem, then, becomes a special case of the awkward problem of sequencing in general.

Our treatment of sequencing follows the established pattern of using a nullability *combinator*, δ, to properly account for the non-commutativity of the sequencing operator. Brzozowski's rule for the derivative of the concatenation of two regular expressions R_1 and R_2 is

$$D_a(R_1\ R_2) = (D_a R_1)\ R_2\ +\ \delta R_1\ (D_a R_2)$$

The term δR_1 evaluates to ϵ if R_1 is nullable, and to $\emptyset$ otherwise. The intuition behind this rule is that if any string in $L(R_1)$ begins with a, then $D_a R_1 \neq \emptyset$ and so $D_a(R_1 R_2)$—the grammar for the "rest" of the input—must include all strings beginning with the "rest" of R_1 and continuing with the strings of $L(R_2)$. In addition, if $L(R_1)$ includes the empty string ($\delta R_1 = \epsilon$), then we must examine the strings in $L(R_2)$ to see if any begin with a. Otherwise, if $L(R_1)$ does not

$$
\begin{array}{rl}
D \;:\; V_T \times e \to e & \qquad \delta \;:\; V_T \times e \to e \\
D_a\epsilon = \emptyset & \qquad \delta_a\epsilon = \epsilon \\
D_a a = \epsilon & \qquad \delta_a b = \emptyset \\
D_a b = \emptyset \text{ when } a \neq b & \\
D_a A = D_a R(A) & \qquad \delta_a A = \delta_a R(A) \\
D_a(e_1\ e_2) = D_a e_1\ e_2\ /\ \delta_a e_1\ D_a e_2 & \qquad \delta_a(e_1\ e_2) = \delta_a e_1\ \delta_a e_2 \\
D_a(e_1/e_2) = D_a e_1\ /\ D_a e_2 & \qquad \delta_a(e_1/e_2) = \delta_a e_1\ /\ \delta_a !e_1\ \delta_a e_2 \\
D_a e^\star = D_a e\ e^\star\ /\ \delta_a e\ D_a e^\star & \qquad \delta_a e^\star = \delta_a e\ \delta_a e^\star\ /\ \delta_a !e\ \delta_a e^\star \\
D_a !e = \emptyset & \qquad \delta_a !e = !D_a(e\quad \_^*) \\
D_a \_ = \epsilon & \qquad \delta_a \_ = \emptyset \\
D_a \emptyset = \emptyset & \qquad \delta_a \emptyset = \emptyset
\end{array}
$$

Fig. 3. Derivative function D and the nullability combinator δ for PEGs.

include the empty string ($\delta R_1 = \emptyset$), then there is some unmatched requirement of R_1 that should prevent us from examining the strings in $L(R_2)$. In this case, the right-hand branch of the sum is effectively discarded.

This intuition extends to PEGs with minor modification for proper treatment of lookahead. In general, PEG sequences will be of the form

$$\&e_1\ ...\ \&e_n\ !e_{n+1}\ ...\ !e_m\ e_0$$

where we effectively expect each e_i's processing of the input string to proceed in parallel. The PEG version of the nullability combinator (figure 3) must not only check whether $\epsilon \in L(e_0)$, but *in the case that it is*, must also advance each of $e_1 ... e_m$, since their processing of the input should not be limited by the behaviour of e_0. That is, we must ensure that

$$D_a(\&e_1\ ...\ \&e_n\ !e_{n+1}\ ...\ !e_m\ e_0) = \&D_a e_1\ ...\ \&D_a e_n\ !D_a e_{n+1}\ ...\ !D_a e_m\ D_a e_0$$

since *all* of the $e_{i>0}$ must examine a in parallel with e_0.

Our nullability combinator therefore takes an additional argument, a, so that it can properly pass it on to D in cases involving lookahead. Our rule for computing the derivative of a sequence expression becomes

$$D_a(e_1\ e_2) = (D_a e_1)\ e_2\ /\ \delta_a e_1\ (D_a e_2)$$

and in the specific case of $!e_1\ e_2$,

$$
\begin{aligned}
D_a(!e_1\ e_2) &= (D_a !e_1)\ e_2\ /\ \delta_a !e_1\ (D_a e_2) \\
&= \emptyset\ e_2\ /\ !(D_a e_1)\ (D_a e_2) \\
&= \emptyset\ /\ !(D_a e_1)\ (D_a e_2) \\
&= !(D_a e_1)\ (D_a e_2) \qquad (1)
\end{aligned}
$$

as desired.[3] The term $\delta_a !e_1$ evaluates to $!(D_a e_1)$ on the right-hand side of the alternation. The left-hand side of the alternation is discarded entirely, because $D_a !e_1 = \emptyset$ for any e_1.

Example 1. Consider the PEG

$$P \leftarrow \&abc \quad \_ \_ \_$$

After the input a, the "rest" of the grammar is $D_a P$:

$$\begin{aligned}
D_a P &= D_a(\&abc \quad _ _ _) \\
&= D_a(\&abc) \quad _ _ _ \quad / \quad \delta_a(\&abc) \quad D_a(_ _ _) \\
&= \emptyset \quad _ _ _ \quad / \quad \&(D_a(abc \quad _^*)) \quad D_a(_ _ _) \\
&= \emptyset \quad / \quad \&(bc \quad _^*) \quad _ _ \\
&= \&(bc \quad _^*) \quad _ _ \\
&= \&bc \quad _ _
\end{aligned}$$

Notice that the left-hand branch of the alternation is always pruned when computing the derivative of a sequence where the first element is a lookahead. If it were not so, the derived grammar would *only* present the lookahead with the next input character, and other branches of the multiple parallel ongoing parses would be incorrectly stalled. Notice also that δ "steps" the lookahead in the right-hand branch of the alternation in parallel with the "stepping" of the remainder of the grammar.

Property 1. Consider now the general case of $(!e_1\ e_2)\ e_3$. Computing the derivative must respect the associativity of sequencing, i.e.,

$$D_a((!e_1\ e_2)\ e_3) = D_a(!e_1\ (e_2\ e_3))$$

Proof.

$$\begin{aligned}
D_a((!e_1\ e_2)\ e_3) &= D_a(!e_1\ e_2)\ e_3\ /\ \delta_a(!e_1\ e_2)\ D_a e_3 \\
&= !(D_a e_1)\ (D_a e_2)\ e_3\ /\ \delta_a(!e_1\ e_2)\ D_a e_3 \quad \text{(by (1))} \\
&= !(D_a e_1)\ (D_a e_2)\ e_3\ /\ \delta_a !e_1\ \delta_a e_2\ D_a e_3 \\
&= !(D_a e_1)\ (D_a e_2)\ e_3\ /\ !(D_a e_1)\ \delta_a e_2\ D_a e_3 \\
&= !(D_a e_1)\ ((D_a e_2)\ e_3\ /\ \delta_a e_2\ D_a e_3) \quad \text{(by left-factoring)} \\
&= !(D_a e_1)\ D_a(e_2\ e_3) \\
&= D_a(!e_1\ (e_2\ e_3))
\end{aligned}$$

[3] While working through examples such as this, we will regularly be simplifying as we go using the identities discussed in section 5.

4.2 Prioritized choice

Lookahead is not the only complication in defining the derivatives of PEGs. Instead of the unbiased sum operator of CFGs, PEGs make use of a *prioritized choice* operator. When evaluating a choice e_1/e_2, if e_1 accepts (a prefix of) the input, then e_2 is not even visited by the traditional PEG algorithm. Our definition of $D_a(e_1/e_2)$ must reflect this bias. We approach the problem by observing that

$$e_1/e_2 \equiv e_1/!(e_1)e_2, \tag{2}$$

which makes clear that any inputs accepted by e_1 prevent the right-hand branch of the alternation from accepting. Further, if alternation in PEGs *were* unbiased, then both D_a and δ_a would simply distribute across alternation, just as they do for alternation in REs and CFGs. Hypothetically, then:

$$\begin{aligned} D_a(e_1/e_2) &\stackrel{?}{=} D_a e_1/D_a e_2 \\ \delta_a(e_1/e_2) &\stackrel{?}{=} \delta_a e_1/\delta_a e_2 \end{aligned} \tag{3}$$

These definitions make a reasonable starting point in our search for a means of accommodating prioritized choice. In the case of D_a, the intuition is that the input terminal a could relate equally to both possible alternatives. In the case of δ_a, which is used only in our rule for sequence expressions, we must evaluate to a failing expression only if *both* $\delta_a e_1$ and $\delta_a e_2$ evaluate to a failing expression; otherwise, we might prune the derived grammar inappropriately.

Applying (2) to these starting points, we learn that the bias in PEG alternation does not affect the definition of D:

$$\begin{aligned} D_a(e_1/e_2) &\equiv D_a(e_1/!(e_1)e_2) \text{ (by (2))} \\ &\stackrel{?}{=} D_a e_1/D_a(!(e_1)e_2) \\ &= D_a e_1/(D_a!e_1\ e_2\ /\ \delta_a!e_1\ D_a e_2) \\ &= D_a e_1/(\emptyset\ e_2\ /\ !D_a e_1\ D_a e_2) \\ &= D_a e_1/!D_a e_1\ D_a e_2 \\ &\equiv D_a e_1/D_a e_2 \text{ (by (2))} \end{aligned}$$

but that the definition of δ must take the bias into account:

$$\begin{aligned} \delta_a(e_1/e_2) &\equiv \delta_a(e_1/!(e_1)e_2) \\ &\stackrel{?}{=} \delta_a e_1/\delta_a(!(e_1)e_2) \\ &= \delta_a e_1/\delta_a!e_1\ \delta_a e_2 \\ &= \delta_a e_1/!D_a e_1\ \delta_a e_2 \\ &\not\equiv \delta_a e_1/\delta_a e_2 \end{aligned}$$

We therefore define

$$\delta_a(e_1/e_2) = \delta_a e_1\ /\ \delta_a!e_1\ \delta_a e_2$$

not only to follow our intuition, outlined above, but also to ensure that the branches are *disjoint* [1, §3.7] so that e_2 only influences the result when e_1 has certainly failed.

Example 2. Consider the PEG

$$Q \leftarrow \&((a/\epsilon)\ !b)\ \_^\star$$

If we attempt to compute D_aQ using our hypothetical definition (3), we get:

$$\begin{aligned}
D_aQ &\leftarrow D_a(\&((a/\epsilon)\ !b)\ _^\star) \\
&\quad \{\text{apply sequence rule for } D\} \\
&= D_a\&((a/\epsilon)\ !b)\ _^\star \,/\, (\delta_a\&((a/\epsilon)\ !b))\ D_a(_^\star) \\
&\quad \{\text{apply } D_a\&e = \emptyset\} \\
&= \emptyset\ _^\star \,/\, (\delta_a\&((a/\epsilon)\ !b))\ D_a(_^\star) \\
&\quad \{\text{apply } \delta_a\&e = \&D_a(e\ _^\star) \text{ and } \emptyset\ e = \emptyset \text{ and } D_a_^\star = _^\star\} \\
&= \emptyset \,/\, \&(D_a(((a/\epsilon)\ !b)\ _^\star))\ _^\star \\
&\quad \{\text{apply sequence rule for } D\} \\
&= \emptyset \,/\, \&(D_a((a/\epsilon)\ !b)\ _^\star \,/\, \delta_a((a/\epsilon)\ !b)\ D_a(_^\star))\ _^\star \\
&\quad \{\text{apply sequence rule for } D \text{ and } \emptyset/e = e\} \\
&= \&((D_a(a/\epsilon)\ !b \,/\, \delta_a(a/\epsilon)\ D_a!b)\ _^\star \,/\, \delta_a((a/\epsilon)\ !b)\ D_a(_^\star))\ _^\star \\
&\quad \{\text{apply choice rule for } D \text{ and sequence rule for } \delta \text{ and } D_a_^\star = _^\star\} \\
&= \&(((D_aa/D_a\epsilon)\ !b \,/\, \delta_a(a/\epsilon)\ D_a!b)\ _^\star \,/\, (\delta_a(a/\epsilon)\ \delta_a!b)\ _^\star)\ _^\star \\
&\quad \{\text{apply base cases for } D\} \\
&= \&(((\epsilon/\emptyset)\ !b \,/\, \delta_a(a/\epsilon)\ \emptyset)\ _^\star \,/\, (\delta_a(a/\epsilon)\ \delta_a!b)\ _^\star)\ _^\star \\
&\quad \{\text{simplify with } e/\emptyset = e,\ \epsilon\ e = e,\ e\ \emptyset = \emptyset\} \\
&= \&(!b\ _^\star \,/\, (\delta_a(a/\epsilon)\ \delta_a!b)\ _^\star)\ _^\star \\
&\quad \{\text{apply } \delta_a!b = !D_a(b\ _^\star) = \cdots = !(\emptyset\ _^\star) = !\emptyset = \epsilon\} \\
&= \&(!b\ _^\star \,/\, (\delta_a(a/\epsilon)\ \epsilon)\ _^\star)\ _^\star \qquad (4) \\
&\quad \{\text{apply equation (3)}\} \\
&= \&(!b\ _^\star \,/\, ((\delta_aa/\delta_a\epsilon)\ \epsilon)\ _^\star)\ _^\star \\
&\quad \{\text{apply base cases for } \delta\} \\
&= \&(!b\ _^\star \,/\, ((\emptyset/\epsilon)\ \epsilon)\ _^\star)\ _^\star \\
&\quad \{\text{simplify with } \emptyset/\epsilon = \epsilon \text{ and } \epsilon\ \epsilon = \epsilon\} \\
&= \&(!b\ _^\star \,/\, \epsilon\ _^\star)\ _^\star \\
&\quad \{\text{simplify with } e_1_^\star/e_2_^\star = (e_1/e_2)_^\star \text{ and } \&(e\ _^\star) = \&e\} \\
&= \&(!b \,/\, \epsilon)\ _^\star
\end{aligned}$$

This is incorrect, however. Since $!b/\epsilon$ is equivalent to ϵ, the faulty D_aQ above is equivalent to $\_^\star$. The constraint preventing a b terminal from appearing next has been lost.

If instead we use δ as defined in figure 3 to compute D_aQ, then repeating our calculation from the point of difference (marked (4) above), we see that

$$
\begin{aligned}
D_aQ \leftarrow\ & D_a(\&((a/\epsilon)\ !b)\ \_^\star) \\
& \cdots \\
=\ & \&(!b\ \_^\star\ /\ (\delta_a(a/\epsilon)\ \epsilon)\ \_^\star)\ \_^\star \\
& \{\text{apply correct choice rule for } \delta \text{ from figure 3}\} \\
=\ & \&(!b\ \_^\star\ /\ ((\delta_a a\ /\ \delta_a !a\ \delta_a\epsilon)\ \epsilon)\ \_^\star)\ \_^\star \\
& \{\text{apply base cases for } \delta\} \\
=\ & \&(!b\ \_^\star\ /\ ((\emptyset\ /\ \delta_a !a\ \epsilon)\ \epsilon)\ \_^\star)\ \_^\star \\
& \{\text{apply } \delta_a !a = !D_a(a\ \_^\star) = \cdots = !(\epsilon\ \_^\star) = !(\_^\star) = \emptyset\} \\
=\ & \&(!b\ \_^\star\ /\ ((\emptyset\ /\ \emptyset\ \epsilon)\ \epsilon)\ \_^\star)\ \_^\star \\
& \{\text{simplify with } \emptyset/e = e \text{ and } \emptyset\ e = \emptyset\} \\
=\ & \&(!b\ \_^\star\ /\ \emptyset)\ \_^\star \\
& \{\text{simplify with } e/\emptyset = e \text{ and } \&(e\ \_^\star) = \&e \text{ and } \&!e = !e\} \\
=\ & !b\ \_^\star
\end{aligned}
$$

which correctly rejects the input if the next token is b.

4.3 Nullability and Well-formedness

The algorithms for recognising and for generating sentences using derivatives (sections 2.2 and 3, respectively) need the nullability predicate ν to decide when to stop. A straightforward extension of Brzozowski's definition of ν suffices for *well-formed* PEGs, i.e., grammars that do not contain left-recursive rules [1]. Figure 4 defines the nullability and well-formedness (WF) predicates for PEGs.

For a well-formed PEG e, nullability $\nu(e)$ means that $(e, \epsilon) \Rightarrow \epsilon$, and vice versa. In other words, a PEG is considered nullable when, given the empty input, the decision can be made to accept immediately, without need to examine further input to decide questions posed by use of lookahead. In Ford's terms, this is equivalent to a successful parse when presented with the empty input sequence.

In order to compute ν and WF, we must use Kleene's fixed-point theorem, choosing $\perp$ as the least element and iterating to stability, joining iterations with $\vee$. For example, the left-recursive definition $X \leftarrow Xx/\epsilon$ is nullable, but not

$$
\begin{array}{rl}
\nu : e \to 2 & \qquad WF : e \to 2 \\
\nu(\epsilon) = \top & \qquad WF(\epsilon) = \top \\
\nu(a) = \bot & \qquad WF(a) = \top \\
\nu(A) = \nu(R(A)) & \qquad WF(A) = WF(R(A)) \\
\nu(e_1 e_2) = \nu(e_1) \wedge \nu(e_2) & \qquad WF(e_1 e_2) = WF(e_1) \wedge (\nu(e_1) \implies WF(e_2)) \\
\nu(e_1 / e_2) = \nu(e_1) \vee \nu(e_2) & \qquad WF(e_1 / e_2) = WF(e_1) \wedge WF(e_2) \\
\nu(e^\star) = \top & \qquad WF(e^\star) = WF(e) \wedge \neg\nu(e) \\
\nu(!e) = \neg\nu(e) & \qquad WF(!e) = WF(e) \\
\nu(\_) = \bot & \qquad WF(\_) = \top \\
\nu(\emptyset) = \bot & \qquad WF(\emptyset) = \top
\end{array}
$$

Fig. 4. Nullability predicate $\nu(e)$ and well-formedness predicate $WF(e)$.

well-formed:

$$
\begin{aligned}
\nu(X) = \nu(R(X)) &= \nu(Xx/\epsilon) \\
&= \nu(Xx) \vee \nu(\epsilon) \\
&= \nu(Xx) \vee \top = \top
\end{aligned}
$$

$$
\begin{aligned}
WF(X) = WF(R(X)) &= WF(Xx/\epsilon) \\
&= WF(Xx) \wedge WF(\epsilon) \\
&= WF(X) \wedge (\nu(X) \implies WF(x)) \\
&= WF(X) \wedge (\top \implies \top) \\
&= WF(X) = \bot \text{ (least element)}
\end{aligned}
$$

Its right-recursive variation $X' \leftarrow xX'/\epsilon$, however, is both nullable and well-formed:

$$
\begin{aligned}
WF(X') = WF(R(X')) &= WF(xX'/\epsilon) \\
&= WF(xX') \wedge WF(\epsilon) \\
&= WF(x) \wedge (\nu(x) \implies WF(X')) \\
&= \top \wedge (\bot \implies \bot) \\
&= \top \wedge \top = \top
\end{aligned}
$$

Example 3. Consider the PEG

$$P \leftarrow \&c \;\; cc$$

The input cc is accepted if $\nu(D_{cc}P)$, that is, if $\nu(D_c(D_cP))$:

$$\begin{aligned}
\nu(D_c(D_cP)) &= \nu(D_c(D_c(\&c \quad cc))) \\
&= \nu(D_c(\&(\epsilon \quad _^*) \quad c)) \\
&\vdots \\
&= \nu(\&(_^* \quad _^*) \quad \epsilon) \\
&= \nu(\&(_^* \quad _^*)) \wedge \nu(\epsilon) \\
&= \nu(_^* \quad _^*) \wedge \top \\
&= \nu(_^*) \wedge \nu(_^*) \\
&= \top
\end{aligned}$$

5 Implementing derivatives for PEGs

As of this writing, we have two implementations of the algorithms described in this paper: one in JavaScript, and another in Racket [10]. In both cases, as reported by Might et al. [8], we found that careful attention to *memoization*, *laziness*, and *fixed-point calculations* was required. In order to avoid the accumulation of unimportant structure while computing derivatives (which would make our implementations impractical) it was also necessary to apply a set of grammar identities to simplify the expressions resulting from functions D and δ.

Memoization is important for both D and δ to avoid redundant computation and space explosion. Furthermore, because PEGs as defined here form a general directed graph, rather than a DAG or tree, our implementations introduce laziness at nonterminals A by memoizing *promises* in order to break cycles in the graph of the grammar. The same approach was used by Bracha (via instances of `ForwardReferenceParser`) in his implementation of executable grammars in Newspeak [11].

Our Racket implementation makes use of the memoization machinery for Racket made available by Might et al. [8], but we were unable to use the associated implementation of iteration to fixed point when implementing ν, WF, and *firsts*. Our definition of ν includes logical negation (in the clause for negative lookahead) and so our fixed point algorithm must take into account the join operator of the Boolean lattice being used, on pain of non-termination in (ill-formed) cases such as computing $\nu(A)$ when $A \leftarrow !A$. Might's implementation does not support the specification of a join operator, instead simply updating variables after each iteration. This fails not only for ν with ill-formed PEGs but also for *firsts* even with well-formed PEGs: the first set for a nonterminal must be grown iteratively using a lattice with least element $\emptyset$ and join operator $\cup$.

Implementations of the method of derivatives generally make heavy use of identities to avoid constructing and processing needless structure. For example, Brzozowski reports that without use of an equational theory for REs, a given RE may have an unbounded number of distinct derivatives, but with a simple set of identities, the number of derivatives of an RE is finite [6]. Owens et al. enrich

$$
\begin{array}{llll}
\&\epsilon \to \epsilon & !\epsilon \to \emptyset & \epsilon\ e \to e & \emptyset\ /\ e \to e \\
\&(\_^\star) \to \epsilon & !(\_^\star) \to \emptyset & e\ \epsilon \to e & e\ /\ \emptyset \to e \\
\&\emptyset \to \emptyset & !\emptyset \to \epsilon & \emptyset\ e \to \emptyset & e_1\ /\ !e_1\ e_2 \to e_1\ /\ e_2 \\
\&(e\ \_^\star) \to \&e & !(e\ \_^\star) \to !e & e\ \emptyset \to \emptyset & e_1\ /\ !e_1 \to e_1\ /\ \epsilon \\
\&\&e \to \&e & !\&e \to !e & \_^\star\ \_^\star \to \_^\star & e\ /\ e \to e \\
\&!e \to !e & !!e \to \&e & !e_1\ !e_1\ e_2 \to !e_1\ e_2 & e_1\ e_2\ /\ e_1\ e_3 \to e_1\ (e_2\ /\ e_3) \\
 & & \&e_1\ \&e_1\ e_2 \to \&e_1\ e_2 & e_1\ \_^\star\ /\ e_2\ \_^\star \to (e_1\ /\ e_2)\ \_^\star
\end{array}
$$

Fig. 5. Simplifications applied when constructing PEG terms.

the equational theory in order to more closely approach a minimal DFA when using derivatives to compile REs [7]. Finally, Might et al. also employ identities in their *compaction* of CFGs during recognition, in between uses of D [8]. In our implementations, as in the implementation of Owens et al., most of the identity-based simplifications are placed in *smart constructors* for our PEG data type. Figure 5 lists the identities respected by our implementations, including many from previous work on derivatives as well as from Ford's original work on PEGs [1, §3.7].

6 Evaluation

Our experience with using derivatives of PEGs to recognise and generate sentences has led us to believe that our definition is correct, not only with respect to the definition of derivatives originating with Brzozowski [6, definition 3.1],

> Given a set R of sequences and a finite sequence s, the *derivative of* R *with respect to* s is denoted by $D_s R$ and is $D_s R = \{t \mid st \in R\}$

but also with respect to Ford's original semantics for PEGs [1]. That is to say, we believe that the following conjectures hold, and our experience has to date supported this belief.

Conjecture 1. For all well-formed expressions e and terminals $a \in V_T$, $WF(D_a e)$.

Well-formedness is preserved by derivation.

Conjecture 2. For all well-formed expressions e and strings $x \in V_T^\star$, $(e\ \_^\star, x) \Rightarrow \epsilon$ iff $\nu(D_x(e\ \_^\star))$.

PEGs are defined by Ford in terms of matching a *prefix* of their inputs, but recognising via derivatives is defined in terms of the *entire* input. Consideration of $(e\ \_^\star)$ ensures that both definitions line up, either consuming the entirety of the input or failing.

Conjecture 3. For all well-formed expressions e, terminals $a \in V_T$, and strings $x, y \in V_T^\star$, $(e, axy) \Rightarrow y$ iff $(D_a e, xy) \Rightarrow y$.

This connects Brzozowski's definition of derivatives with Ford's semantics. If a PEG e can accept input axy yielding input suffix y, then e "after the terminal a", $D_a e$, should accept input xy yielding input suffix y, and vice versa.

Conjecture 4. For all well-formed expressions e and strings $x \in V_T^\star$, $gen(e)$ can produce x iff $(e, x) \Rightarrow \epsilon$.

Any string produced by our *gen* algorithm is recognisable by e, and any string recognisable by e may be produced by *gen*.

Conjecture 5. For all well-formed expressions e, $\{a \mid ax \in L(e)\} \subseteq \mathit{firsts}(e)$.

Our *firsts* function (figure 2) must be complete with respect to the actual language of its argument in order for *gen* to work correctly. It may, of course, be larger than it needs to be, at the cost of some inefficiency in *gen*.

7 Related Work

The notion of derivatives in grammatical formalisms is not new: Brzozowski first introduced it in 1964, in the context of regular expressions [6]. This work was later extended by Owens et al. to generate efficient DFA-based recognisers for REs [7], and by Might et al. to generate parsers for CFGs. 50 years after Brzozowski's original contribution, we extend the notion of derivatives to PEGs. Independently, Moss has tackled this problem, taking quite a different approach that augments the syntax of PEGs with the ongoing state of a parse [12]. By constrast, our approach does not require any new syntax, and shows that PEGs are closed under derivative.

Applications of Brzozowski's derivatives are not limited to recognizing and parsing. For example, they have also been used to compile Esterel programs to automata [13], to generate DFAs from regular expressions that are used to monitor program traces [14], and to prove the totality of parser combinators [15]. In this paper, we present a new application of derivatives to sentence generation.

While there are many different techniques for generating sentences from CFGs (e.g., random generation [16,17,18,19,4], exhaustive enumeration [20,21,3,2], and coverage-based generation [22,23]), the problem of generating sentences from PEGs has received very little attention until now. To the best of our knowledge, Petit Parser [24] is the only other system that can generate random sentences from a PEG. However, Petit Parser's sentence generator ignores both positive and negative lookahead expressions, which can lead to incorrect results. This is a serious limitation because in addition to being the source of much of the power of the PEG formalism[4], positive and negative lookahead expressions are frequently used in practical grammars to (for example) correctly recognize keywords:

$$\mathit{While} \leftarrow \mathit{'while'}\ !\mathit{Alnum}$$
$$\mathit{Return} \leftarrow \mathit{'return'}\ !\mathit{Alnum}$$

[4] For example, the $a^n b^n c^n$ grammar discussed in section 1 makes essential use of lookahead [1, §3.4].

The $!Alnum$ expressions, if $R(Alnum)$ matches a single alphanumeric character, prevent the rules from matching identifiers that contain their respective keyword as a prefix, e.g., *while01* and *returnedItems*. By combining our definition of derivatives for PEGs with our generic derivative-based sentence generation technique, we are able to correctly generate sentences from any PEG, including those that use positive and negative lookahead.

8 Applications and Future work

The ability to generate sentences from PEGs could give language designers and implementers powerful new ways to write and debug grammars. For example, we are currently working on an IDE for a PEG-based parser generator that will show examples of sentences in the language defined, even as the programmer edits the grammar. Changes in the grammar will be reflected immediately in the set of examples displayed, and the programmer will be able to select some or all of the randomly-generated examples to make them into unit tests that are run after each change. That way, if a later edit results in one of those examples being rejected, the programmer will know right away. We expect this kind of interface will help programmers better understand their grammars, and enable them to catch bugs earlier than with traditional tools.

Sentence generation could also be used to produce *probabilistic proofs* of PEG identities, which can be difficult to reason about. For example, in order to determine whether

$$e_1\ {e_2}^*\ /\ e_3\ {e_2}^* \equiv (e_1\ /\ e_3)\ {e_2}^*$$

one might use a PEG grammar describing parsing expressions themselves to generate random values for e_1, e_2, and e_3, then substitute those values into both of the expressions above, and generate a large number of sentences from each of them. If all sentences generated by the left hand side of the potential identity are matched by the expression on right hand side and vice-versa, there is a high probability that they are equivalent.

While the initial motivation of the work presented in this paper was sentence generation, our definition of derivative in section 4 combines with the algorithm of section 2.2 to build a recogniser for PEGs. It would be interesting to go a step further in this direction and adapt Might et al.'s technique for building derivatives-based parser combinators for CFGs [8] to PEGs.

Researchers have proposed extensions to the semantics of PEGs that allow grammars to include left-recursive rules [25,26]. Without these extensions, an application of a rule that is left-recursive will always result in infinite recursion, just as it would in a recursive-descent parser. But the semantics of left recursion in PEGs remains a topic of debate. For example, it is not at all clear whether a rule that mixes left- and right-recursive applications should produce parse trees that are left- or right-associative [27]. We believe the notion of derivatives may help settle this debate. When parsing with derivatives, choices are evaluated in parallel—i.e., the derivative of a choice expression "makes progress" on both of

its branches—so it may be possible to use the foundations laid out in this paper to find a "natural" semantics for left-recursive PEGs.

Another extension of PEGs synergistic with our sentence generation technique is OMeta [28]. OMeta extends PEGs with an expressive form of pattern matching that works on structured data (objects, arrays, etc.) as well as strings. By extending our notion of derivative to OMeta, we should be able to use grammars to generate random objects for testing purposes, a la QuickCheck [29].

PEG sentence generation has other applications in testing. For example, it should now be possible to build PEG-based equivalents of popular grammar-based testing tools such as CESE [3] and SAGE [2], both of which are based on CFGs. Our random sentence generation technique could also be used to ensure (probabilistically) that a compiler can handle all possible nestings / interactions among the constructs that are available in a dynamic programming language. (Doing the same for the compiler of a statically-typed language would require generating well-typed programs, which is still an open problem.)

9 Conclusion

In this paper, we have adapted Brzozowski's notion of derivatives to PEGs, and introduced a new sentence generation technique based on derivatives that is applicable to any grammatical formalism for which the derivative can be defined, which now includes PEGs. We have also outlined some ways in which sentence generation can improve tool support for language designers and implementers who use PEGs to define the syntax of their languages.

References

1. Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. ACM SIGPLAN Notices, 39(1), 111–122.
2. Godefroid, P., Kiezun, A., Levin, M., Grammar-based whitebox fuzzing. PLDI 2008: 206-215.
3. Majumdar, R., Xu, R. G., Directed test generation using symbolic grammars. In ASE, 2007.
4. Dreyfus, A., Heam, P., Kouchnarenko, O., Random grammar- based testing for covering all non-terminals, in Proceedings of the 6th IEEE Conference of Software Testing, Verification and Validation Workshops (ICSTW), 2013, pp. 210–215.
5. Redziejowski, R. R. (2007). Parsing expression grammar as a primitive recursive-descent parser with backtracking. Fundamenta Informaticae, 79(3-4), 513–524.
6. Brzozowski, J. A. (1964). Derivatives of Regular Expressions. Journal of the ACM, 11(4), 481–494.
7. Owens, S., Reppy, J., & Turon, A. (2009). Regular-expression derivatives re-examined. Journal of Functional Programming, 19(02), 173–190.
8. Might, M., Darais, D., & Spiewak, D. (2011). Parsing with derivatives. ACM SIGPLAN Notices, 46(9), 189–195.
9. Aho, A. V., Sethi, R., & Ullman, J. D. (1986). Compilers: Principles, Techniques, and Tools, Addison-Wesley. ISBN 0-201-10088-6.

10. Flatt, M., PLT: Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc. (2010) http://racket-lang.org/tr1/.
11. Bracha, G. Executable Grammars in Newspeak. JAOO 2007: Java Technology and Object-Oriented Software Engineering, Aarhus, Denmark, September 2007.
12. Aaron Moss. Derivatives of Parsing Expression Grammars. CoRR abs/1405.4841 (2014)
13. Berry, G., The Streel v5 Language Primer Version 5.21 release 2.0. classes.soe.ucsc.edu/cmpe117/Spring02/esterel_primer.ps, 1999.
14. Sen, K., Rosu, G., Generating optimal monitors for extended regular expressions. Electronic Notes in Theoretical Computer Science (ENTCS) 89, 2. (2003).
15. Danielsson, N. A., Total parser combinators. SIGPLAN Not. 45, 9 (Sept. 2010), 285–296.
16. Seaman, R.P., Testing compilers of high level programming languages. IEEE Comput. Sys and Technol. (1974), pp. 366–375.
17. Bird, D., Munoz, C., Automatic Generation of Random Self-Checking Test Cases. IBM Systems Journal, 22(3):229–245, 1983.
18. Sirer, E., Bershad, B., Using production grammars in software testing. In DSL, 1999.
19. Forrester, J. E., Miller, B. P., An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In Proceedings of the 4th USENIX Windows System Symposium, Seattle, August 2000.
20. Mandl, R., Orthogonal latin squares: an application of experiment de- sign to compiler testing. Communications of the ACM, 28 (10) (1985), pp. 1054–1058.
21. Coppit, D., Lian, J., yagg: an easy-to-use generator for structured test inputs. In ASE, 2005.
22. Purdom, P. A., Sentence generator for testing parsers. BIT, 1972, 12(3): 366–375.
23. Lammel, R., Schulte, W., Controllable combinatorial coverage in grammar-based testing. In TestCom, 2006.
24. Renggli, L., Ducasse, S., Girba, and Nierstrasz, O., Practical dynamic grammars for dynamic languages. In Proc. of DYLA 2010 (4th Workshop on Dynamic Languages), 2010.
25. Warth, A., Douglass, J. R., & Millstein, T. (2008). Packrat parsers can support left recursion. In Proc. PEPM '08.
26. S. Medeiros, F. Mascarenhas, and R. Ierusalimschy, Left Recursion in Parsing Expression Grammars. In Proceedings of SBLP. 2012, 27-41
27. Tratt, L. (2010, October 25). Direct Left-Recursive Parsing Expression Grammars. Technical Report EIS-10-01. London, United Kingdom: Middlesex University.
28. A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. In DLS '07: Proceedings of the 2007 Dynamic Languages Symposium, pages 11–19, New York, NY, USA, 2007. ACM.
29. Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In Proc. Of International Conference on Functional Programming (ICFP), ACM SIGPLAN, 2000.