Derivatives of Parsing Expression Grammars

Aaron Moss

Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

`a3moss@uwaterloo.ca`

This paper introduces a new derivative parsing algorithm for recognition of parsing expression grammars. Derivative parsing is shown to have a polynomial worst-case time bound, an improvement on the exponential bound of the recursive descent algorithm. This work also introduces asymptotic analysis based on inputs with a constant bound on both grammar nesting depth and number of backtracking choices; derivative and recursive descent parsing are shown to run in linear time and constant space on this useful class of inputs, with both the theoretical bounds and the reasonability of the input class validated empirically. This common-case constant memory usage of derivative parsing is an improvement on the linear space required by the packrat algorithm.

1 Introduction

Parsing expression grammars (PEGs) are a parsing formalism introduced by Ford [6]. Any LR(k) language can be represented as a PEG [7], but there are some non-context-free languages that may also be represented as PEGs (*e.g.* $a^n b^n c^n$ [7]). Unlike context-free grammars (CFGs), PEGs are unambiguous, admitting no more than one parse tree for any grammar and input. PEGs are a formalization of recursive descent parsers allowing limited backtracking and infinite lookahead; a string in the language of a PEG can be recognized in exponential time and linear space using a recursive descent algorithm, or linear time and space using the memoized *packrat* algorithm [6]. PEGs are formally defined and these algorithms outlined in Section 3. This paper introduces a PEG recognition algorithm in Section 4 based on the context-free grammar derivative of Might *et al.* [13], which is in turn based on Brzozowski's regular expression derivative [4]. The essential idea of the derivative parsing algorithm is to keep an abstract syntax tree representing the "rest" of the original grammar, iteratively transforming it into a new grammar for every character in the input string. Unlike a packrat parser, which stores results of earlier parses to enable backtracking, the derivative parser never backtracks, instead calculating all possible "backtracks" concurrently at each point in the input. The analysis of this algorithm, presented in Section 5, reveals that it has quartic worst-case time complexity and cubic worst-case space complexity, though for inputs where both the grammar nesting depth and the number of backtracking options are bounded by a constant these bounds can be improved to linear time and constant space. This analysis is extended to consider the existing recursive descent and packrat algorithms on such inputs, providing a theoretical explanation for the superior practical performance of recursive descent, contrary to packrat's better asymptotic bounds. The experimental results presented in Section 6 empirically validate these theoretical predictions, as well as demonstrating the reasonability of the constant bounds for real-world inputs such as source code and structured data formats.

2 Related work

Ford [6] formalized PEG parsing and introduced the recursive descent and packrat algorithms. Medeiros and Ierusalimschy [11] have developed a parsing machine for PEGs. Their parsing machine is conceptually similar to a recursive descent parser, but they demonstrate superior performance in practice.

Mizushima *et al.* [14] augment packrat parsers with a *cut operator* which indicates that the current expression will not backtrack to another alternative. If there are no other valid alternatives for any expression currently being parsed then the entire packrat memoization table can be safely discarded; their empirical results on Java, XML, and JSON grammars show roughly constant space usage. Mizushima *et al.* also present a method for safe automatic insertion of cut operators, but their results demonstrate that it is not consistently effective and manual augmentation of the grammar is necessary for optimal performance. Redziejowkski [17] suggests that such automatic cut point insertion is likely to be insufficient for all but LL(1) grammars, limiting the utility of this approach. The derivative parsing algorithm presented in this paper, by contrast, has no need for manual augmentation of the grammar, and can trim backtracking choices dynamically based on the input string, rather than only statically based on the grammar.

Kuramitsu [9] has developed an *elastic sliding window* variant of packrat which uses constant space. His method uses a fixed-size memoization table with hash collision-based eviction, relying on dynamic analysis to selectively stop memoizing some nonterminals. Though reported runtime results using the elastic sliding window method are competitive with other approaches, it does forfeit the linear worst-case guarantee of packrat parsing. Redziejowkski [16] employs a similar but less sophisticated scheme in his Mouse parser-generator, storing a small fixed number of recent parsing results for each nonterminal. These methods heuristically discard backtracking information, in contrast to the derivative parsing algorithm, which keeps precisely the necessary backtracking information required.

Henglein and Rasmussen [8] present *progressive tabular parsing* (PTP) for the TDPL formalism PEGs are based on. PTP can iteratively generate a subset of the full parse table for a grammar on an input string, dynamically trimming a prefix of that table based on a tunable lookahead parameter. They prove linear time and space bounds for PTP; however, empirical comparison of their algorithm with other approaches is left to future work, so it is impossible to compare the constant factors to packrat, which has identical asymptotic bounds. Henglein and Rasmussen also show some evidence that constant space usage may be achieved in practice for JSON by their dynamic table trimming mechanism; it would be interesting to compare the efficacy of their method to that of derivative parsing or the automatic table trimming approaches mentioned above on a wider variety of grammars.

Might *et al.* [13] extended the Brzozowski regular expression derivative [4] to context-free grammars; Adams *et al.* [1] further optimized this approach and proved a cubic worst-case time bound. The derivative parsing algorithm presented here is based on their work, with extensions to account for the lookahead and ordered choice features of PEGs. Danielsson [5] uses derivatives as the underlying parsing mechanism of a parser combinator system. Brachthäuser *et al.* [3] have developed a parser combinator encapsulating the derivative calculation, allowing parsers to be manipulated as first-class objects in a parser combinator framework; they use this method to compose distinct parsers in an interleaved manner.

3 Parsing expression grammars

A PEG is expressed as a set of matching rules $A := \alpha$, where each *nonterminal A* is replaced by the *parsing expression* α. A parsing expression either matches an input string, possibly consuming input, or

fails, consuming no input. Formally, given a set of strings Σ^* over an alphabet Σ, a parsing expression α is a function $\alpha : \Sigma^* \to \{\text{match}, \text{fail}\} \times \Sigma^*$. $\mathcal{N}$ denotes the set of nonterminals, $\mathcal{X}$ the set of parsing expressions, and $\mathcal{R}$ the function mapping each nonterminal to its corresponding parsing expression, $\mathcal{R} : \mathcal{N} \to \mathcal{X}$. A grammar $\mathcal{G}$ is the tuple $(\mathcal{N}, \mathcal{X}, \Sigma, \mathcal{R}, \sigma)$, where $\sigma \in \mathcal{X}$ is the *start expression* to parse. In this paper, lowercase Greek letters are used for parsing expressions ($\varphi \in \mathcal{X}$), uppercase Roman letters for nonterminals ($N \in \mathcal{N}$), lowercase monospace letters for characters ($\mathtt{c} \in \Sigma$), and lowercase non-italic Roman letters for strings ($\mathrm{s} \in \Sigma^*$). For a parsing expression φ, the language $\mathcal{L}(\varphi)$ is the set of strings matched by φ, formally $\mathcal{L}(\varphi) = \{\mathrm{s} \in \Sigma^* : \exists \mathrm{s}', \varphi(\mathrm{s}) = (\text{match}, \mathrm{s}')\}$. A parsing expression φ is *nullable* if $\mathcal{L}(\varphi) = \Sigma^*$. Since parsing expressions match prefixes of their input, nullable expressions can be thought of as those that match the empty string without any restrictions on what comes after it.

The simplest parsing expression is the *empty expression* ε; ε always matches, consuming no input. Another fundamental parsing expression is the *character literal* c that matches and consumes a c or fails. Expressions are concatenated by a *sequence expression*: $\alpha\beta$ matches if both α and β match one after the other, failing otherwise. PEGs also provide an *alternation expression*: α/β first attempts to match α, then tries β if α fails. Unlike the unordered choice in CFGs, once a subexpression matches in a PEG none of the others are tried. Specifically, ab/a and a/ab are distinct expressions, and the ab in the second expression never matches, as any string it would match has already matched a. The *negative lookahead expression* $!\alpha$ never consumes any input, matching if α fails. Expressions can be recursive; the *nonterminal expression N* matches only if its corresponding parsing expression $\mathcal{R}(N)$ matches, consuming whatever input $\mathcal{R}(N)$ does. For instance, $A := a\,A\ /\ \varepsilon$ matches a string of any number of a's.

This repetition can be achieved by the *repetition expression* $\alpha*$, syntactic sugar for a nonterminal like A for $a*$ above. Due to the ordered choice in PEGs this matches greedily, so, *e.g.* $a*\,a$ never matches, since $a*$ consumes all available a's, leaving none for a to match. Since repetition expressions add no expressive power to PEGs this paper generally omits them from discussion. Some other common syntactic sugar for PEGs is $\alpha?$ for α/ε and $\alpha+$ for $\alpha\,\alpha*$. $\&\alpha$, the *positive lookahead operator*, matches only if α matches, but consumes no input; it can be written $!(!\alpha)$. $\bullet$ abbreviates "any character in Σ", while $[a_0 \cdots a_k]$ is shorthand for the alternation $a_0/\cdots/a_k$. Less commonly, several uses of negative lookahead may be profitably simplified through use of a dedicated parsing expression: the negative character class $[^\wedge a_0 \cdots a_k]$ for $![a_0 \cdots a_k]\,\bullet$, the end-of-input expression $\$$ for $!\bullet$, and the *until expression* $\alpha \to \beta$ for $(!\beta\,\alpha)*\,\beta$ (this repetition of α until a β is seen is actually implemented as a fresh nonterminal $U := \beta/\alpha U$, taking advantage of the ordered choice property of PEGs to avoid matching β twice). Table 1 has formal definitions of each of the fundamental expressions in decreasing order of precedence.

3.1   Recursive descent algorithm

Since parsing expression grammars are a formalization of recursive descent parsers, it should not be surprising that they may be recognized by a direct conversion of the expressions in Table 1 into a recursive algorithm with a function for each nonterminal. This approach is simple and space-efficient, requiring only enough space to store the input string and function call stack (possibly linear in the size of the input). Recursive descent parsing may require exponential runtime; for example, consider a string of the form $a^n c^n$ as input to the grammar in Figure 1. In this case, the function A is called twice after the first a, each of those calls resulting in two calls to A after the second a, and so forth, as at each position $0 < i < n$ in the input string A assumes that each of the $n - i$ remaining a's is eventually followed by a b, fails, and must repeat this process assuming the a's are followed by a c.

This time bound does not account for left-recursive grammars, where a nonterminal may call itself

$$a(s) = \begin{cases} (\text{match}, s') & s = a\,s' \\ (\text{fail}, s) & \text{otherwise} \end{cases}$$

$$\varepsilon(s) = (\text{match}, s)$$

$$A(s) = (\mathscr{R}(A))(s)$$

$$!\alpha(s) = \begin{cases} (\text{match}, s) & \alpha(s) = (\text{fail}, s) \\ (\text{fail}, s) & \text{otherwise} \end{cases}$$

$$\alpha\beta(s) = \begin{cases} (\text{match}, s'') & \alpha(s) = (\text{match}, s') \wedge \beta(s') = (\text{match}, s'') \\ (\text{fail}, s) & \text{otherwise} \end{cases}$$

$$\alpha/\beta(s) = \begin{cases} (\text{match}, s') & \alpha(s) = (\text{match}, s') \\ (\text{match}, s'') & \alpha(s) = (\text{fail}, s) \wedge \beta(s) = (\text{match}, s'') \\ (\text{fail}, s) & \text{otherwise} \end{cases}$$

Table 1: Formal definitions of parsing expressions.

without consuming any input. Though left-recursion is allowable in CFGs, it causes infinite loops in PEGs; an example of such a grammar would be $A := A\,a\,/\,\varepsilon$. Ford [7] defines a *well-formed* grammar as one that contains no directly or indirectly left-recursive rules. This property is structurally checkable, so this paper considers left-recursion in PEGs to be undefined behaviour, and implicitly assumes all grammars are well-formed to guarantee a finite time bound for recursive descent and packrat parsing.

3.2 Packrat parsing

The essential idea of packrat parsing, introduced by Ford [6], is to memoize calls to nonterminal functions in the recursive descent algorithm. If repetition expressions are treated as syntactic sugar for an anonymous nonterminal as described above, all parsing expressions except for nonterminal invocations can be parsed in time proportional to their number of subexpressions, a constant. Once a nonterminal has been evaluated at a position it can be parsed in constant time by reading the result from the memoization table. In any well-formed grammar, nonterminals do not left-recursively invoke themselves, and so may be parsed in constant time after parsing of all subsidiary nonterminals. Since there are a constant number of nonterminals, which may be parsed no more than once for each position in the input, this algorithm takes linear time, using linear space to store the memoization entries for each nonterminal at each index.

$$S := A\ \$$$
$$A := a\,A\,b\ /\ a\,A\,c\ /\ \varepsilon$$

Figure 1: Grammar with exponential runtime using the recursive descent algorithm.

4   Derivative parsing

This paper presents a new algorithm for recognizing parsing expression grammars based on the derivative parsing algorithm for context-free grammars introduced by Might *et al.* [13]. Essentially, the *derivative* of a parsing expression φ with respect to a character c is a new parsing expression $d_c(\varphi)$ that recognizes everything after the initial c in $\mathscr{L}(\varphi)$. As an example, $d_f(foo/bar/baz) = oo$, while $d_b(foo/bar/baz) = ar/az$, $d_a(ar/az) = r/z$, and $d_r(r/z) = \varepsilon$. Where the specific character is not relevant, the derivative of φ may be written φ'. Formally, $\mathscr{L}(d_c(\varphi)) = \{s : c\,s \in \mathscr{L}(\varphi)\}$. The essence of the derivative parsing algorithm is to take repeated derivatives of the original parsing expression for each character in the input and to check for a nullable or failure parser at each step. The repeated derivative $d_{c_k} \circ \cdots \circ d_{c_1}(\varphi)$ may be written $d_{c_1 \cdots c_k}(\varphi)$, or $\varphi^{(k)}$ where the specific characters are not important. To compute derivatives of parsing expressions, two new parsing expressions are introduced for failure states: $\varnothing$ and ∞; $\varnothing(s) = \infty(s) = (\text{fail}, s)$. $\varnothing$ represents a generic parse failure, while ∞ represents a left-recursive infinite loop, which this algorithm can detect and terminate on.

Derivative parsing essentially tries all possible parses concurrently; since parsing expression grammars are inherently unambiguous, they may actually be better suited to this approach than context-free grammars. Nonetheless, tracking exactly which expressions in the current parse tree can match against or consume the current character is the key problem in this algorithm. To solve this problem, the algorithm uses the concept of *backtracking generations* (*generations*) as a way to account for choices in the parsing process. Each backtracking generation represents a point in the input at which a parsing expression may have matched and ceased to consume input. Generation 0 is the "leftmost" ongoing parse, where each new character is consumed by the first subexpression of any sequence expression, while higher generations represent backtracking decisions that occurred at some earlier point and have not yet been resolved. Expressions may have multiple outstanding backtracking decisions; to represent this, a set of generations is denoted with uppercase sans-serif characters, such that $\mathsf{X} = \{x_0, x_1, \cdots, x_k\}$, where the indices $0, \cdots, k$ are assigned from smallest to largest value.

To perform the derivative computation, some expressions must be augmented with generation information. The ε expression is augmented with a generation i, written $\varepsilon[i]$; $\varepsilon[0]$ represents a match at the current input location, while $\varepsilon[i > 0]$ means that the backtracking decision deferred at generation i did actually result in a match. In general, expressions may have different generation sets than their subexpressions; to translate one generation set into another composition of generation sets is defined as $\mathsf{F} \circ \mathsf{G} = \{f_i : i \in \mathsf{G}\}$ (*e.g.* $\{0,3,4\} \circ \{0,2\} = \{0,4\}$). Alternation expressions α/β are annotated with $[\mathsf{A}, \mathsf{B}, m]$, where A and B are the mapping sets for α and β, respectively, and m is the maximum backtracking generation for the expression. A and B map new generations of α and β from the same input index to the same value, and generations that first occur at different input indices to different values, allowing the separate backtracking generations of α and β to be expressed as generations of the compound expression α/β. As an example, consider $\varphi = a!c/!(ab)$; $a!c$ starts by consuming characters, so its generation mapping set $\mathsf{A} = \{0\}$, while $!(ab)$ represents a *possible* match (which would later produce a $\varepsilon[1]$), so its initial mapping set $\mathsf{B} = \{0,1\}$, to map that $\varepsilon[1]$ to a generation 1 match of the containing expression. Thus, the initial form of φ is $a!c/!(ab)[\{0\}, \{0,1\}, 1]$. When the derivative of this expression is taken, the generation information must be updated; $d_a(\varphi) = !c/!b$, but since the $!c$ represents a possible match at a later position than the $!b$ it needs a new backtracking generation, 2. To map a $\varepsilon[1]$ success result from $!c$ to a $\varepsilon[2]$ for φ, the generation information is updated to $!c/!b[\{0,2\}, \{0,1\}, 2]$.

Since sequence expressions $\alpha\beta$ encode the successor relationship, the bulk of the complication of dealing with backtracking must also be handled in sequence expressions. The first type of backtracking handled by a sequence expression is *nullability backtracking*, where α is a nullable but non-empty

φ	$back(\varphi)$	$match(\varphi)$
a	$\{0\}$	$\{\}$
$\varepsilon[i]$	$\{i\}$	$\{i\}$
$\varnothing$	$\{0\}$	$\{\}$
∞	$\{0\}$	$\{\}$
A	$back(\mathscr{R}(A))$ where $back(A) = \{0\}$	$match(\mathscr{R}(A))$ where $match(A) = \{\}$
$!\alpha$	$\{1\}$	$\{\}$
$\alpha[\mathsf{A}, m]$	$\mathsf{A} \circ back(\alpha)$	$\mathsf{A} \circ match(\alpha)$
$\alpha/\beta[\mathsf{A}, \mathsf{B}, m]$	$\mathsf{A} \circ back(\alpha) \cup \mathsf{B} \circ back(\beta)$	$\mathsf{A} \circ match(\alpha) \cup \mathsf{B} \circ match(\beta)$

$$\varphi = \alpha\beta[\beta_1, \cdots, \beta_k, \beta_\emptyset, \mathsf{B}_1, \cdots, \mathsf{B}_k, \mathsf{B}_\emptyset, l_1, \cdots, l_k, m]$$
$$back(\varphi) = (0 \in back(\alpha) \text{ ? } \{0\} : \{\}) \cup_{i \in 1..k} (\mathsf{B}_i \circ back(\beta_i)) \cup \{l_i : l_i > 0\} \cup \mathsf{B}_\emptyset \circ back(\beta_\emptyset)$$
$$match(\varphi) = \cup_{i \in match(\alpha)} (\mathsf{B}_i \circ match(\beta_i)) \cup \{l_i : l_i > 0\} \cup \mathsf{B}_\emptyset \circ match(\beta_\emptyset)$$

Table 2: *back* and *match* Definitions

expression, *e.g.* $\alpha\beta = (\delta/\varepsilon)\beta$. This example is almost equivalent to $\delta\beta/\beta$, and is handled similarly to an alternation expression, except that a later nullable derivative $\delta^{(k)}$ causes the second alternative $\beta^{(k)}$ to be discarded[1]. To account for this possibility the sequence expression $\alpha\beta$ is augmented with $\beta_\emptyset$ and $\mathsf{B}_\emptyset$, the *failure backtrack* and the generation set that maps its backtracking generations into the sequence expression's generation space. $\beta_\emptyset$ is the parse to continue if $\alpha^{(k)}$ is nullable (*i.e.* a match, where β should possibly begin parsing) but some later $\alpha^{(k+i)}$ fails (*i.e.* $\alpha^{(k)}$ was the match which should be continued from).

Sequence expressions $\alpha\beta$ also handle *lookahead backtracking*. For each generation where α may have ceased to consume input, yet did not unambiguously match or fail (*e.g.* by including a lookahead expression), $\alpha\beta$ must begin to track derivatives of β so that the proper derivative can be computed if one of those generations eventually results in a match. Sequence expressions implement lookahead backtracking by keeping the *lookahead follower* β_i, the derivative of β started when generation i first appeared, its associated generation map B_i, and the index l_i of the last generation of $\alpha\beta$ where β_i was nullable (0 for none such). The l_i are necessary to track possible matches of $\alpha\beta$ where both subexpressions have been consumed. In addition to nullability and lookahead backtracking information, sequence expressions are also annotated with m, the maximum backtracking generation of the expression.

More formally, the functions *back* and *match* produce sets of backtracking generations from parsing expressions; $back(\varphi)$ is the set of backtracking generations currently being parsed by φ while $match(\varphi)$ is the subset of $back(\varphi)$ that has successfully matched. In terms of the existing literature on derivative parsing, *match* is an extension of the nullability function and $0 \in match(\varphi)$ means that φ is nullable. The values of *back* and *match* for all expressions are defined in Table 2; parsing expressions which need additional bookkeeping information to perform these calculations include it in square brackets. Note that *back* and *match* for a nonterminal A are fixed-point computations where A acts like an infinite loop ∞ if encountered recursively. This yields correct results if *match* is pre-computed for each subexpression in the original grammar by a fixed-point computation. Might *et al.* [13] use an approach based on Kleene's least-fixed-point theorem that can be straightforwardly adapted to this algorithm. Once this initial fixed

---

[1]$(\delta/\varepsilon)\beta$ is not exactly equivalent to $\delta\beta/\beta$; consider $\delta = a$, $\beta = a+$; $\mathsf{a} \in \mathscr{L}(a\,a+/a+)$, but $\mathsf{a} \notin \mathscr{L}((a/\varepsilon)a+)$, as (a/ε) consumes the a but $a+$ still requires one.

point is calculated the nonterminal rules preclude the introduction of any further recursion, and thus the need for further fixed point calculations. A *map expression* $\alpha[\mathsf{A},m]$ is also introduced here to remap *back* and *match* sets into a different generation space; it has the same semantics as α, *i.e.* $\alpha[\mathsf{A},m](\mathrm{s}) = \alpha(\mathrm{s})$.

Derivatives update the generation maps of their expressions via the following update function:

$$up(\mathsf{P}, \varphi, \varphi', m) = \begin{cases} \mathsf{P} \cup \{m+1\} & \max(back(\varphi')) > \max(back(\varphi)) \\ \mathsf{P} & \text{otherwise} \end{cases}$$

up adds a new generation to the map P for any expression φ that adds a new backtracking generation when transformed to φ'. Since backtracking generations correspond to points in the input where a backtracking decision has been deferred, there is no more than one new generation for each parsing expression per derivative taken, and expressions statically defined in the original grammar can have a backtracking generation no greater than 1. To transform these static generations into the generation space of an ongoing derivative computation, the following *new generation set* function is defined:

$$ngs(\varphi, m) = \begin{cases} \{0, m+1\} & \max(back(\varphi)) > 0 \\ \{0\} & \text{otherwise} \end{cases}$$

Before it can be used in a derivative calculation, the original grammar needs some pre-processing to insert backtracking annotations. Any ε expressions in the original grammar become $\varepsilon[0]$. For an alternation expression $\alpha/\beta[\mathsf{A}, \mathsf{B}, m]$ the initial values of A and B are $back(\alpha)$ and $back(\beta)$, respectively; m is initialized to $\max(\mathsf{A}, \mathsf{B})$. Sequence expressions are more complicated: given a sequence expression $\alpha\beta$, if α is nullable (*i.e.* $0 \in match(\alpha)$) then the initial failure lookahead $\beta_\emptyset = \beta$, otherwise $\beta_\emptyset = \varnothing$; in either case $\mathsf{B}_\emptyset = back(\beta_\emptyset)$. If $1 \in back(\alpha)$ then $\alpha\beta$ has an initial lookahead follower $\beta_1 = \beta$ with a generation set $\mathsf{B}_1 = back(\beta)$, if β is nullable then the lookahead nullability marker $l_1 = 1$, otherwise $l_1 = 0$; $m = \max(0, \mathsf{B}_\emptyset, \mathsf{B}_1, l_1)$, for any of these defined. All these constructions preserve the property that expressions in the original grammar have a maximum backtracking generation of 1.

Having defined the necessary helper functions, Table 3 describes a single derivative step. a' is a straightforward derivative of a terminal, consuming an a or failing. $\varepsilon[0]'$ is a failure on anything except end-of-input because there is nothing left in an $\varepsilon[0]$ to take a derivative of, while $\varepsilon[i > 0]'$ is preserved to indicate a possible match that has yet to be verified; the failure expressions $\varnothing$ and ∞ are similarly preserved by the derivative operation. Nonterminals are again assumed to be ∞ while calculating their derivatives to break infinite loops in the computation; for well-formed grammars this case should not occur. Negative lookahead, mapping and alternation expressions are basically distributive with respect to the derivative; the initial three cases for both $(!\alpha)'$ and $\alpha[\mathsf{A},m]'$ distribute matches and failures according to the appropriate semantics, while the fourth case of $\alpha[\mathsf{A},m]'$ strips unnecessary identity maps. The third case for $\alpha/\beta[\mathsf{A}, \mathsf{B}, m]'$ is of particular note; it implements the ordered choice property of PEGs by stripping the β alternative in the case of any match in the α alternative.

The first case of the sequence expression derivative accounts for a straightforward match of α at generation 0, which is followed by a parse of β mapped into the sequence expression's generation space. In the second and third cases α has matched at generation $i > 0$ and is mapped to the appropriate lookahead follower; the third case accounts for the possibility that the lookahead follower has already passed its final match by alternating it with a match at its nullability index l_i. The fifth case propagates an infinite loop failure, while the fourth case produces the failure backtrack if α fails; this backtrack will be a failure expression itself if α has never matched. Generally speaking, the sixth case takes derivatives of α and all the β_i and $\beta_\emptyset$ lookahead and nullability backtracks. Particularly, k' accounts for whether α produced any

$$a' = \begin{cases} \varepsilon[0] & c = \mathtt{a} \\ \varnothing & \text{otherwise} \end{cases} \qquad\qquad \varepsilon[i]' = \begin{cases} \varepsilon[i] & i > 0 \vee c = \Diamond \\ \varnothing & i = 0 \end{cases}$$

$$\varnothing' = \varnothing \qquad\qquad\qquad\qquad\qquad \infty' = \infty$$

$$A' = \mathscr{R}(A)' \text{ under } A' = \infty$$

$$(!\alpha)' = \begin{cases} \varnothing & match(\alpha') \neq \emptyset \\ \varepsilon[1] & \alpha' = \varnothing \\ \infty & \alpha' = \infty \\ !(\alpha') & \text{otherwise} \end{cases}$$

$$\alpha[A,m]' = \begin{cases} \varepsilon[A' \circ \{i\}] & \alpha' = \varepsilon[i] & \text{where } A' = up(A, \alpha, \alpha', m) \\ \varnothing & \alpha' = \varnothing & m' = \max(m, A') \\ \infty & \alpha' = \infty & \\ \alpha' & A' = \{0, 1, ..., m'\} & \\ \alpha'[A', m'] & \text{otherwise} & \end{cases}$$

$$\alpha/\beta[A,B,m]' = \begin{cases} \beta'[B', m'] & \alpha' = \varnothing & \text{where } A' = up(A, \alpha, \alpha', m) \\ \infty & \alpha' = \infty & B' = up(B, \beta, \beta', m) \\ \alpha'[A', m'] & \beta' = \varnothing \vee match(\alpha') \neq \emptyset & m' = \max(m, A', B') \\ \alpha'/\beta'[A', B', m'] & \text{otherwise} & \end{cases}$$

$$\alpha\beta[\beta_1, \cdots, \beta_k, \beta_\emptyset, B_1, \cdots, B_k, B_\emptyset, l_1, \cdots, l_k, m]'$$

$$= \begin{cases} \beta[ngs(\beta, m), m'] & \alpha' = \varepsilon[0] \\ \beta_i'[B_i', m'] & \alpha' = \varepsilon[0 < i \leq k'] \wedge l_i' = 0 \\ (\beta_i'/\varepsilon[1])[B_i', \{0, l_i'\}, m'] & \alpha' = \varepsilon[0 < i \leq k'] \wedge l_i' > 0 \\ \beta_\emptyset'[B_\emptyset', m'] & \alpha' = \varnothing \\ \infty & \alpha' = \infty \\ \alpha'\beta^\dagger[\beta_1', \cdots, \beta_{k'}', \beta_\emptyset^\dagger, B_1', \cdots, B_{k'}', B_\emptyset^\dagger, l_1', \cdots, l_{k'}', m'] & \text{otherwise} \end{cases}$$

where

$$m' = \max(m, B_i', B_\emptyset', l_i') \text{ for any of these used}$$

$$\beta^\dagger = \begin{cases} \beta' & c = \Diamond \wedge \alpha' = \varepsilon[i \in \{0, k+1\}] \\ \beta & \text{otherwise} \end{cases} \qquad k' = \begin{cases} k+1 & \max(back(\alpha')) > \max(back(\alpha)) \\ k & \text{otherwise} \end{cases}$$

$$\beta_{k+1}' = \beta \qquad\qquad\qquad\qquad\qquad \beta_\emptyset^\dagger = \begin{cases} \beta & 0 \in match(\alpha') \\ \beta_\emptyset' & \text{otherwise} \end{cases}$$

$$\begin{aligned} B_{1 \leq i \leq k}' &= up(B_i, \beta_i, \beta_i', m) \\ B_{k+1}' &= ngs(\beta, m) \end{aligned} \qquad\qquad B_\emptyset^\dagger = \begin{cases} ngs(\beta, m) & 0 \in match(\alpha') \\ up(B_\emptyset, \beta_\emptyset, \beta_\emptyset', m) & \text{otherwise} \end{cases}$$

$$l_{k+1} = 0 \qquad\qquad\qquad\qquad\qquad l_{1 \leq i \leq k+1}' = \begin{cases} m+1 & 0 \in match(\beta_i') \\ l_i & \text{otherwise} \end{cases}$$

Table 3: Definition of $d_c(\varphi)$; δ' abbreviates $d_c(\delta)$.

$$parse(\varphi, \mathrm{s} = [\mathrm{c_1 c_2 \ldots c_n} \Diamond]) = \begin{cases} \mathrm{match} & match(\varphi) \neq \emptyset \\ \mathrm{fail} & \varphi \in \{\varnothing, \infty\} \\ \mathrm{fail} & \mathrm{s} = [\,] \wedge match(\varphi) = \emptyset \\ parse(d_{c_1}(\varphi), [\mathrm{c_2 \ldots c_n} \Diamond]) & \mathrm{otherwise} \end{cases}$$

Table 4: Derivative parsing recognizer function

new backtracking generations in this derivative step, while l'_i tracks the last nullability of the lookahead backtracks. The $\beta_\emptyset^\dagger$ and $B'_\emptyset$ cases are especially notable, as they implement the ordered choice property of PEGs by resetting to β whenever α is nullable. The special cases for end-of-input ($c = \Diamond$) are necessary to resolve outstanding lookahead expressions; for instance, $!\bullet$ is one idiomatic way to represent end-of-input in a PEG, but $\bullet$ must parse to a failure by receiving a non-character (*i.e.* $\Diamond$) to recognize $!\bullet$ as a match. In the $\beta^\dagger$ special case of the sequence expression derivative, β must also be matched against $\Diamond$ if $d_\Diamond(\alpha)$ resulted in a fresh ε-expression.

Given this definition of derivative, the *parse* function defined in Table 4 evaluates a parsing expression φ on an input string $\mathrm{c_1 c_2 \ldots c_n}$. This function computes repeated derivatives until either the input is exhausted or the derivative becomes a success or failure value. The fact that the rest of the string is not returned on a match should not be a practical problem: further parsing could be performed by wrapping the start expression in another expression or the *parse* function could also be augmented with logic to track the indices of each backtracking generation to return the proper suffix of the input on a match.

The derivative parsing algorithm as presented works correctly, but has exponential time and space complexity, since each subexpression of a parsing expression may expand by the size of the grammar during each derivative. This problem can be avoided by memoizing the derivative computation; if a table of parsing expression derivatives φ' is constructed for each derivative step, duplicate derivative calculations can be avoided by using a single instance of φ' for all derivatives of φ. This changes the derivative tree into a directed acyclic graph (DAG), and also bounds the increase in derivative size and computation time, allowing this parsing algorithm to achieve the polynomial time and space bounds proved in Section 5. This memoization is also the reason for the generation sets and *back* and *match* functions; these mechanisms allow a single subexpression to have multiple parents, yet be treated differently in each context. As an additional optimization, the values of *back* and *match* are also memoized.

The treatment of backtracking generations can also be improved by eliminating information that is never used. If a generation i is not present in $back(\alpha)$, then the sequence expression $\alpha\beta$ does not need to keep β_i, B_i, and l_i, since α can never reduce to $\varepsilon[i]$, which would select β_i. Similarly, in the map expression $\alpha[A = \{a_0, a_1, \cdots, a_k\}, m]$, it is not necessary to store a_i for any $i \notin back(\alpha)$, so long as the indices for the other values in A are preserved; this transforms A to a partial function of indices to values. The same principle can be applied to the backtracking sets in the alternation and sequence expressions as well. If these optimizations are applied, the space required for backtracking information becomes proportional to the number of current backtracking choices, rather than the potentially much larger number of backtracking choices seen throughout the parsing process.

To summarize, the derivative parsing algorithm for PEGs works by iterative transformation of an abstract syntax tree representing the "rest" of the grammar. Unlike a packrat parser, which stores parse results for efficient backtracking, the derivative parser never backtracks, instead calculating all possible "backtracks" concurrently at each point in the input. Unlike CFGs, PEGs do not allow multiple parse

trees or left-recursion, so this algorithm produces only one parse tree, and does not require general-purpose fixed-point computations after pre-processing, as in derivative parsing for CFGs.

5 Analysis

The derivative parsing algorithm runs in time proportional to the cost of a derivative computation times the size of the input, and requires as much space as the largest derivative computed. The cost of a derivative computation $d_c(\varphi)$ is linear in the size of φ, but since φ is a memoized DAG, it may re-use the computation of some sub-expressions of φ. To account for this, $^{\#}(\varphi)$ is defined to be the number of *unique* subexpressions of φ, including φ itself. Some expressions are larger than others, and take longer to compute the derivative of, *e.g.* $\alpha/\beta[\mathsf{A},\mathsf{B},m]$ requires more storage and computation than $!\alpha$. To formalize this idea, the asymptotic bounds on the size $|\varphi|$ of the root node of φ are given in Table 5; this size is expressed in terms of b, the current number of generations. Each backtrack set is obviously $O(b)$, while a sequence expression keeps bookkeeping information for $O(b)$ generations.

It is evident from Table 3 that, with the exception of A', no derivative adds more expression nodes or makes an existing node asymptotically larger. In fact, many rules strictly reduce the size of their expression node or discard subexpressions. Furthermore, since none of the derivative rules add more than one new backtracking generation, b is linear in the number of derivatives taken.

Having bounded the size of a single expression node, it can now be shown that the number of expression nodes is also linear in the number of derivatives taken. The proof of this statement follows directly from the fact that taking the derivative of a nonterminal A is the only operation that adds nodes to the expression, replacing A with $\mathcal{R}(A)'$. But since $\mathcal{R}(A)$ is statically defined $^{\#}(\mathcal{R}(A)')$ is a constant, and since derivatives are memoized the expression nodes comprising $\mathcal{R}(A)'$ are only added once per derivative step. $\mathcal{R}(A)'$ may include expansions of other nonterminals to their respective derivatives, but cannot include a recursive expansion of A, since recursive expansions of A are replaced by ∞. Since an ∞ expression is the same size as an A expression, this expansion does not add any expression nodes, and $^{\#}(\mathcal{R}(A)')$ is bounded by a constant the size of the grammar. According to this argument, the number of expression nodes added during a derivative computation is bounded by $\sum_{A \in \mathcal{N}} {}^{\#}(\mathcal{R}(A)')$, a constant, and thus the number of expression nodes is at most linear in the number of derivatives taken.

Having established that the number of expression nodes is at most linear in the number of derivatives taken and the size of each of those nodes is no more than quadratic in the number of backtracking options (where the number of backtracking options is also linear in the number of derivatives taken), it follows directly that the space required by the derivative parsing algorithm is $O(b^2 n)$ ($O(n^3)$). As each derivative operation must traverse one of these expressions, the worst-case running time for this algorithm is $O(b^2 n^2)$ ($O(n^4)$). This polynomial bound guarantees quicker worst-case execution than the recursive descent algorithm, but does not match the bound of packrat parsing.

$$|a| = |\varepsilon[i]| = |\varnothing| = |\infty| = |A| = |!\alpha| = O(1)$$
$$|\alpha[\mathsf{A},m]| = |\alpha/\beta[\mathsf{A},\mathsf{B},m]| = O(b)$$
$$|\alpha\beta[\beta_1,\cdots,\beta_k,\beta_\varnothing,\mathsf{B}_1,\cdots,\mathsf{B}_k,\mathsf{B}_\varnothing,l_1,\cdots,l_k,m]| = O(b^2)$$

Table 5: Size $|\varphi|$ of the root node of an expression φ in terms of b, the current number of generations.

5.1 "Well-behaved" inputs

However, many real-world inputs have properties that allow these worst-case bounds to be significantly tightened. If the number of backtracking choices b at each point in the input is bounded by a constant then each expression node is also constant size and these bounds are reduced to linear space and quadratic time. The results in Section 6 show this bound holds in practice; as a theoretical argument, many practical grammars are LR, and thus can choose the correct branch of an alternation expression after reading a constant number of terminals past the end of the expression.

It is well-known (and verified in Section 6) that real-world inputs such as source code and structured data typically do not contain deeply nested recursive structure. To formalize this, the *nesting depth* of an input string with respect to a grammar is the maximum number of nested expressions containing any terminal expression, where the depth of α in $\alpha*$ is 1, as in an iterative or tail-call optimized implementation. Bounding the nesting depth of the input by a constant implies an equivalent constant bound on the depth of the DAG representing the derivative parser, as the DAG can only get deeper by taking the derivative of a nonterminal. Since the bound on backtracking restricts fanout, there is a constant bound on the number of expression nodes produced in the derivative parse of an input file with constant-bounded nesting depth and backtracking choices. Since these nodes require constant space, as shown above, these two conditions on the input are sufficient to guarantee constant space usage for a derivative parser, implying a linear time bound.

The time and space bounds of the recursive descent algorithm can also be tightened for inputs with constant-bounded backtracking and nesting depth. Nesting depth corresponds directly to stack size for this algorithm, so a constant bound on nesting depth is also a constant bound on the space usage of the recursive descent algorithm, assuming the input is not counted in this space bound and allows random seek (otherwise recursive descent still needs linear space to buffer the input). For the recursive descent algorithm, a bound on the number of backtracking options b at each point in the input can be interpreted as a bound on the number of times any character in the input is read; hence, a constant bound on b implies that no character is read more than b times, a linear bound on the runtime of the algorithm.

It has been empirically shown [9, 14] that packrat parsers can also achieve linear time and constant space for many practical grammars by selectively discarding backtracking information; however, the two conditions discussed here are not sufficient to guarantee this space usage. As evidence, consider $a*b \,/\, a*c$; the memoization table should store entries for every a in the input (which each begin an $a*$ parse), a linear space requirement, while both the derivative and recursive descent algorithms would use constant space to parse this expression. It is known [2, 9, 11] that recursive descent parsing often outperforms packrat parsing in practice, contrary to the worst-case analysis; to the author's knowledge, the explanation of this phenomenon as a class of practical inputs on which recursive descent has superior asymptotic bounds to packrat is a novel contribution of this work.

In summary, derivative parsing has a polynomial runtime bound, unlike recursive descent parsing, yet can parse a wide class of useful inputs in linear time and constant space, improving on the space bound of packrat parsing with no asymptotic sacrifice in time.

6 Experimental results

The author has implemented a derivative parser in C++ to empirically investigate the runtime performance of derivative parsing. The most significant optimization not covered in the theoretical description of the algorithm is that the implementation takes advantage of the associativity of alternation to flatten binary trees of alternation nodes to lists of choices; this resulted in modest speedup for grammars con-

taining alternations with many options. As done by Adams *et al.* [1], the implementation also uses a memoization field in each node instead of a separate table. Limiting the number of lookahead expressions by replacing them with negative character classes, end-of-input, and until expressions was also a useful optimization, saving 20% runtime in some tests. An imperative variant of the algorithm was attempted, with the intent to increase reuse of long-lived compound expression nodes by utilizing mutation, but it did not increase performance and added significant implementation complexity.

To test the predictions made by the analysis presented in Section 5, parser-combinator-based recursive descent and packrat parser-generators were also implemented by the author. All three parsers were tested using XML, JSON, and Java inputs from Mizushima *et al.* [14]. Code and test data are available online [15]. All tests were compiled with g++ 6.2.0 running on a machine with 8 GB of RAM, a dual-core 2.6 GHz processor, and SSD main storage. No formal proof of the correctness of the derivative algorithm is presented, but it produces the same results as the other algorithms on an extensive test suite.

Figure 2 shows the runtimes of each of the three algorithms, separated by grammar and graphed against input size. As can be seen, all three algorithms take linear time on the given inputs, though the constant factor depends both on the algorithm and the complexity of the grammar. More complex grammars result in longer runtime; in this case the XML grammar has 23 nonterminals, the JSON grammar has 24 nonterminals, and the Java grammar has 178 nonterminals.

Figure 3 demonstrates that the linear time guarantee provided by packrat parsing comes at a heavy cost in memory (note the logarithmic scale on both axes), with the linear bound on memory usage fitting tightly and resulting in memory usage two orders of magnitude greater than the size of the input. By contrast, derivative parsing uses constant space comparable to the input size for the well-behaved instances in this test, while the recursive descent algorithm uses a similarly small amount of space, with the slight linear growth due to buffering of the input to facilitate backtracking.

Figures 4 and 5 empirically validate the claims in Section 5.1 of constant bounds on backtracking and nesting depth. Backtracks reported in Figure 4 are the maximum number of times the input string or memoization table was read at any one index for recursive descent and packrat parsing, and the maximum number of leaf nodes in the expression tree for derivative parsing; it can be seen from these results that these maximums are effectively constant for a given grammar. It is interesting to note that packrat memoization is not markedly effective at reducing backtracking, with backtracks eliminated by memoization fairly closely balanced by duplicated hits due to memoization cache misses. The lower amount of backtracking in the derivative algorithm is likely an artifact of the methodology: since failures are
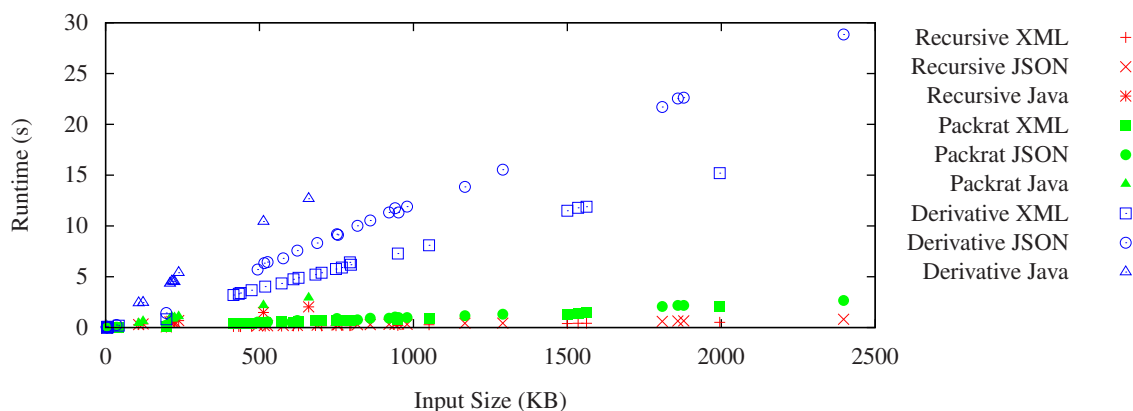

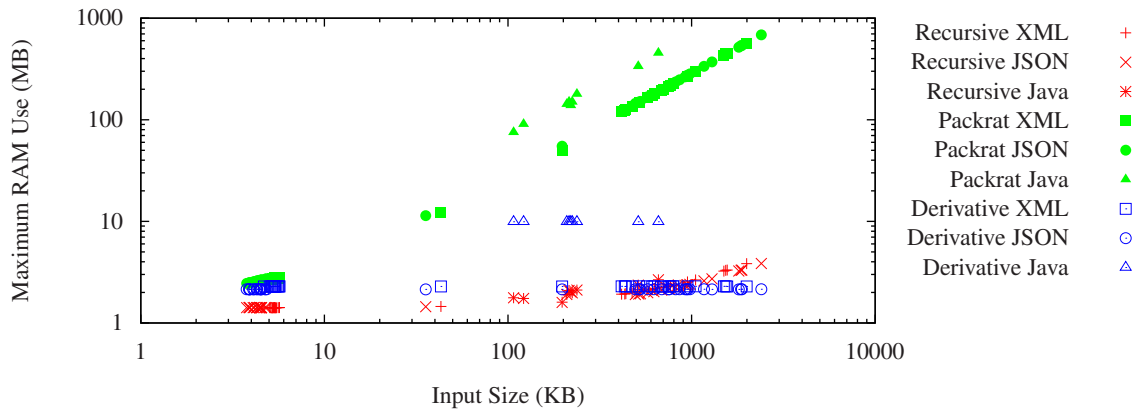Figure 2: Algorithm runtime with respect to input size; lower is better.

Figure 3: Maximum algorithm memory use with respect to input size; lower is better.
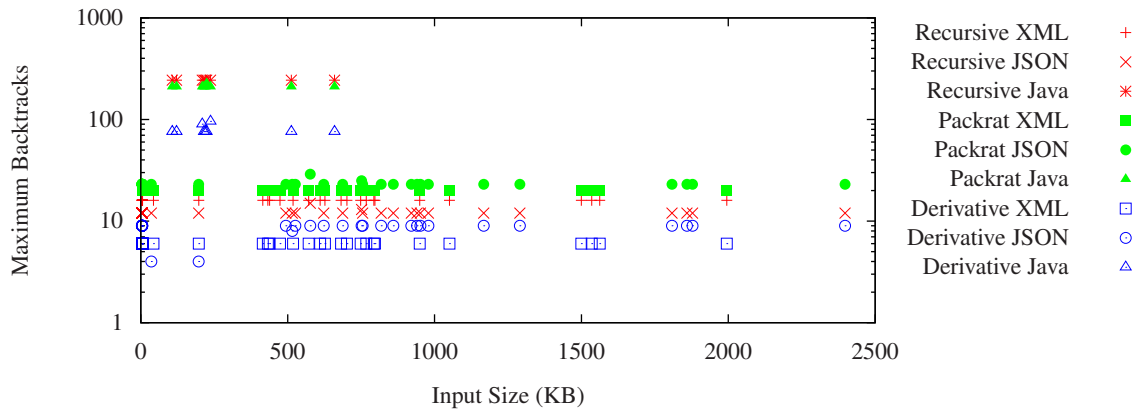

Figure 4: Maximum backtracking performed by each algorithm, plotted against input size.
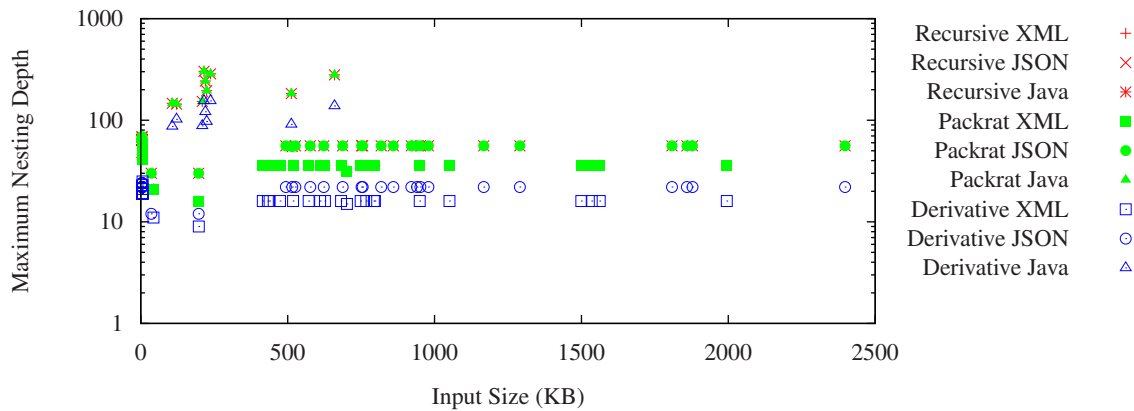

Figure 5: Maximum nesting depth for each algorithm, plotted against input size.

removed as part of the derivative function their nodes are not counted by the instrumentation. Figure 5 shows roughly constant maximum nesting depth for each grammar; the recursive and packrat algorithms have essentially identical results, as should be expected, while the derivative algorithm's lesser nesting depth is due to dynamic compaction of the derivative DAG based on the input – if one branch of an alternation fails the alternation will be replaced with its successful branch in the derivative algorithm, while the alternation expression will still be on the recursive descent stack.

If less well-behaved grammars are considered, the performance results are markedly different. All three parsing algorithms were timed matching the grammar in Figure 1 on strings of the form $a^n c^n$, with a one-minute cutoff for execution time. As predicted, naive recursive descent parsing had exponential runtime growth, with $n = 27$ being the largest instance it could parse inside the time limit. Derivative parsing and packrat both showed linear growth in memory usage. Corresponding to this linear growth in memory usage was a quadratic growth in runtime for the derivative parsing algorithm, exceeding the time bound after $n = 4000$. Packrat parsing was very performant on this simple example, with no instance up to $n = 10000$ reporting any perceptible runtime.

7 Conclusions and future work

This work shows that a polynomial runtime guarantee for PEG parsing incurs significant penalties for many common inputs; the new analysis presented in Section 5.1 provides a theoretical explanation for the good behaviour of recursive descent parsing on a restricted set of inputs, while the experimental results in Section 6 demonstrate the real-world practicality of these restrictions on the input. The derivative parsing algorithm presented in this paper provides a choice of trade-offs to guarantee polynomial runtime; it may be a practical choice in memory-constrained or online systems where the asymptotic increase in memory usage incurred by packrat parsing or input buffering is undesirable.

The algorithm presented in this paper is, formally speaking, a recognition algorithm rather than a parsing algorithm, as it does not produce a parse tree. It should be fairly simple to augment ε expressions with parse tree nodes, composing these nodes as their containing expressions matched or failed. An alternate approach would be to integrate semantic action expressions into the grammar; this should be possible by augmenting each parsing expression with an environment to store local variables. A third option would be to extend the derivative of parsing expressions defined in this paper to a derivative of parsing expression parsers, analogously to the extension defined by Might *et al.* for CFG derivatives [13]. All these extensions are left to future work.

It would also be interesting to attempt to handle left-recursion in derivative parsing with sensible semantics. Ford [6] removes left-recursive rules by grammar transformation, Medeiros *et al.* [12] modify PEG semantics to support left-recursive grammars, and Laurent and Mens [10] use a custom parsing expression to handle left-recursion; one of these approaches could possibly be applied dynamically in the derivative parsing algorithm.

References

[1] Michael D. Adams, Celeste Hollenbeck & Matthew Might (2016): *On the Complexity and Performance of Parsing with Derivatives*. In: *Proceedings of the 37th ACM SIGPLAN Conference on Pro-

gramming Language Design and Implementation, PLDI '16, ACM, New York, NY, USA, pp. 224–236, doi:10.1145/2980983.2908128.

[2] Ralph Becket & Zoltan Somogyi (2008): *DCGs + Memoing = Packrat Parsing but Is It Worth It?*, pp. 182–196. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-77442-6_13.

[3] Jonathan Immanuel Brachthäuser, Tillmann Rendel & Klaus Ostermann (2016): *Parsing with First-class Derivatives*. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, ACM, New York, NY, USA, pp. 588–606, doi:10.1145/3022671.2984026.

[4] Janusz A Brzozowski (1964): *Derivatives of regular expressions*. Journal of the ACM (JACM) 11(4), pp. 481–494, doi:10.1145/321239.321249.

[5] Nils Anders Danielsson (2010): *Total Parser Combinators*. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, ACM, New York, NY, USA, pp. 285–296, doi:10.1145/1932681.1863585.

[6] Bryan Ford (2002): *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*. Master's thesis, Massachusetts Institute of Technology, doi:10.1145/581478.581483.

[7] Bryan Ford (2004): *Parsing expression grammars: a recognition-based syntactic foundation*. In: *ACM SIGPLAN Notices*, 39(1), ACM, pp. 111–122, doi:10.1145/982962.964011.

[8] Fritz Henglein & Ulrik Terp Rasmussen (2017): *PEG Parsing in Less Space Using Progressive Tabling and Dynamic Analysis*. In: *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2017, ACM, New York, NY, USA, pp. 35–46, doi:10.1145/3018882.3018889.

[9] Kimio Kuramitsu (2015): *Packrat Parsing with Elastic Sliding Window*. Journal of Information Processing 23(4), pp. 505–512, doi:10.2197/ipsjjip.23.505.

[10] Nicolas Laurent & Kim Mens (2015): *Parsing Expression Grammars Made Practical*. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, ACM, New York, NY, USA, pp. 167–172, doi:10.1145/2814251.2814265.

[11] Sérgio Medeiros & Roberto Ierusalimschy (2008): *A Parsing Machine for PEGs*. In: *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, ACM, New York, NY, USA, pp. 2:1–2:12, doi:10.1145/1408681.1408683.

[12] Sérgio Medeiros, Fabio Mascarenhas & Roberto Ierusalimschy (2014): *Left recursion in parsing expression grammars*. Science of Computer Programming, doi:10.1016/j.scico.2014.01.013.

[13] Matthew Might, David Darais & Daniel Spiewak (2011): *Parsing with Derivatives: A Functional Pearl*. In: *ACM SIGPLAN Notices*, 46(9), ACM, pp. 189–195, doi:10.1145/2034574.2034801.

[14] Kota Mizushima, Atusi Maeda & Yoshinori Yamaguchi (2010): *Packrat parsers can handle practical grammars in mostly constant space*. In: *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ACM, pp. 29–36, doi:10.1145/1806672.1806679.

[15] Aaron Moss: *Egg*. https://github.com/bruceiv/egg/tree/deriv.

[16] Roman R Redziejowski (2007): *Parsing expression grammar as a primitive recursive-descent parser with backtracking*. Fundamenta Informaticae 79(3-4), pp. 513–524.

[17] Roman R Redziejowski (2016): *Cut points in PEG*. Fundamenta Informaticae 143(1-2), pp. 141–149, doi:10.3233/FI-2016-1308.