arXiv:1405.4841v1 [cs.FL] 19 May 2014

Derivatives of Parsing Expression Grammars

Aaron Moss
a3moss@uwaterloo.ca

May 20, 2014

Abstract

This paper introduces a new memoized derivative parsing algorithm for recognition of parsing expression grammars. The algorithm runs in worst case quartic time and cubic space. However, existing research suggests that due to the limited amount of backtracking and recursion in real-world grammars and input, practical performance may be closer to linear time and constant space; experimental validation of this conjecture is in progress.

1 Introduction

Parsing expression grammars (PEGs) are a parsing formalism introduced by Ford[2]. 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 naïve recursive descent implementation, or linear time and space using the memoized "packrat" algorithm also introduced by Ford[2]. This paper outlines these existing algorithms and introduces a third algorithm based on the context free grammar derivative of Might, Darais, and Spiewak[4], which is in turn based on Brzozowski's regular expression derivative[1]. Analysis of this algorithm reveals that it has quartic worst case time complexity and cubic worst space complexity. However, the results of Mizushima *et al.*[5] regarding the actual amount of backtracking in PEG-based parsers suggest that the real-world performance of this algorithm may approach linear time and constant space. After the asymptotic analysis of this algorithm and speculation about possible practical performance, there is a brief discussion of possible extensions to this work.

2 Parsing Expression Grammars

A parsing expression grammar is similar in concept to a context-free grammar (CFG). Any CFG which is LR(k) can be represented as a PEG[3], but there are some non-context-free languages which may also be represented as parsing expression grammars (for example, $a^n b^n c^n$, which Ford provides a PEG for in [3]).

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 an alphabet Σ, and a set of strings Σ^* from Σ, a parsing expression α is a function $\alpha : \Sigma^* \to \{\mathsf{match}, \mathsf{fail}\} \times \Sigma^*$. I denote the set of nonterminals $\mathcal{N}$, the set of parsing expressions $\mathcal{X}$, and the function which maps 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* which will be parsed. In this paper I will use lowercase Greek letters for parsing expressions ($\varphi \in \mathcal{X}$), uppercase Roman letters for nonterminals ($N \in \mathcal{N}$), and lowercase monospace letters for characters ($\mathsf{c} \in \Sigma$). For a parsing expression φ the language $\mathcal{L}(\varphi)$ is the set of strings matched by φ, formally $\mathcal{L}(\varphi) = \{\mathrm{s} \in \Sigma^* : \exists \mathrm{s}', \varphi(\mathrm{s}) = (\mathsf{match}, \mathrm{s}')\}$. A parsing expression φ is *nullable* if $\mathcal{L}(\varphi) = \Sigma^*$.

The simplest parsing expression is the character literal c, which matches and consumes a c, or fails otherwise. The empty expression ε always matches, consuming no input. Two expressions can be matched in sequence: $\alpha\beta$ matches if both α and β match one after the other, failing otherwise. PEGs also provide an alternation operator: α/β first attempts to match α, then tries β if α fails. Unlike the unordered choice in context free grammars, once a subexpression matches in a PEG, none of the other subexpressions will ever be tried. Specifically, ab/a and a/ab are distinct PEGs, and the ab in the second version will never match, as any string it would match on would have already matched a. The lookahead operator $!\alpha$ never consumes any input, matching if and only 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 sort of repetition can be achieved without a new nonterminal by the repetition operator $\alpha*$; this matching is greedy, so, for instance, $a*\ a$ will never match, since the $a*$ will consume all available a's, leaving none for the a to match. Since repetition operators add no expressive power to PEGs, as $\alpha*$ may always be replaced with $N_{\alpha*} := \alpha N_{\alpha*}/\varepsilon$ for some new nonterminal $N_{\alpha*}$, this paper largely ignores them as syntactic sugar. A small optimization to this technique would be to reduce the number of anonymous nonterminals corresponding to repetition expressions $\alpha*$ by representing equivalent repetition expressions by the same nonterminal; it is not decidable in general whether two parsing expressions recognize the same language, but $(a/b)*$ and $(a/b)*$ obviously do, and should not be represented by distinct nonterminals. Some other common syntactic sugar for PEGs is $\alpha?$ for α/ε, $\alpha+$ for $\alpha\ \alpha*$, and $\&\alpha$ for $!!\alpha$. A typical abbreviation for "any character in Σ" is $\cdot$, and most PEG grammars also include some square-bracket-enclosed character class notation as shorthand for alternation between a number of character literals. Table 1 has formal definitions of each of the basic expressions, omitting the syntactic sugar for brevity.

$$
\begin{aligned}
a(\mathrm{s}) &= \begin{cases} (\mathsf{match}, rest(\mathrm{s})) & \mathtt{a} = first(\mathrm{s}) \\ (\mathsf{fail}, \mathrm{s}) & \text{otherwise} \end{cases} \\
\varepsilon(\mathrm{s}) &= (\mathsf{match}, \mathrm{s}) \\
A(\mathrm{s}) &= (\mathcal{R}(A))(\mathrm{s}) \\
!\alpha(\mathrm{s}) &= \begin{cases} (\mathsf{match}, \mathrm{s}) & \alpha(\mathrm{s}) = (\mathsf{fail}, \mathrm{s}) \\ (\mathsf{fail}, \mathrm{s}) & \text{otherwise} \end{cases} \\
\alpha\beta(\mathrm{s}) &= \begin{cases} (\mathsf{match}, \mathrm{s}'') & \alpha(\mathrm{s}) = (\mathsf{match}, \mathrm{s}') \wedge \beta(\mathrm{s}') = (\mathsf{match}, \mathrm{s}'') \\ (\mathsf{fail}, \mathrm{s}) & \text{otherwise} \end{cases} \\
\alpha/\beta(\mathrm{s}) &= \begin{cases} (\mathsf{match}, \mathrm{s}') & \alpha(\mathrm{s}) = (\mathsf{match}, \mathrm{s}') \\ (\mathsf{match}, \mathrm{s}'') & \alpha(\mathrm{s}) = (\mathsf{fail}, \mathrm{s}) \wedge \beta(\mathrm{s}) = (\mathsf{match}, \mathrm{s}'') \\ (\mathsf{fail}, \mathrm{s}) & \text{otherwise} \end{cases}
\end{aligned}
$$

Table 1: Formal definitions of parsing expressions; expressions bind tighter toward the top of the list.

2.1 Recursive Descent Algorithm

Since parsing expression grammars are a formalization of recursive descent parsers, it should not be surprising that they may be parsed by a direct conversion of the expressions in Table 1 into a recursive algorithm with a function for each nonterminal. This naïve approach is simple and space-efficient, requiring only enough space to store the input string and function call stack (which may be linear in the size of the input). In languages without tail-call optimization, it is common to compute $\alpha*$ as an iterative loop rather than a nonterminal requiring a recursive function call, but this does not change the asymptotic bounds of this method. This naïve method may require exponential time to run; for example, consider a string of the form $\mathtt{a}^n\mathtt{c}^n$ as input to Grammar 1. In this case, the function corresponding to A will be called twice after the first $\mathtt{a}$, each

$$
\begin{aligned}
S &:= A\ !\cdot \\
A &:= a\ A\ b \\
&\quad / \ a\ A\ c \\
&\quad / \ \varepsilon
\end{aligned}
$$

Grammar 1: Grammar that may have exponential runtime using the naïve algorithm.

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 will assume each of the $n - i$ remaining `a`'s is eventually followed by a `b`, fail, and then have to repeat this process assuming the `a`'s are followed by a `c`.

This time bound neglects to account for left-recursive grammars (trivially, $A = A$), which may enter infinite loops. Ford[3] defines a *well-formed* grammar as one which contains no directly or mutually left-recursive rules. This property is structurally checkable, so this paper implicitly assumes that all grammars are well-formed to guarantee a finite time bound for the discussion of naïve and packrat parsers.

2.2 Packrat Parsing

The essential idea of packrat parsing, introduced by Ford in [2], is to memoize the calls to the nonterminal functions from the naïve implementation. If repetition expressions are treated as syntactic sugar for an anonymous nonterminal (as described above) all grammar constructs except for nonterminals can be parsed in time linear in their size; since the size of constructs in the grammar is a constant, these expressions can be parsed in constant time. Furthermore, once a nonterminal has been evaluated at a position it can also be parsed in constant time by reading the result from the memoization table. For any well-formed grammar, a nonterminal will not recursively call itself at the same location in the input string; this means that once any recursively-called subsidiary nonterminals are parsed a nonterminal can be parsed in constant time. Since there are a constant number of nonterminals which may be parsed no more than once for each position in the input string, this algorithm takes linear time. If the string parameters from Table 1 are stored as indices or pointers into the input string, each memoization table entry contains the name of a rule, the index at which it was parsed, and the resulting match or fail and end index, and takes constant space. Since there is at most one entry per grammar rule per string index, this table takes linear space.

2.2.1 Cut Insertion

Mizushima, Maeda, and Yamaguchi[5] show that if you can detect situations in which a packrat parser will never backtrack past its current position then you can save a significant amount of space by discarding the memoization entries before that position. The approach Mizushima *et al.* take is to add a "cut" operator $\uparrow$ which indicates that the current expression will not backtrack to try a later alternative; to motivate this, consider Grammar 2. As can be seen from this grammar, if an `if`, `while`, or `begin` is parsed by S, no later alternative will possibly match; this fact is communicated to the parser by the cut operators after if, $while$, and $begin$. No cut operator is needed after set, as it is the last alternative, thus no other alternative can possibly match.

Cut operators may only sensibly be inserted in expressions of the form $\alpha\ \beta\ /\ \gamma$; in this case a cut operator may be inserted between α and β, where

$$
\begin{aligned}
S := \; & if \uparrow E \; S \; (else \; S)? \\
& / \; while \uparrow E \; S \\
& / \; begin \uparrow S+ \; end \\
& / \; set \; L \; E
\end{aligned}
$$

Grammar 2: Simplified grammar of programming language statements (adapted from [5]), featuring cut operators. Imagine E and L are expressions and lvalues, respectively.

$(\uparrow \beta \; / \; \gamma)\,(\mathrm{s})$ evaluates to $\beta(\mathrm{s})$, discarding the possibility of backtracking to γ. As manual insertion of cut operators can be tedious and error-prone, Mizushima *et al.* also describe a method of automatically inserting cut operators: an expression $\alpha \; \beta \; / \; \gamma$ can be converted to $\alpha \uparrow \beta \; / \; \gamma$ without changing the language of the expression if $\mathcal{L}(\alpha) \cap \mathcal{L}(\gamma) = \emptyset$. Since parsing expressions match prefixes of their input, this condition means α and γ cannot match any common prefixes, *e.g.* $\mathcal{L}(ab/d) \cap \mathcal{L}(a/c) = \mathcal{L}(ab) \neq \emptyset$. Specifically, if γ is nullable then no cut operator can be inserted after α. Unfortunately, as shown by Ford[3], it is undecidable whether the language of a given parsing expression is empty, so automatic cut insertion must rely on a conservative approximation of $\mathcal{L}(\alpha) \cap \mathcal{L}(\gamma) = \emptyset$; Mizushima *et al.* check disjointness of a "first set" of nonterminals which may possibly start a matching parse of the expression. Mizushima *et al.* also describe how to insert cut operators automatically in one case where γ is nullable, namely $(\alpha\beta)*$ (which is equivalent to $N_{(\alpha\beta)*} := \alpha\beta N_{(\alpha\beta)*} \; / \; \varepsilon$) based on a "follow set" of nonterminals which may possibly follow a successful parse of $(\alpha\beta)*$. Both "first" and "follow" sets for parsing expressions were originally described by Redziejowkski[6] analogously to their definitions for traditional top-down parsers.

Mizushima *et al.* tested cut insertion on grammars for Java, XML, and JSON; their results show that typical input strings for such grammars use fairly limited backtracking, and thus can be parsed in roughly constant space by a parser suitably augmented with cut operators. However, their results also show that the approximation of language non-intersection they use is not effective for any grammar where the non-intersection cannot be decided with only one terminal of lookahead, a result which limits the usefulness of automatic cut insertion in grammars which take advantage of the unlimited lookahead capabilities of parsing expressions. An example of a parsing expression which cannot have cut operators automatically inserted into it by the method of Mizushima *et al.* is $(!a \; \cdot)* \; a \; \cdot* \; / \; (!b \; \cdot)* \; b \; \cdot*$; a cut operator could be inserted after the a, but this method (or any other which only considered a fixed number of lookahead nonterminals) could not automatically insert it.

3 Derivative Parsing

This paper presents a third algorithm for recognizing parsing expression grammars, based on the derivative parsing algorithm for context free grammars introduced by Might, Darais, and Spiewak[4]. Essentially, the derivative of a parsing expression φ with respect to a character c is a new parsing expression $d_c(\varphi)$ which recognizes everything after the initial c in $\mathcal{L}(\varphi)$. Formally, $\mathcal{L}(d_c(\varphi)) = \{rest(\mathrm{s}) : \mathrm{s} \in \mathcal{L}(\varphi) \wedge first(\mathrm{s}) = \mathsf{c}\}$. If the specific character is not relevant I may also write the derivative as φ'. The essence of the derivative parsing algorithm is to take repeated derivatives of the original parsing expression for each character in the input string and check for a nullable or failure parser at the end. I will use $d_{c_0 c_1 \cdots c_k}(\varphi)$ as shorthand for $d_{c_k} \circ \cdots \circ d_{c_1} \circ d_{c_0}(\varphi)$, or $\varphi^{(k)}$ for the same when the specific characters are unimportant in context.

To compute derivatives of parsing expressions, I introduce two new parsing expressions to represent failure states: $\varnothing$ and ∞; $\varnothing(\mathrm{s}) = \infty(\mathrm{s}) = (\mathsf{fail}, \mathrm{s})$. $\varnothing$ represents a generic parse failure, while ∞ represents a left-recursive infinite loop, which this algorithm can detect and terminate on. To resolve backtracking choices I also introduce a new expression ℓ_i which signifies a successful match of a lookahead expression.

Derivative parsing essentially tries all possible parses concurrently; since parsing expression grammars are inherently unambiguous they are actually better suited to this approach than context free grammars. Nonetheless, tracking exactly which constructs in the current parse tree can match against or consume the current character is the key problem in this algorithm. To solve this problem I introduce the concept of a *backtracking generation* (or *generation*) as a way to account for backtracking choices in the parsing process. Generation 0 is a straightforward parse where each character is consumed at the lowest possible subexpression in the parse tree, while higher generations represent backtracking decisions that occurred at some earlier point in the parse and have not yet been resolved. I will denote a set of generations with uppercase sans-serif characters, such that $\mathsf{X} = \{x_0, x_1, \cdots x_k\}$, where the indices $0..k$ are assigned from smallest value to largest value. I define two functions *back* and *match* from parsing expressions to sets of backtracking generations; they will be described in more detail below, but essentially $back(\varphi)$ is the set of backtracking generations currently being parsed by φ while $match(\varphi)$ is the subset of $back(\varphi)$ which has successfully matched. The values of *back* and *match* for all expressions are defined in Table 2; some compound expressions such as sequence and alternation need additional bookkeeping information (listed in square brackets) to perform these calculations. Note that *back* and *match* for a nonterminal A are essentially fixed point computations where A acts like an infinite loop ∞ if encountered recursively.

One key idea for compound expressions is translating the backtracking generation sets of their subexpressions into a single space of backtracking generations. To perform this operation I define composition of generation sets as $\mathsf{F} \circ \mathsf{G} = \{f_i : i \in \mathsf{G}\}$. I also define a new map expression $\alpha[\mathsf{A}, m]$ which performs this composition; A is the map to compose, while m is the maximum generation.

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

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

Table 2: *back* and *match* Definitions

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; these annotations allow the separate backtracking generations for α and β to be expressed in a consistent manner as generations of α/β.

Since sequence expressions $\alpha\beta$ encode the "is followed by" relationship, the bulk of the complication of dealing with backtracking must also be handled in sequence expressions. The first type of backtracking that a sequence expression must handle is nullability backtracking, where α is a nullable but non-empty expression, *e.g.* $\alpha\beta = (\delta/\varepsilon)\beta$. This example is basically equivalent to $\delta\beta/\beta$, and is handled similarly to an alternation expression except that because of the longest-match rule for PEGs if some later derivative $\delta^{(k)}$ is also nullable then the second alternative, $\beta^{(k)}$, will be discarded. The other type of backtracking handled by sequence expressions is lookahead backtracking. Each generation in $back(\alpha)$ corresponds to a previous point in the input string where α began to consume no characters (*e.g.* by including a lookahead expression), yet could not yet decide whether to match or fail. For each of these generations, the sequence expression must begin to take derivatives of β so that the proper derivative can be computed if one of those generations eventually matches. If these lookahead follower expressions are nullable they need to account for nullability backtracking as well. Incorporating the bookkeeping for both these lookahead types yields a sequence expression $\alpha\beta[\beta_1, \cdots, \beta_k, \beta_\emptyset, \mathsf{B}_1, \cdots, \mathsf{B}_k, \mathsf{B}_\emptyset, l_1, \cdots, l_k, m]$, where β_i

is the lookahead follower for generation i, $\beta_\emptyset$ is the nullability backtrack, B_i is the mapping set for β_i, l_i is the last generation where β_i was nullable (or 0 for none such), and m is the maximum backtracking generation for the expression.

Might *et al.* discovered that they required certain compaction rules to keep their derivatives at a reasonable size. I also define a set of compaction rules in Table 3 which are necessary to maintain certain invariants about parsing expressions. In practice, computation of these compaction rules would be interleaved with the derivative to avoid unnecessary work, but the rules are separated out here for lucidity.

$$\ell_0 = \varepsilon$$

$$!\alpha = \begin{cases} \varnothing & match(\alpha) \neq \emptyset \\ \ell_1 & \alpha = \varnothing \\ \infty & \alpha = \infty \end{cases}$$

$$\alpha[\mathsf{A}, m] = \begin{cases} \ell_{\mathsf{A} \circ \{0\}} & \alpha = \varepsilon \\ \ell_{\mathsf{A} \circ \{i\}} & \alpha = \ell_i \\ \varnothing & \alpha = \varnothing \\ \infty & \alpha = \infty \\ \alpha & \mathsf{A} = \{0, 1, \cdots, m\} \end{cases}$$

$$\alpha/\beta[\mathsf{A}, \mathsf{B}, m] = \begin{cases} \beta[\mathsf{B}, m] & \alpha = \varnothing \\ \alpha[\mathsf{A}, m] & \beta = \varnothing \vee match(\alpha) \neq \emptyset \\ \infty & \alpha = \infty \end{cases}$$

$$\alpha\beta[\beta_1, \cdots, \beta_k, \beta_\emptyset, \mathsf{B}_1, \cdots, \mathsf{B}_k, \mathsf{B}_\emptyset, l_1, \cdots, l_k, m]$$
$$= \begin{cases} \beta[ngs(\beta, m-1), m] & \alpha = \varepsilon \\ \beta_i[\mathsf{B}_i, m] & \alpha = \ell_i \wedge l_i = 0 \\ \beta_i/\ell_1[\mathsf{B}_i, \{0, l_i\}, m] & \alpha = \ell_i \wedge l_i > 0 \\ \beta_\emptyset[\mathsf{B}_\emptyset, m] & \alpha = \varnothing \\ \infty & \alpha = \infty \end{cases}$$

Table 3: Expression compaction rules

Parsing expression derivatives also need to update the backtracking generation maps of their expressions. This is done via the updating function, which I define as

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

which adds a new backtracking generation to the map P for any expression φ

that has added a new backtracking generation. Since backtracking generations correspond to points in the parse where a new backtracking decision has been deferred, there will be no more than one new backtracking 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 backtracking generations into the backtracking space of an ongoing derivative computation, I introduce the new generation set function, which I define as

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

With all these helper functions in place, the derivative of a parsing expression can be defined as in Table 4. Following convention, end-of-input is represented as a \$ character.

In Table 4, ε' is a failure because there's nothing left of an ε-expression to take a derivative of. Nonterminals are again assumed to be ∞ while calculating their derivatives to break infinite loops in the computation process; this may be necessary even for grammars which have no left-recursion if evaluated using one of the other two algorithms, *e.g.* $A := \ !a \cdot /a/A$. The $\beta^\dagger$ special case in the sequence expression derivative is because the \$ end-of-input pseudo-character breaks the property that if α consumes a character then β (or its backtracking derivatives) do not have to. The other cases cover both the nullability and lookahead backtracking discussed above.

Given this definition of derivative, the *parse* function defined in Table 5 will compute the parsing expression defined in Table 1, though without returning the unparsed portion of the string on match or fail. 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 practical problem, as any more parsing required could be performed by wrapping the start expression in another expression.

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 expressions φ and their derivatives $d_{c_i}(\varphi)$ is constructed for each recursion of *parse* and expressions check this memoization table before recomputing their derivative, no expression will compute its derivative more than once. This memoization table can be discarded at the end of the recursion of *parse*, as the next set of derivatives will in general be of different expressions with respect to a different character. If all the instances of the same nonterminal are references to the same in-memory object and compound expressions like $\alpha\beta$ and α/β store their subexpressions α and β by reference rather than by value then the memory address an expression can be used as its key into the memoization table and there will be no more than one instance of each subexpression in any parsing expression derivative. This bounds the increase in derivative size and computa-

$$a' = \begin{cases} \varepsilon & c = a \\ \varnothing & \text{otherwise} \end{cases}$$

$$\varepsilon' = \begin{cases} \varepsilon & c = \$ \\ \varnothing & \text{otherwise} \end{cases}$$

$$\ell_i' = \ell_i$$

$$\varnothing' = \varnothing$$

$$\infty' = \infty$$

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

$$(!\alpha)' = !(\alpha')$$

$$\alpha[\mathsf{A}, m]' = \alpha'[\mathsf{A}' = up(\mathsf{A}, \alpha, m), m' = \max(m, \mathsf{A}')]$$

$$\alpha/\beta[\mathsf{A}, \mathsf{B}, m]' = \alpha'/\beta'[\mathsf{A}' = up(\mathsf{A}, \alpha, m), \mathsf{B}' = up(\mathsf{B}, \beta, m), m' = \max(m, \mathsf{A}', \mathsf{B}')]$$

$$\begin{aligned} &\alpha\beta[\beta_1, \cdots, \beta_k, \beta_\emptyset, \mathsf{B}_1, \cdots, \mathsf{B}_k, \mathsf{B}_\emptyset, l_1, \cdots, l_k, m]' \\ &\quad = \alpha'\beta^\dagger[\beta_1^\dagger, \cdots, \beta_{k'}^\dagger, \beta_\emptyset^\dagger, \mathsf{B}_1', \cdots, \mathsf{B}_{k'}', \mathsf{B}_\emptyset', l_1', \cdots, l_{k'}', m'] \end{aligned}$$

where

$$\beta^\dagger = \begin{cases} \beta' & c = \$ \wedge \alpha' \in \{\varepsilon, \ell_{k+1}\} \\ \beta & \text{otherwise} \end{cases}$$

$$k' = \begin{cases} k+1 & \max(back(\alpha')) > \max(back(\alpha)) \\ k & \text{otherwise} \end{cases}$$

$$\beta_{1 \le i \le k}^\dagger = \beta_i'$$

$$\beta_{k+1}^\dagger = \beta$$

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

$$\mathsf{B}_{1 \le i \le k}' = up(\mathsf{B}_i, \beta_i, m)$$

$$\mathsf{B}_{k+1}' = ngs(\beta, m)$$

$$\mathsf{B}_\emptyset' = \begin{cases} ngs(\beta, m) & 0 \in match(\alpha') \\ up(\mathsf{B}_\emptyset, \beta_\emptyset, m) & \text{otherwise} \end{cases}$$

$$l_{k+1} = 0$$

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

$$m' = \max(m, \mathsf{B}_i', \mathsf{B}_\emptyset, l_i') \text{ for any of these used}$$

Table 4: Definition of $d_c(\varphi)$; φ' abbreviates $d_c(\varphi)$.

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

Table 5: Derivative parsing function

tion time, allowing this parsing algorithm to achieve the time and space bounds proved in Section 4.

The treatment of backtracking generations can also be improved by eliminating information that will never be used. For instance, if a generation i is not present in $back(\alpha)$, then the sequence expression $\alpha\beta$ does not need to keep β_i in its bookkeeping information, since α will never reduce to ℓ_i, which would select it. Similarly, in the map expression $\alpha[\mathsf{A} = \{a_0, a_1, \cdots, a_k\}, m]$, we don't actually need 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 from a function from its indices to its values to a partial function from its indices to its values, keeping only the relevant mappings. 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.

4 Analysis

This algorithm runs in time linear in the size of the input times the cost of a derivative computation, and requires as much space as the largest derivative computed. The cost of a derivative computation $d(\varphi)$ is linear in the size of φ, but since φ is a memoized directed graph, it may re-use the computation of some sub-expressions of φ. To account for this, I define $^{\#}(\varphi)$ to be the number of *unique* subexpressions of φ (including φ itself). Now, 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 time than $!\alpha$). To formalize this idea I note the asymptotic bounds on the size of $|\varphi|$ in Table 6; this size is expressed in terms of b, the maximum number of backtracking generations currently being parsed. Each backtrack set is obviously $O(b)$, while the bookkeeping information in a sequence expression keeps information for $O(b)$ generations.

It is evident from the expression compaction rules in Table 3 that no expression compaction rule increases the size of its expression or adds new expression nodes. In fact, nearly all of the compaction rules strictly reduce the size of their expression node, and many discard references to subexpressions, further

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

Table 6: Size $|\varphi|$ of an expression φ

shrinking the overall size of the expression. Since none of the derivative cases in Table 4 adds more than one new backtracking generation, b is linear in the number of derivatives taken. Having bounded the size of a single expression node, I will now show 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 which adds expression nodes to the larger expression, replacing the nonterminal expression A with $d(\mathcal{R}(A))$. But since $\mathcal{R}(A)$ is statically defined, $^{\#}(\mathcal{R}(A))$ is a constant, and since derivatives are memoized the expression nodes comprising $d(\mathcal{R}(A))$ will only be added once per derivative step. $d(\mathcal{R}(A))$ may expand other nonterminal expressions to their respective derivatives, but at no point in this process will A be recursively expanded, since due to the left-recursion elimination rule recursive expansions of A are replaced with ∞, which does not add any more expression nodes. According to the preceding argument then, 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.

It follows directly that the space required by the derivative parsing algorithm is $O(nb^2)$ (or $O(n^3)$). As each derivative operation must traverse one of these expressions, the worst case running time for this algorithm is $O(n^2b^2)$ ($O(n^4)$). This polynomial bound guarantees quicker worst case execution than the naïve recursive descent algorithm, but does not match the time bounds of packrat parsing. However, due to the properties of this bound, the practical performance of this algorithm may be similar to a packrat parser with perfect cut insertion. The first justification of this assertion is that it is a well-known fact that the grammar nesting depth of real-world inputs such as source code and structured data formats is typically bounded by a small constant, so the factor of n in these bounds corresponding to nodes added from evaluating grammar rules can be ignored on such inputs. The second justification is from the results of Mizushima *et al.*, which suggest that there is very little backtracking in practical grammars, and a grammar sufficiently augmented with cut operators will use a roughly constant amount of memory. If the backtracking factor b can also be bounded by a small constant, then the derivative parsing algorithm will run in constant space and linear time on such a well-formed grammar and input, matching the

performance achieved by the cut-augmented packrat parser of Mizushima *et al.* Derivative parsing has the further advantages that it does not require manual cut insertion for best performance and that it can trim its backtracking state dynamically based on the input string rather than just statically based on the grammar.

5 Future Work

Given the cubic factor of time and space between the worst case bounds for derivative parsing and the conjectured real-world performance, this algorithm needs to be implemented and tested to determine its practicality; this testing is in progress, but has not produced reportable results yet. A direct comparison to a packrat parser augmented with cut insertion is essential to such an experiment, and a comparison to a parser using a more traditional LR parsing approach to recognize the same language would also provide insight into the practical utility of PEGs as a grammar formalism. Derivative parsing is naturally concurrent, so it would be interesting to design and test a parallel implementation of this algorithm as well.

Another extension to this work that should be investigated is the addition of semantic actions to the derivative parsing algorithm; the algorithm presented only recognizes whether or not an input string is in the grammar, while a practical parser generator can also build a parse tree or perform other computation while matching a grammar. Implementing semantic actions should be possible with the augmentation of each parsing expression with an environment to store local variables and a reference to the expression's return value in the enclosing environment. These environments would have to be cloned in each alternation or sequence backtracking choice, but copy on write semantics may be able to be implemented to reduce the number of extraneous copies.

References

[1] Janusz A Brzozowski, *Derivatives of regular expressions*, Journal of the ACM (JACM) **11** (1964), no. 4, 481–494.

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

[3] ———, *Parsing expression grammars: a recognition-based syntactic foundation*, ACM SIGPLAN Notices, vol. 39, ACM, 2004, pp. 111–122.

[4] Matthew Might, David Darais, and Daniel Spiewak, *Parsing with derivatives: A functional pearl*, ACM SIGPLAN Notices, vol. 46, ACM, 2011, pp. 189–195.

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

[6] Roman R Redziejowski, *Applying classical concepts to parsing expression grammar*, Fundamenta Informaticae **93** (2009), no. 1, 325–336.