Simplified Parsing Expression Derivatives

Aaron Moss^(✉)

University of Portland, Portland, USA
mossa@up.edu

Abstract. This paper presents a new derivative parsing algorithm for parsing expression grammars; this new algorithm is both simpler and faster than the existing parsing expression derivative algorithm presented by Moss [12]. This new algorithm improves on the worst-case space and runtime bounds of the previous algorithm by a linear factor, as well as decreasing runtime by about half in practice.

Keywords: Parsing · Parsing expression grammar · Derivative parsing

1 Introduction

A derivative parsing algorithm for parsing expression grammars (PEGs) was first published by Moss [12]; this paper presents a simplified and improved algorithm, as well as a practical comparison of the two algorithms both to each other and to other PEG parsing methods. This new algorithm preserves or improves the performance bounds of the earlier algorithm, trimming a linear factor off the worst-case time and space bounds, while preserving the linear time and constant space bounds for the class of “well-behaved” inputs defined in [12].

2 Parsing Expression Grammars

Parsing expression grammars are a language formalism similar in power to the more familiar context-free grammars (CFGs). PEGs are a formalization of recursive-descent parsing with limited backtracking and infinite lookahead; Fig. 1 provides definitions of the fundamental parsing expressions. a is a *character literal*, matching and consuming a single character of input; ε is the *empty expression* which always matches without consuming any input, while $\emptyset$ is the *failure expression*, which never matches. A is a *nonterminal*, which is replaced by its corresponding parsing expression $\mathcal{R}(A)$ to provide recursive structure in the formalism. The *negative lookahead expression* $!\alpha$ provides much of the unique power of PEGs, matching only if its subexpression α does not match, but consuming no input¹. The *sequence expression* $\alpha\beta$ matches α followed by β , while the *alternation expression* α/β matches either α or β . Unlike the unordered choice in CFGs, if its first alternative α matches, an alternation expression never backtracks to attempt its second alternative β ; this *ordered choice* is responsible for the unambiguous nature of PEG parsing.

¹ The *positive lookahead expression* $\&\alpha$ can be expressed as $!!\alpha$.

$$\begin{aligned}
a(s) &= \begin{cases} s' & s = a s' \\ \text{fail} & \text{otherwise} \end{cases} & A(s) &= (\mathcal{R}(A))(s) \\
\varepsilon(s) &= s & \alpha\beta(s) &= \begin{cases} \beta(\alpha(s)) & \alpha(s) \neq \text{fail} \\ \text{fail} & \text{otherwise} \end{cases} \\
\emptyset(s) &= \text{fail} \\
!\alpha(s) &= \begin{cases} s & \alpha(s) = \text{fail} \\ \text{fail} & \text{otherwise} \end{cases} & \alpha/\beta(s) &= \begin{cases} \alpha(s) & \alpha(s) \neq \text{fail} \\ \beta(s) & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 1. Formal definitions of parsing expressions; $\mathcal{R}(A)$ is the expansion of A

Parsing expressions are functions that recognize prefixes of strings, producing either the unconsumed suffix of a match, or `fail` on failure. The *language* $\mathcal{L}(\varphi)$ of a parsing expression φ over strings from an alphabet Σ is the set of strings *matched* by φ ; precisely, $\mathcal{L}(\varphi) = \{s \in \Sigma^* : \exists s' \in \Sigma^*, \varphi(s) = s'\}$. This paper uses the notation ϵ for the empty string (distinct from the empty expression ε) and $s[i]$ for the suffix $s_i s_{i+1} \cdots s_{n-1}$ of some string $s = s_0 s_1 \cdots s_{n-1}$.

2.1 Related Work

A number of recognition algorithms for parsing expression grammars have been presented in the literature, though none have combined efficient runtime performance with good worst-case bounds. Ford [4] introduced both the PEG formalism and two recognition algorithms: *recursive descent* (a direct translation of the functions in Fig. 1) and *packrat* (memoized recursive descent). The recursive descent algorithm has exponential worst-case runtime, though it behaves well in practice (as shown in Sect. 6); packrat improves the runtime bound to linear, but at the cost of best-case linear space usage. Ford [5] also showed that there exist PEGs to recognize non-context-free languages (*e.g.* $a^n b^n c^n$), and conjectured that some context-free languages exist for which there is no PEG. Mizushima *et al.* [11] have demonstrated the use of manually-inserted “cut operators” to trim memory usage of packrat parsing to a constant, while maintaining the asymptotic worst-case bounds; Kuramitsu [8] and Redziejewski [14] have built modified packrat parsers that use heuristic table-trimming mechanisms to achieve similar real-world performance without manual grammar modifications, but which sacrifice the polynomial worst-case runtime. Medeiros and Ierusalimsky [9] have developed a parsing machine for PEGs, similar in concept to a recursive descent parser, but somewhat faster in practice. Henglein and Rasmussen [7] have proved linear worst-case time and space bounds for their progressive tabular parsing algorithm, with some evidence of constant space usage in practice for a simple JSON grammar, but their work lacks empirical comparisons to other algorithms.

Moss [12] and Garnock-Jones *et al.* [6] have developed derivative parsing algorithms for PEGs. This paper extends the work of Moss, improving the theoretical quartic time and cubic space bounds by a linear factor each, and halving runtime in practice. Garnock-Jones *et al.* do not include empirical performance results for their work, but their approach elegantly avoids defining new parsing expressions through use of a *nullability combinator* to represent lookahead followers as later alternatives of an alternation expression.

3 Derivative Parsing

Though the backtracking capabilities of PEGs are responsible for much of their expressive power and ease-of-use, backtracking is also responsible for the worst-case resource bounds of existing algorithms. Recursive-descent parsing uses exponential time in the worst case to perform backtracking search, while packrat parsing trades this worst-case time for high best-case space usage. Derivative parsing presents a different trade-off, with low common-case memory usage paired with a polynomial time bound. A derivative parsing approach pursues all backtracking options concurrently, eliminating the repeated backtracking over the same input characteristic of worst-case recursive-descent, but also discarding bookkeeping information for infeasible options, saving space relative to packrat.

The essential idea of derivative parsing, first introduced by Brzozowski [3], is to iteratively transform an expression into an expression for the “rest” of the input. For example, given $\gamma = \text{foo}/\text{bar}/\text{baz}$, $d_b(\gamma) = \text{ar}/\text{az}$, the suffixes that can follow **b** in $\mathcal{L}(\gamma)$. After one derivative, the first character of the input has been consumed, and the grammar mutated to account for this missing character. Once repeated derivatives have been taken for every character in the input string, the resulting expression can be checked to determine whether or not it represents a match, *e.g.* $d_z \circ d_a \circ d_b(\gamma) = \varepsilon$, a matching result. Existing work shows how to compute the derivatives of regular expressions [3], context-free grammars [10], and parsing expression grammars [6, 12]. This paper presents a simplified algorithm for parsing expression derivatives, as well as a formal proof of the correctness of this algorithm, an aspect lacking from the earlier presentations.

The difficulty in designing a derivative parsing algorithm for PEGs is simulating backtracking when the input must be consumed at each step, with no ability to re-process earlier input characters. Consider $!(ab)a$; ab and a must be parsed concurrently, and an initial match of a must be reversed if ab later matches. Alternations introduce further complications; consider $!(ab)/a!c$: the final a must be parsed concurrently with $!(ab)$, but also “held back” until after the a in $a!c$ has been matched. To track the connections among such backtracking choices, Moss [12] used a system of “backtracking generations” to label possible backtracking options for each expression, as well as a complex mapping algorithm to translate the backtracking generations of parsing expressions to the corresponding generations of their parent expressions. The key observation of

the simplified algorithm presented here is that an index into the input string is sufficient to label backtracking choices consistently across all parsing expressions.

Typically [3, 10, 12], the derivative $d_c(\varphi)$ is a function from an expression φ and a character $c \in \Sigma$ to a derivative expression. Formally, $\mathcal{L}(d_c(\varphi)) = \{\mathbf{s} \in \Sigma^* : c\mathbf{s} \in \mathcal{L}(\varphi)\}$. This paper defines a derivative $d_{c,i}(\varphi)$, adding an index i for the current location in the input. This added index is used as a label to connect backtracking decisions across derivative subexpressions by annotation of certain parsing expressions. A sequence expression $\alpha\beta$ must track possible indices where α may have stopped consuming characters and β began to be parsed; to this end, $\alpha\beta$ is annotated with a list of *lookahead followers* $[\beta_{i_1} \cdots \beta_{i_k}]$, where β_{i_j} is the repeated derivative of β starting at each index i_j where α may have stopped consuming characters. To introduce this backtracking, ε and $!\alpha$, neither of which consume any characters, become ε_j , a match at index j , and $!_j\alpha$, a lookahead expression at index j . These annotated expressions are formally defined in Fig. 2; note that they produce either a string or fail under the same conditions as their equivalents in Fig. 1. Considered in isolation these extensions appear to introduce a dependency on the string $\mathbf{s}$ into the expression definition (given that $\mathbf{s}[k]$ is a suffix of $\mathbf{s}[j]$), but within the context of the derivative parsing algorithm any ε_j or $!_j$ must be in the α subexpression of a sequence expression $\alpha\beta[\beta_{i_1} \cdots \beta_{i_k}]$ and paired with a corresponding β_j lookahead follower such that $\beta(\mathbf{s}[j]) = \beta_j(\mathbf{s}[k])$, eliminating the dependency. Figure 3 defines a *normalization function* $\langle \bullet \rangle_i$ to annotate parsing expressions with their indices; derivative parsing of φ starts by taking $\langle \varphi \rangle_0$.

$$\begin{aligned} \varepsilon_j(\mathbf{s}[k]) &= \mathbf{s}[j] \\ !_j\alpha(\mathbf{s}[k]) &= \begin{cases} \mathbf{s}[j] & \alpha(\mathbf{s}[k]) = \text{fail} \\ \text{fail} & \text{otherwise} \end{cases} \\ \alpha\beta[\beta_{i_1} \cdots \beta_{i_k}](\mathbf{s}[k]) &= \alpha\beta(\mathbf{s}[k]) \end{aligned}$$

Fig. 2. Formal definitions of added parsing expressions

$$\begin{aligned} \langle a \rangle_i &= a & \langle A \rangle_i &= \langle \mathcal{R}(A) \rangle_i \\ \langle \varepsilon \rangle_i &= \varepsilon_i & \langle \alpha\beta \rangle_i &= \begin{cases} \langle \alpha \rangle_i \beta[\beta_i = \langle \beta \rangle_i] & \epsilon \in \mathcal{L}(\alpha) \\ \langle \alpha \rangle_i \beta[] & \text{otherwise} \end{cases} \\ \langle \emptyset \rangle_i &= \emptyset & \langle !\alpha \rangle_i &= !_i \langle \alpha \rangle_i \\ \langle !\alpha \rangle_i &= !_i \langle \alpha \rangle_i & \langle \alpha/\beta \rangle_i &= \langle \alpha \rangle_i / \langle \beta \rangle_i \end{aligned}$$

Fig. 3. Definition of normalization function

Expressions that are known to always match their input provide opportunities for short-circuiting a derivative computation. For instance, if ν is an expression that is known to match, ν/β never tries the β alternative, while $!\nu$ always

fails, allowing these expressions to be replaced by the simpler ν and $\emptyset$, respectively. A similar optimization opportunity arises when expressions that have stopped consuming input are later invalidated; the augmented sequence expression $\alpha\beta[\beta_{i_1} \cdots \beta_{i_k}]$ keeps an ongoing derivative β_j of β for each start position j that may be needed, so discarding unreachable β_j is essential for performance. Might *et al.* [10] dub this optimization “compaction” and demonstrate its importance to derivative performance; this work includes compaction in the derivative step based on functions *back* and *match* defined in Fig. 4 over normalized parsing expressions. By these definitions, based on [12], *back*(φ) is the set of indices where φ may have stopped consuming input, while *match*(φ) is the set of indices where φ matched. Note that $|\text{match}(\varphi)| \leq 1$ and the definition of *match*(α/β) depends on the invariant that the β alternative is discarded if α matches.

$$\begin{array}{ll}
\text{back}(a) = \{\} & \text{match}(a) = \{\} \\
\text{back}(\varepsilon_i) = \{i\} & \text{match}(\varepsilon_i) = \{i\} \\
\text{back}(\emptyset) = \{\} & \text{match}(\emptyset) = \{\} \\
\text{back}(!_i\alpha) = \{i\} & \text{match}(!_i\alpha) = \{\} \\
\text{back}(\alpha\beta[\beta_{i_1} \cdots \beta_{i_k}]) & \text{match}(\alpha\beta[\beta_{i_1} \cdots \beta_{i_k}]) \\
= \cup_{j \in [i_1 \cdots i_k]} \text{back}(\beta_j) & = \cup_{j \in \text{match}(\alpha)} \text{match}(\beta_j) \\
\text{back}(\alpha/\beta) = \text{back}(\alpha) \cup \text{back}(\beta) & \text{match}(\alpha/\beta) = \text{match}(\beta)
\end{array}$$

Fig. 4. Definitions of *back* and *match*

With these preliminaries established, the derivative is defined in Fig. 5. The derivative consumes character literals, while preserving ε_j matches and $\emptyset$ failures. To a first approximation, the derivative distributes through lookahead and alternation, though match and failure results trigger expression simplification. The bulk of the work done by the algorithm is in the sequence expression $\alpha\beta$ derivative. At a high level, the sequence derivative takes the derivative of α , then updates the appropriate derivatives of β , selecting one if α matches. Any index j in $\text{back}(d_{c,i}(\alpha))$ where α may have stopped consuming input needs to be paired with a corresponding backtrack follower β_j ; introducing a new follower β_i involves a normalization operation. Testing for a match at end-of-input is traditionally [3, 6, 10] handled in derivative parsing with a nullability combinator δ which reduces the grammar to ε or $\emptyset$; this work uses the derivative with respect to an end-of-input character $\# \notin \Sigma$ to implement this combinator. As such, if α matches at end-of-input, $d_{n,\#}(\langle\beta\rangle_n)$ must also be evaluated. As in previous work [10, 12], $\langle\bullet\rangle_i$, $d_{c,i}$, *back*, and *match* are all memoized for performance.

The derivative with respect to a character can be extended to the derivative with respect to a string $\mathbf{s} = \mathbf{s}_1\mathbf{s}_2 \cdots \mathbf{s}_n$ by repeated application: $d_{\mathbf{s},i}(\varphi) = (d_{s_n,i+n} \circ d_{s_{n-1},i+n-1} \circ \cdots \circ d_{s_1,i+1})(\varphi)$. After augmentation with an initial normalization step and final end-of-input derivative, the overall derivative parsing

$$\begin{aligned}
d_{c,i}(a) &= \begin{cases} \varepsilon_i & c = \mathbf{a} \\ \emptyset & \text{otherwise} \end{cases} \\
d_{c,i}(\varepsilon_j) &= \varepsilon_j \\
d_{c,i}(\emptyset) &= \emptyset \\
d_{c,i}(!_j \alpha) &= \begin{cases} \emptyset & \text{match}(d_{c,i}(\alpha)) \neq \{\} \\ \varepsilon_j & d_{c,i}(\alpha) = \emptyset \\ !_j d_{c,i}(\alpha) & \text{otherwise} \end{cases} \\
d_{c,i}(\alpha \beta[\beta_{i_1} \cdots \beta_{i_k}]) &= \begin{cases} \emptyset & d_{c,i}(\alpha) = \emptyset \\ \langle \beta \rangle_i & d_{c,i}(\alpha) = \varepsilon_i \wedge c \neq \# \\ d_{c,i}(\langle \beta \rangle_i) & d_{c,i}(\alpha) = \varepsilon_i \wedge c = \# \\ d_{c,i}(\beta_j) & d_{c,i}(\alpha) = \varepsilon_j \wedge j < i \\ d_{c,i}(\alpha) \beta[\beta_j^\dagger : j \in \text{back}(d_{c,i}(\alpha))] & \text{otherwise, where} \\ & \beta_i^\dagger = \langle \beta \rangle_i, \\ & \beta_j^\dagger = d_{c,i}(\beta_j), j < i \end{cases} \\
d_{c,i}(\alpha/\beta) &= \begin{cases} d_{c,i}(\beta) & d_{c,i}(\alpha) = \emptyset \\ d_{c,i}(\alpha) & d_{c,i}(\beta) = \emptyset \vee \text{match}(d_{c,i}(\alpha)) \neq \{\} \\ d_{c,i}(\alpha)/d_{c,i}(\beta) & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 5. Definition of derivative step; # is end-of-input

algorithm is then $\varphi^{(n)} = d_{\#,n+1} \circ d_{\mathbf{s},0}(\langle \varphi \rangle_0)$. If $\varphi^{(n)} = \varepsilon_j$, then $\varphi(\mathbf{s}) = \mathbf{s}[j]$, otherwise $\varphi(\mathbf{s}) = \text{fail}$. As an example, see Fig. 6.

4 Correctness

There is insufficient space in this paper to include a formal proof of the correctness of the presented algorithm. The author has produced such a proof, however; the general approach is outlined here.

$$\begin{aligned}
\gamma &= (!_0(ab)/a!c)a \\
\langle \gamma \rangle_0 &= (!_0(ab[])/a!c[])a[\beta_0 = a] \\
d_{a,1}(\langle \gamma \rangle_0) &= (!_0b/_1c)a[\beta_0 = \varepsilon_1, \beta_1 = a] \\
d_{c,2} \circ d_{a,1}(\langle \gamma \rangle_0) &= \varepsilon_1 \quad [\text{Note: } \beta_0 \text{ from } !_0b \text{ success}] \\
d_{\#,3} \circ d_{c,2} \circ d_{a,1}(\langle \gamma \rangle_0) &= \varepsilon_1
\end{aligned}$$

Fig. 6. Derivative execution example on string **ac**

The proof makes extensive use of structural induction, thus it must also show that such induction terminates when applied to recursively-expanded non-terminals. If evaluation of a parsing expression involves a left-recursive call to a nonterminal, this evaluation never terminates; as such, left-recursive grammars are generally excluded from consideration. Ford [5, § 3.6] introduced the notion that a parsing expression is *well-formed* if it does not occur anywhere in its own recursive left-expansion or have any subexpression that does; Fig. 7 formalizes the immediate left-expansion LE and the recursive left-expansion LE^+ consistently with Ford's definition. The normalization step presented in this paper expands nonterminals left-recursively, eliminating recursive structure from the parsing expressions considered by the derivative algorithm; this expansion is safe for well-formed grammars.

$$\begin{aligned}
LE(a) &= \{\} & LE(!\alpha) &= \{\alpha\} \\
LE(\varepsilon) &= \{\} & LE(!_j \alpha) &= \{\alpha\} \\
LE(\varepsilon_j) &= \{\} & LE(\alpha\beta) &= \begin{cases} \{\alpha, \beta\} & \epsilon \in \mathcal{L}(\alpha) \\ \{\alpha\} & \text{otherwise} \end{cases} \\
LE(\emptyset) &= \{\} & & \\
LE(A) &= \{\mathcal{R}(A)\} & LE(\alpha/\beta) &= \{\alpha, \beta\} \\
LE(\alpha\beta[\beta_{i_1} \cdots \beta_{i_k}]) &= \{\alpha, \beta_{i_1}, \cdots \beta_{i_k}\} \\
LE^+(\varphi) &= LE(\varphi) \cup_{\gamma \in LE(\varphi)} LE^+(\gamma)
\end{aligned}$$

Fig. 7. Definition of LE left-expansion function and its transitive closure LE^+ ; LE computed by iteration to a fixed point.

To prove the equivalence of derivative parsing with recursive descent, it must be shown that normalization does not change the semantics of a parsing expression, that the derivative step performs the expected transformation of the language of an expression, and that the end-of-input derivative correctly implements the behavior of an expression on the empty string. In each of these cases, the proof proceeds by treating the relevant parsing expressions as functions over their input and proving that they produce equivalent results.

Proof of correctness of the derivative step depends on a number of invariant properties of the normalized parsing expressions (*e.g.* there is a lookahead follower β_j in $\alpha\beta[\beta_{i_1} \cdots \beta_{i_k}]$ for every ε_j that may arise from derivatives of α); these properties must be shown to be established by the $\langle \bullet \rangle_i$ function and maintained by $d_{c,i}$. Other lemmas needed to support the proof describe the dynamic behavior of the derivative algorithm (*e.g.* $match(\varphi)$ implies that the derivative of φ eventually becomes a ε_j success result).

Without appealing to a formal proof of correctness, it should be noted that the experimental results in Sect. 6 demonstrate successful matching of a large number of strings, and thus a low (possibly zero) false-negative rate for the

derivative algorithm; further automated correctness tests are available with the source distribution [13].

5 Analysis

In [12], Moss demonstrated the polynomial worst-case space and time of his algorithm with an argument based on bounds on the depth and fanout of the DAG formed by his derivative expressions. These bounds, cubic space and quartic time, were improved to constant space and linear time for a broad class of “well-behaved” inputs with constant-bounded backtracking and depth of recursive invocation. This paper includes a similar analysis of the algorithm presented here, improving the worst-case bounds of the previous algorithm by a linear factor, to quadratic space and cubic time, while maintaining the optimal constant space and linear time bounds for the same class of “well-behaved” inputs.

For an input string of length n , the algorithm runs $O(n)$ derivative steps; the cost of each derivative step $d_{c,i}(\varphi)$ is the sum of the cost of the derivative algorithm in Fig. 5 on each expression node in the recursive left-expansion LE^+ of φ . Since by convention the size of the grammar is a constant, all operations on any expression γ from the original grammar (particularly $\langle \gamma \rangle_i$) run in constant time and space. It can be observed from the derivative step and index equations in Figs. 5 and 4 that once the appropriate subexpression derivatives have been calculated, the cost of a derivative step on a single expression node δ is proportional to the size of the immediate left-expansion of δ , $LE(\delta)$. Let b be the maximum $|LE(\delta)|$ over all $\delta \in LE^+(\varphi)$; by examination of Fig. 7, $|LE(\delta)|$ is bounded by the number of backtracking followers β_{i_j} in the annotated sequence expression. Since no more than one backtracking follower may be added per derivative step, $b \in O(n)$. Assuming $\langle \bullet \rangle_i$ is memoized for each i , only a constant number of expression nodes may be added to the expression at each derivative step, therefore $|LE^+(\varphi)| \in O(n)$. By this argument, the derivative parsing algorithm presented here runs in $O(n^2)$ worst-case space and $O(n^3)$ worst-case time, improving the previous space and time bounds for derivative parsing of PEGs by a linear factor each. This linear improvement over the algorithm presented in [12] is due to the new algorithm only storing $O(b)$ backtracking information in sequence nodes, rather than $O(b^2)$ as in the previous algorithm.

In practical use, the linear time and constant space results presented in [12] for inputs with constant-bounded backtracking and grammar nesting (a class that includes most source code and structured data) also hold for this algorithm. If b is bounded by a constant rather than its linear worst-case, the bounds discussed above are reduced to linear space and quadratic time. Since b is a bound on the size of $LE(\varphi)$, it can be seen from Fig. 7 that this is really a bound on sequence expression backtracking choices, which existing work including [12] has shown is often bounded by a constant in practical use.

Given that the bound on b limits the fanout of the derivative expression DAG, a constant bound on the depth of that DAG implies that the overall size of the DAG is similarly constant-bounded. Intuitively, the bound on the

depth of the DAG is a bound on recursive invocations of a nonterminal by itself, applying a sort of “tail-call optimization” for right-recursive invocations such as $R_{\alpha^*} := \alpha R_{\alpha^*} / \varepsilon$. The conjunction of both of these bounds defines the class of “well-behaved” PEG inputs introduced by Moss in [12], and by the constant bound on derivative DAG size this algorithm also runs in constant space and linear time on such inputs.

6 Experimental Results

In addition to being easier to implement than the previous derivative parsing algorithm, the new parsing expression derivative also has superior performance.

To test this performance, the simplified parsing expression derivative (SPED) algorithm was compared against the parser-combinator-based recursive descent (Rec.) and packrat (Pack.) parsers used in [12], as well as the parsing expression derivative (PED) implementation from that paper. The same set of XML, JSON, and Java inputs and grammars used in [12] are used here; the inputs originally come from [11]. Code and test data are available online [13]. All tests were compiled with g++ 6.2.0 and run on a Windows system with 8 GB of RAM, a 2.6 GHz processor, and SSD main storage.

Figure 8 shows the runtime of all four algorithms on all three data sets, plotted against the input size; Fig. 9 shows the memory usage of the same runs, also plotted against the input size, but on a log-log scale.

Fig. 8. Algorithm runtime with respect to input size; lower is better.

Contrary to its poor worst-case asymptotic performance, the recursive descent algorithm is actually best in practice, running most quickly on all tests,

Fig. 9. Maximum algorithm memory use with respect to input size; lower is better.

and using the least memory on all but the largest inputs (where the derivative parsing algorithms’ ability to not buffer input gives them an edge). Packrat parsing is consistently slower than recursive descent, while using two orders of magnitude more memory. The two derivative parsing algorithms have significantly slower runtime, but memory usage closer to recursive descent than packrat.

Though on these well-behaved inputs all four algorithms run in linear time and space (constant space for the derivative parsing algorithms), the constant factor differs by both algorithm and grammar complexity. The XML and JSON grammars are of similar complexity, with 23 and 24 nonterminals, respectively, and all uses of lookahead expressions $!\alpha$ and $\&\alpha$ eliminated by judicious use of the more specialized negative character class, end-of-input, and until expressions described in [12]. It is consequently unsurprising that the parsers have similar runtime performance on those two grammars. By contrast, the Java grammar is significantly more complex, with 178 nonterminals and 54 lookahead expressions, and correspondingly poorer runtime performance.

Both the packrat algorithm and the derivative parsing algorithm presented here trade increased space usage for better runtime. Naturally, this trade-off works more in their favour for more complex grammars, particularly those with more lookahead expressions, as suggested by Moss [12]. Grouping the broadly equivalent XML and JSON tests together and comparing mean speedup, recursive descent is 3.3x as fast as packrat and 18x as fast as SPED on XML and JSON, yet only 1.6x as fast as packrat and 3.7x as fast as SPED for Java. Packrat’s runtime advantage over SPED also decreases from 5.5x to 2.3x between XML/JSON and Java.

Though the packrat algorithm is a modest constant factor faster than the derivative parsing algorithm across the test suite, it uses as much as 300x as

much peak memory on the largest test cases, with the increases scaling linearly in the input size. Derivative parsing, by contrast, maintains a grammar-dependent constant memory usage across all the (well-behaved) inputs tested. This constant memory usage is within a factor of two on either side of the memory usage of the recursive descent implementation on all the XML and JSON inputs tested, and 3–5x more on the more complex Java grammar. The higher memory usage on Java is likely due to the lookahead expressions, which are handled with runtime backtracking in recursive descent, but extra concurrently-processed expressions in derivative parsing.

Derivative parsing in general is known to have poor runtime performance [1, 10], as these results also demonstrate. However, this new algorithm does provide a significant improvement on the current state of the art for parsing expression derivatives, with a 40% speedup on XML and JSON, a 50% speedup on Java, and an up to 13% decrease in memory usage. This improved performance may be beneficial for use cases that specifically require the derivative computation, such as the modular parsers of Brachthäuser *et al.* [2] or the sentence generator of Garnock-Jones *et al.* [6].

7 Conclusion and Future Work

This paper has introduced a new derivative parsing algorithm for PEGs based on the previously-published algorithm in [12]. Its key contributions are simplification of the earlier algorithm and empirical comparison of this new algorithm to previous work. The simplified algorithm also improves the worst-case space and time bounds of the previous algorithm by a linear factor. The author has produced a formal proof of correctness for this simplified algorithm, but was unable to include it in this paper due to space constraints.

While extension of this recognition algorithm to a parsing algorithm remains future work, any such extension may rely on the fact that successfully recognized parsing expressions produce a ε_e expression in this algorithm, where e is the index where the last character was consumed. As one approach, $\langle \bullet \rangle_b$ might annotate parsing expressions with b , the index where they began to consume characters. By collecting subexpression matches and combining the two indices b and e on a successful match, this algorithm should be able to return a parse tree on match, rather than simply a recognition decision. The parser derivative approach of Might *et al.* [10] may be useful here, with the added simplification that PEGs, unlike CFGs, have no more than one valid parse tree, and thus do not need to store multiple possible parses in a single node.

References

1. Adams, M.D., Hollenbeck, C., Might, M.: On the complexity and performance of parsing with derivatives. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, pp. 224–236. ACM, New York (2016)

2. Brachthäuser, J.I., Rendel, T., Ostermann, K.: Parsing with first-class derivatives. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*, pp. 588–606. ACM, New York (2016)
3. Brzozowski, J.A.: Derivatives of regular expressions. *J. ACM (JACM)* **11**(4), 481–494 (1964)
4. Ford, B.: Packrat parsing: a practical linear-time algorithm with backtracking. Master’s thesis, Massachusetts Institute of Technology, September 2002
5. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 111–122. ACM (2004)
6. Garnock-Jones, T., Eslamimehr, M., Warth, A.: Recognising and generating terms using derivatives of parsing expression grammars. arXiv preprint [arXiv:1801.10490](https://arxiv.org/abs/1801.10490) (2018)
7. Henglein, F., Rasmussen, U.T.: 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*, pp. 35–46. ACM, New York (2017)
8. Kuramitsu, K.: Packrat parsing with elastic sliding window. *J. Inf. Process.* **23**(4), 505–512 (2015)
9. Medeiros, S., Ierusalimsky, R.: A parsing machine for PEGs. In: *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008*, pp. 2:1–2:12. ACM, New York (2008)
10. Might, M., Darais, D., Spiewak, D.: Parsing with derivatives: a functional pearl. In: *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 189–195. ACM (2011)
11. Mizushima, K., Maeda, A., Yamaguchi, Y.: 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*, pp. 29–36. ACM (2010)
12. Moss, A.: Derivatives of parsing expression grammars. In: *Proceedings of the 15th International Conference on Automata and Formal Languages, AFL 2017, Debrecen, Hungary, 4–6 September 2017*, pp. 180–194 (2017). <https://doi.org/10.4204/EPTCS.252.18>
13. Moss, A.: Egg (2018). <https://github.com/bruceiv/egg/tree/deriv>
14. Redziejowski, R.R.: Parsing expression grammar as a primitive recursive-descent parser with backtracking. *Fundam. Inform.* **79**(3–4), 513–524 (2007)