Parsing Expression GLL

Moss, Aaron
mossa@up.edu

Harrington, Brynn
harringt23@up.edu

Hoppe, Emily
hoppe23@up.edu

May 3, 2022

Abstract

This paper presents an extension of the GLL parsing algorithm for context-free grammars which also supports parsing expression grammars with ordered choice and lookahead. The new PEGLL algorithm retains support for unordered choice, and thus parses a common superset of context-free grammars and parsing expression grammars. As part of this work, the authors have modified an existing GLL parser-generator to support parsing expression grammars, adding operators for common parsing expressions and modifying the lexer algorithm to better support ordered choice.

1 Introduction

The inherently unambiguous nature of parsing expression grammars (PEGs) makes them an attractive choice for modelling structured text such as programming languages and computing data formats, but in practice the superior performance of parsers based on context-free grammars (CFGs) has led to CFGs being more-widely used, despite the difficulty of disambiguating them. This work is an initial effort toward a unified framework that provides the advantages of both grammar formalisms: it is an algorithm adopted from an efficient, general-purpose CFG parser that supports PEG semantics without discarding support for the unordered choice operator of CFGs in cases where that ambiguity may be desirable.

More specifically, this paper presents a modification of the GLL parser-generator of Scott & Johnstone[1, 2]. The key contribution is the *FailCRF* data structure, which adds a failure path to Scott & Johnstone's call-return forest; the addition of a failure path allows the lookahead and ordered choice operations of the PEG formalism to be supported. The authors have extended Ackerman's GoGLL[3] parser-generator to implement this new algorithm, adding syntactic sugar for common PEG operators and modifying the lexer algorithm to allow the PEG parser to override the usual maximal-munch rule.

$$u(s) = \begin{cases} v & s = uv \\ \mathsf{fail} & \text{otherwise} \end{cases} \qquad A(s) = (\mathcal{R}(A))(s)$$

$$\varepsilon(s) = s \qquad \alpha\beta(s) = \begin{cases} \beta(\alpha(s)) & \alpha(s) \neq \mathsf{fail} \\ \mathsf{fail} & \text{otherwise} \end{cases}$$

$$\varnothing(s) = \mathsf{fail} \qquad \alpha/\beta(s) = \begin{cases} \alpha(s) & \alpha(s) \neq \mathsf{fail} \\ \beta(s) & \text{otherwise} \end{cases}$$

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

$$\&\alpha(s) = \begin{cases} s & \alpha(s) \neq \mathsf{fail} \\ \mathsf{fail} & \text{otherwise} \end{cases}$$

Figure 1: Formal definitions of parsing expressions

2 Parsing Expression Grammars

The primary difference between parsing expression grammars and the more familiar context-free grammars is *ordered choice*: PEGs, as a formalism of recursive-descent parsing introduced by Ford [4], do not try subsequent alternatives of an alternation if an earlier alternative matches. The other significant difference between the PEG and CFG formalisms are the PEG *lookahead* expressions, $!\alpha$ and $\&\alpha$, which match only if the subexpression α does not (resp. does) match, but consume no input regardless. These lookahead operators provide the infinite lookahead of the PEG formalism. The other fundamental PEG operators act much like their CFG equivalents, and are described in Fig. 1 as functions over an input string s drawn from some alphabet Σ producing either a (matching) suffix of s or the special value $\mathsf{fail} \notin \Sigma^*$. In summary, the *string literal* u matches and consumes the string u, the *empty expression* ε always matches without consuming anything, while the *failure expression* $\varnothing$ never matches. A *nonterminal* A is replaced by the parsing expression $\mathcal{R}(A)$ it corresponds to. The *sequence* expression $\alpha\beta$ matches α followed by β, while the *ordered choice* expression α/β only tries β if α does not match. To differentiate CFG *unordered choice*, it is represented in this paper as $\alpha|\beta$.

3 GLL Parsing

Generalized LL (GLL) parsing, introduced by Scott & Johnstone[1, 5], extends the power of LL parsing to all CFGs through use of a *call-return forest* (CRF) to represent the recursive-descent call stack of the LL parsing algorithm. For efficiency, the CRF is implemented using the *graph-structured stack* (GSS) data structure introduced by Tomita[6] for the GLR parsing algorithm. The gist of the GLL approach is that each CRF node represents a function call (equivalently, nonterminal invocation) in a recursive-descent parse, and includes an

input position, a nonterminal to match, and a grammar slot to return to on completion. The graph structure of this stack comes from a dynamic de-duplication of CRF nodes which share a nonterminal and input position, changing a stack data structure into a directed acyclic graph (DAG). The GLL algorithm keeps a queue of CRF nodes which are pending parsing, and handles the nondeterminism of unordered choice by enqueuing a CRF node for each choice.

Scott *et al.*[5] introduced *binary subtree representation* (BSR) sets as an output format to represent nonterminal matches in GLL. The essential insight is that, while the traditional *shared packed parse forest* (SPPF)[6] data structure representing possible parse trees requires significant complication in the parser algorithm to properly store and update edges between parse tree nodes, those edges can be efficiently reconstructed from an indexed set of edgeless parse-tree nodes (the BSR set) with minimal added information.

A BSR element is a 4-tuple containing a *grammar slot* $X ::= \alpha\theta \cdot \beta$, and three input indices i, j, and k, $i \leq j \leq k$. The BSR element represents a successful match of the nonterminal X up to the end of θ, the single terminal or nonterminal immediately before the dot of the grammar slot; i is the input index where X began to match, j is the index where θ began to match, and k is the index where θ finished matching. Note that if $\beta = \varepsilon$, the BSR node represents a complete match of X. Parse trees can be straightforwardly reconstructed from BSR sets: a predecessor of a BSR element $(X ::= \alpha\theta \cdot \beta, i, j, k)$ is any element $(X ::= \alpha \cdot \theta\beta, i, \ell, j)$, while its child where θ is some nonterminal A is any element $(A ::= \delta \cdot, j, m, k)$. Successor and parent elements can be defined analogously.

4 Parsing Expression GLL

The *Parsing Expression GLL* (PEGLL) algorithm introduced in this paper uses similar data structures and abstractions as GLL for CFGs. The main loop is outlined in Figure 2; it first initializes an empty queue R of slot descriptors to parse, an empty cache U of previously seen slot descriptors, and an empty set T of BSR elements to report. It then queues the start rule of the grammar at input position 0 for parsing, parses each descriptor, and completes by returning whether or not the start rule matched at position 0. Note that (unlike CFGs), PEGs match prefixes of their input, so the start rule may only consume the input up to some index k; if this behavior is not desired, a match can be returned only if k is the length of the input string.

The primary thing the loop in Figure 2 does is dispatch the current descriptor to the code which executes its parse. The code for each nonterminal may be generated according to the patterns in Figures 3 and 4. This parser-generator assumes any ordered choice expressions are at the very top level of a nonterminal (parenthesized subexpressions are added as syntactic sugar, see Section 4.2). A label is then generated for each alternate, failing over to the next alternate if it does not match, with a synthesized failure alternate at the end of each alternation.

The $\textsc{rtn}(X, c_U, c_I)$ function in PEGLL code is detailed below in Figure 7, but

Figure 2: PEGLL main loop

```
R ← ∅, U ← ∅, T ← ∅
ADDNT(S, 0)
while R ≠ ∅ do
    (L, c_U, c_I) ← R.REMOVE
    t ← TOKENS(c_I) t' ← t
    loop
        switch L
            ⟨ generate code for each rule R ⟩
        end switch
        nextSlot:
    end loop
    nextDesc:
end while
if ∃α, j, k, (S ::= α·, 0, j, k) ∈ T then
    return match at maximal such k
else
    return fail;
end if
```

Figure 3: Pattern for a nonterminal $X ::= \tau_1 / \ldots / \tau_p, p \geq 1$

```
case X ::= ·τ_1:
    ⟨ generate code for X ::= ·τ_1 with failure path X ::= ·τ_2 ⟩
    RTN(X, c_U, c_I); goto nextDesc

⋮

case X ::= ·τ_p:
    ⟨ generate code for X ::= ·τ_p with failure path X ::= ·∅ ⟩
    RTN(X, c_U, c_I) goto nextDesc

case X ::= ·∅:
    RTN(X, c_U, fail) goto nextDesc
```

Figure 4: Pattern for a nonterminal $X ::= \varepsilon$

```
case X ::= ·ε:
    T ← T ∪ (X ::= ε·, c_I, c_I, c_I)
    RTN(X, c_U, c_I) goto nextDesc
```

Figure 5: Pattern for a sequence $X ::= \cdot x_1 \dots x_d$ with failure path L_f

$r \leftarrow$ testSelect($X ::= \cdot x_1 \dots x_d$, t)
if testSelect failed **then**
    report error
    $(L, c_I, t) \leftarrow (L_f, c_U, t')$ **goto** nextSlot
end if
⟨ generate code for $X ::= x_1 \cdot x_2 \dots x_d$ with failure path L_f ⟩
⋮
⟨ repeat for remaining atoms $x_2 \dots x_d$ in sequence ⟩

Figure 6: Pattern for an atomic expression $X ::= \alpha\varphi \cdot \beta$ with failure path L_f

$\varphi = a \implies$
    $R \leftarrow R \cup (X ::= \alpha a \cdot \beta, c_U, c_I, r)$
    $c_I \leftarrow r$
    $t \leftarrow$ tokens(c_I)

$\varphi = Y \implies$
    call($X ::= \alpha\ Y \cdot \beta$, L_f, Y, c_U, c_I) **goto** nextDesc
case $X ::= \alpha\ Y \cdot \beta$:

$\varphi = \&Y \implies$
    call($X ::= \alpha\ \&Y \cdot \beta$, L_f, Y, c_U, c_I) **goto** nextDesc
case $X ::= \alpha\ \&Y \cdot \beta$:

$\varphi = !Y \implies$
    call($L_f, X ::= \alpha\ !Y \cdot \beta$, Y, c_U, c_I) **goto** nextDesc
case $X ::= \alpha\ !Y \cdot \beta$:

its essential purpose is to modify the "call stack" in the CRF graph consistently with a recursive call to the nonterminal X at position c_U returning at position c_I.

To parse each of the sequence expressions τ_i inside a nonterminal alternative, PEGLL repeatedly executes the appropriate code for each expression in the sequence, moving to the failure path if that expression does not work. The code patterns for the sequence expression are in Figure 5, while the code patterns for each atomic expression are in Figure 6. Terminal matches advance the input index c_I to the right extent r of the token and find the new tokens t at that position, while mid-sequence errors reset c_I and t to their initial values in the descriptor, c_U and t'. Note also that nonterminal calls preserve the failure path from their caller, and in particular that negative-lookahead expressions swap the success and failure results of the call.

Together, the helper functions call and rtn simulate the call stack of a

recursive-descent PEG parser; full code is in Figure 7. The significant difference between the *FailCRF* of PEGLL and the *CRF* of Scott *et al.*[5] is that edges in the *FailCRF* are labelled as either "match" edges or "fail" edges, representing successful and unsuccessful return paths, respectively. The motivation for this modification is handling negative lookahead expressions — $!X$ matches only if X does not, and thus the CRF needs to encode the failure of X as the trigger of a move from slot $Y ::= \alpha \cdot !X\beta$ to slot $Y ::= \alpha !X \cdot \beta$. Fail edges are also used to encode the ordered choice rule; while the GLL algorithm attempts to parse all alternates of a nonterminal concurrently, PEGLL attempts one at a time, following its fail edge to the next alternate if that one fails.

In more detail, the $\textsc{call}(L_m, L_f, X, i, j)$ function enqueues parsing of nonterminal X at position j, returning to slot L_m on match or L_f on failure, where L_m or L_f began parsing at position i. The $\textsc{rtn}(X, j, h)$ function reports the result of parsing nonterminal X begining at position j, where h is either the last consumed position or the special value fail indicating that the parse did not succeed. For convenience, this presentation assumes the call-return forest (CRF) is stored in a global cache. The "popped cache" of previously-parsed results is included to allow updating of previous parse results when they are used in a new context and also assumed to be global.

In addition to call-stack management, a PEGLL parser-generator depends on helper functions to fill the descriptor queue R, descriptor cache U, and the BSR set T representing the parse forest. The descriptor queue management functions are all in Figure 8. addMatch and addFail enqueue the next descriptor after a nonterminal match or failure, accounting for the fact that lookahead expressions consume no input and adding a BSR element for matches. addNt is a convenience function which enqueues the first alternate of a nonterminal in the descriptor queue. addDesc checks if a given descriptor has already been processed, adding it to the descriptor queue and the cache of processed descriptors if not.

4.1 Lexing

Traditional maximal-munch lexers interact poorly with PEG parsers; in particular, the recursive-descent structure of a parsing expression grammar may impose context-sensitive priorities between tokens, and the usual scannerless design of a PEG may result in difficult-to-decompose token sets. Nonetheless, experience [3, 7] has shown that a separate lexing pass is a useful performance optimization. As such, the PEGLL parser-generator uses a regular-expression lexer modified from the traditional maximal-munch approach [8]. Rather than returning a single maximal-munch token at a given input position, as in the classical algorithm , the PEGLL lexer returns (from the $\textsc{tokens}(i)$ function) a map of all the tokens which match starting at position i to their greatest right extent, defering the choice of which token to actually match to the PEG rules in the parser. The testSelect function, described in Figure 9 in the parser is used to check if any tokens in the FIRST set of the expression [9] are also present in the token set returned from the lexer.

Figure 7: Code for CALL and RTN functions

```
function CALL(slot L_m, slot L_f, nonterminal X, int i, int j)
    u_m ← CRF node (L_m, i), created if not in cache
    u_f ← CRF node (L_f, i), created if not in cache
    v ← CRF node (X, j), created if not in cache
    if v was not previously in cache then
        add a match edge from v to u_m and a fail edge from v to u_f
        ADDNT(X, j)
    else
        if there is not a match edge from v to u_m then
            add a match edge from v to u_m
            for all (X, j, h), h ≠ fail in popped cache do
                ADDMATCH(L_m, i, j, h)
            end for
        end if
        if there is not a fail edge from v to u_f then
            add a fail edge from v to u_f
            for all (X, j, fail) in popped cache do
                ADDFAIL(L_f, i, j)
            end for
        end if
    end if
end function

function RTN(nonterminal X, int j, int h)
    if (X, j, h) not in popped cache then
        add (X, j, h) to popped cache
        for all children (L, i) of (X, j) in the CRF do
            if h ≠ fail then
                ADDMATCH(L, i, j, h)
            else
                ADDFAIL(L, i, j)
            end if
        end for
    end if
end function
```

Figure 8: Code for descriptor queue management functions

```
function addMatch(slot L, int i, int j, int h)
    T ← T ∪ (L, i, j, h)
    if there is a lookahead expression before the dot in L then
        addDesc(L, i, j)
    else
        addDesc(L, i, h)
    end if
end function

function addFail(slot L, int i, int j)
    if there is a lookahead expression before the dot in L then
        addDesc(L, i, j)
    else
        addDesc(L, i, i)
    end if
end function

function addNt(nonterminal X, int i)
    L ← initial slot of first alternate of X
    addDesc(L, i, i)
end function

function addDesc(slot L, int i, int h)
    if (L, i, h) ∉ U then
        R ← R ∪ (L, i, h)
        U ← U ∪ (L, i, h)
    end if
end function
```

Figure 9: Code for testSelect function

```
function testSelect(slot L, token set T, int c_I)
    b ← −1
    if L is nullable then b ← c_I
    end if
    for all (t, r) in T do
        if r > b and t ∈ FIRST(L) then b ← r
        end if
    end for
    return b
end function
```

$$\mathcal{R}_{\alpha?} = \alpha/\varepsilon$$
$$\mathcal{R}_{\alpha^*} = \alpha\mathcal{R}_{\alpha^*}/\varepsilon$$
$$\mathcal{R}_{\alpha^+} = \alpha\mathcal{R}_{\alpha^*}$$

Figure 10: Each syntactic sugar expression φ is replaced by a fresh nonterminal $\mathcal{R}_\varphi$ defined as in this table.

4.2 Syntactic Sugar

The preceding discussion has outlined how PEG semantics are supported within the framework of the GLR algorithm; however, support for the fundamental PEG operators does not cover how several common PEG idioms are supported. Much like the EBNF notation for CFGs, PEGs typically support the "syntactic sugar" operators defined in Figure 10 for optional and repeated matches. The $\alpha?$, α^*, and α^+ operators have their usual semantics of match α zero-or-one, zero-or-more, and one-or-more times, respectively, though note that the ordered-choice semantics of PEGs imply greedy match. The PEGLL parser supports expressions using these operators by replacing such expressions during AST generation with a fresh nonterminal defined as in Figure 10.

4.3 Unordered Choice

PEGLL also supports an unordered-choice operator; the main challenge in implementing unordered choice in PEGLL is determining when to trigger the failure path for a nonterminal. PEGLL introduces the failure paths in the FailCRF data structure to support the sequential nature of ordered choice – when one alternate fails, it enqueues the next, and when the final alternate fails the algorithm marks the failure of the entire nonterminal. Unordered choice, by contrast, has concurrent semantics; all alternates can conceptually be executed in parallel (as in Scott & Johnstone's GLL [1, 2]), and the nonterminal only fails if *all* alternates fail.

The approach taken by PEGLL is formalized in Figure 11, but the essential idea is to duplicate all alternates of an unordered choice into a *pass* alternate and a *fail* alternate. Matching of an unordered choice begins with the first fail alternate, and proceeds sequentially through the alternates in source order, as in PEGLL's handling of ordered choice. Fail alternates all enqueue the next fail alternate on failure, or the next pass alternate on success, while pass alternates enqueue the next pass alternate on either success or failure; the final fail alternate marks the failure of the nonterminal on failure, while any alternate that succeeds marks the success of the nonterminal. The effect of this duplication of alternates is to encode the failure or success of previous unordered alternates in the control-flow of the algorithm, rather than modifying the FailCRF to store this information in the data structure. This choice was made to avoid a memory-usage penalty in all FailCRF nodes to support unordered choice, but

Figure 11: Pattern for an unordered-choice nonterminal $X ::= \tau_1 | \ldots | \tau_p, p \geq 1$

case $X_{fail} ::= \cdot\tau_1$:
    $\langle$ generate code for $X ::= \cdot\tau_1$ with failure path $X_{fail} ::= \cdot\tau_2$ $\rangle$
    RTN(X, c_U, c_I);
    $(L, c_I, t) \leftarrow (X_{pass} ::= \cdot\tau_2, c_U, t')$ **goto** nextSlot

case $X_{pass} ::= \cdot\tau_2$:
    $\langle$ generate code for $X ::= \cdot\tau_2$ with failure path $X_{pass} ::= \cdot\tau_3$ $\rangle$
    RTN(X, c_U, c_I);
    $(L, c_I, t) \leftarrow (X_{pass} ::= \cdot\tau_3, c_U, t')$ **goto** nextSlot

$\vdots$

case $X_{fail} ::= \cdot\tau_p$:
    $\langle$ generate code for $X ::= \cdot\tau_p$ with failure path $X_{fail} ::= \cdot\varnothing$ $\rangle$
    RTN(X, c_U, c_I) **goto** nextDesc

case $X_{fail} ::= \cdot\varnothing$:
    RTN(X, c_U, fail) **goto** nextDesc

empirical investigation of the relative trade-offs of both approaches would be a fruitful direction for future work.

5 Related Work

There are a number of pre-existing approaches to PEG parsing; Ford [4] introduced the PEG formalism and two algorithms, a backtracking *recursive descent* algorithm and a memoized *packrat* algorithm. In practice, recursive decent is more efficient, but has exponential worst-case runtime, while packrat trades increased space and time in the common case for linear worst-case runtime[10]; most subsequent work has attempted to mitigate one or both of these shortcomings. Mizushima *et al.* [11] use *cut operators* to reduce the memory usage of packrat parsing, while Kuramitsu [12] and Redziejowski [13] use heuristic table-trimming mechanisms to similar effect. Medeiros & Ierusalimschy [14] and Henglein & Rasmussen [15] have developed a parsing machine approach and a tabular parsing algorithm, respectively, both of which have some evidence of efficient performance. Moss [16] and Garnock-Jones *et al.* [**?**] have independently developed derivative parsing algorithms for PEGs, neither of which claim improved performance, but which allow use of PEGs with algorithmic tools based on derivative parsing.

Scott & Johnstone's GLL algorithm [1, 2] is based on the GLR algorithm of Tomita[6]. The PEGLL implementation (based on Ackerman's GoGLL [3]) also

uses a modified call-return forest (CRF) for control flow and a binary subtree representation (BSR) for parse results, both as in Scott *et al.*[5]. **TODO cite ALL(*) paper, any likely threads from there**

6 Conclusion & Future Work

This paper has described PEGLL, a new algorithm for parsing parsing expression grammars, based on Scott & Johnstone's GLL [1, 2]. The primary contribution is the FailCRF data structure, which adds failure paths to GLL's call-return forest, and code-generation algorithms to implement PEG semantics on top of this data structure. In addition to supporting the fundamental ordered-choice and lookahead operators of PEG, PEGLL also includes support for repetition operators as syntactic sugar, and also for CFG-style unordered choice using a duplication approach to track whether any of the unordered alternates have matched.

An Apache-licenced implementation of PEGLL is available on GitHub[1]; this implementation passes the research team's internal correctness tests, but performance testing is left to future work. In addition to measuring and optimizing the performance of the PEGLL algorithm, a straightforward piece of future work would be to expand the expressivity of PEGLL's operator set; parenthesized subexpressions would be a natural expansion, as would more variants on the repetition operators – in particular, non-greedy variants could be implemented by using unordered choice instead of ordered choice in the generated nonterminals.

A more ambitious advance would be to support left-recursive rules in PEGLL. Combined with PEGLL's existing support for unordered choice, allowing left-recursive rules would make the set of languages parsed by PEGLL a superset of both parsing-expression languages and context-free languages, allowing language designers to apply the idioms and capabilities of both formalisms as desired. Given that GLL already supports left-recursive rules **TODO cite** and a number of approaches have been suggested for PEG-parsing as well **TODO cite**, this goal should be achievable.

References

[1] E. Scott and A. Johnstone, "GLL parsing," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 7, pp. 177–189, 2010.

[2] E. Scott and A. Johnstone, "Structuring the GLL parsing algorithm for performance," *Science of Computer Programming*, vol. 125, pp. 1–22, 2016.

[3] M. Ackerman, "GoGLL." `https://github.com/goccmack/gogll`, 2019. accessed 24-Nov-2021.

[1] `https://github.com/bruceiv/pegll`

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

[5] E. Scott, A. Johnstone, and L. T. van Binsbergen, "Derivation representation using binary subtree sets," *Science of Computer Programming*, vol. 175, pp. 63–84, 2019.

[6] M. Tomita, *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*, vol. 8. Springer Science & Business Media, 1985.

[7] N. Laurent, *Principled Procedural Parsing*. PhD thesis, Université catholique de Louvain, 2019.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques & Tools*. Boston: Pearson Education, second ed., 2007.

[9] R. R. Redziejowski, "Applying classical concepts to parsing expression grammar," *Fundamenta Informaticae*, vol. 93, no. 1-3, pp. 325–336, 2009.

[10] A. Moss, "Derivatives of parsing expression grammars," in *Proceedings 15th International Conference on Automata and Formal Languages, AFL 2017, Debrecen, Hungary, September 4-6, 2017.*, pp. 180–194, 2017.

[11] K. Mizushima, A. Maeda, and Y. Yamaguchi, "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*, PASTE '10, (New York, NY, USA), pp. 29–36, ACM, 2010.

[12] K. Kuramitsu, "Packrat parsing with elastic sliding window," *Journal of Information Processing*, vol. 23, no. 4, pp. 505–512, 2015.

[13] R. R. Redziejowski, "Parsing expression grammar as a primitive recursive-descent parser with backtracking," *Fundamenta Informaticae*, vol. 79, no. 3-4, pp. 513–524, 2007.

[14] S. Medeiros and R. Ierusalimschy, "A parsing machine for PEGs," in *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, (New York, NY, USA), pp. 2:1–2:12, ACM, 2008.

[15] F. Henglein and U. T. Rasmussen, "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, (New York, NY, USA), pp. 35–46, ACM, 2017.

[16] A. Moss, "Simplified parsing expression derivatives," in *International Conference on Language and Automata Theory and Applications*, pp. 425–436, Springer, 2020.