# Parsing Expression GLL

Moss, Aaron
mossa@up.edu

Harrington, Brynn
harringt23@up.edu

Hoppe, Emily
hoppe23@up.edu

November 26, 2021

**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. Performance results of the generated parsers are compared to competing parser-generators.

## 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. This paper also presents benchmarking results from comparing our new *PEGLL* parser-generator against existing algorithms.

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

$$\varepsilon(s) = s$$

$$\varnothing(s) = \mathsf{fail}$$

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

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

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

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

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

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, 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 $\alpha$ 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 $\Sigma$ 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* $\varepsilon$ 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 $\alpha$ followed by $\beta$, while the *ordered choice* expression $\alpha/\beta$ only tries $\beta$ if $\alpha$ 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, 4], 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[5] 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.*[4] 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)[5] 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* $X ::= \alpha \cdot \theta\beta$, 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 $\theta$, the single terminal or nonterminal immediately after the dot of the grammar slot; $i$ is the input index where $X$ began to match, $j$ is the index where $\theta$ began to match, and $k$ is the index where $\theta$ 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 successor of a BSR element $(X ::= \alpha \cdot \theta\beta, i, j, k)$ is any element $(X ::= \alpha\theta \cdot \beta, i, k, \ell)$, while its child where $\theta$ is some nonterminal $A$ is any element $(A ::= \delta \cdot \gamma, j, m, k)$, for $\gamma$ a single terminal or nonterminal.

# 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.

[4] 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.

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