Parsing Expression GLL

Moss, Aaron
mossa@up.edu

Harrington, Brynn
harringt23@up.edu

Hoppe, Emily
hoppe23@up.edu

December 28, 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.

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

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

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

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

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

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, 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, 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* $X ::= \alpha\theta \cdot \beta$, 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 and an empty set T of BSR elements to report, 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 non-terminal 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 non-terminal (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 $\text{rtn}(X, c_U, c_I)$ function in PEGLL code is detailed below in Figure 7, but its essential purpose is to modify the "call stack" in the CRF graph consistently

Figure 2: PEGLL main loop

```
R = ∅;  T = ∅;
ntAdd(S, 0);
while R ≠ ∅ {
    (L, c_U, c_I) = R.remove();
    t = tokens(c_I);  t' = t;
    while true {
        switch L {
            ⟨ generate code for each rule R ⟩
        }
    nextSlot:  }
nextDesc:  }
if ∃α, j, k, (S ::= α·, 0, j, k) ∈ T {
    return match at maximal such k;
} else {
    return fail;
}
```

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

```
X ::= ·τ_1 :
        ⟨ generate code for X ::= ·τ_1 with failure path X ::= ·τ_2 ⟩
        rtn(X, c_U, c_I);  goto nextDesc;
⋮
X ::= ·τ_p :
        ⟨ generate code for X ::= ·τ_p with failure path X ::= ·∅ ⟩
        rtn(X, c_U, c_I);  goto nextDesc;
X ::= ·∅ :
        rtn(X, c_U, fail);  goto nextDesc;
```

Figure 4: Pattern for a non-terminal $X ::= \varepsilon$

```
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 $X ::= \cdot x_1 \ldots x_d$ with failure path L_f

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

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

$\varphi = a \implies$
    $\text{bsrAdd}(X ::= \alpha a \cdot \beta, c_U, c_I, r)$;
    $c_I = r$;
    $t = \text{tokens}(c_I)$;

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

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

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

with a recursive call to the non-terminal 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.

The code patterns for a PEGLL parser-generator depend on a number of

Figure 7: Code for call and rtn functions

```
void 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 {
        %       add a match edge from v to u_m and a fail edge from v to
u_f;
        %       ntAdd(X, j);
        % } else {
        %       % if there is not a match edge from v to u_m {
        %       %       add a match edge from v to u_m;
```

helper functions to manipulate the call graph and BSR set representing the parse tree. The 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 rtn(X, k, j) function reports the result of parsing nonterminal X begining at position k, where j is either the last consumed position or the special value fail indicating that the parse did not succeed. Together, call() and return() 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.*[4] is that edges in the *FailCRF* are labelled as either "match" edges or "fail" edges, representing successful and unsuccessful return paths, respectively. 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.

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, 6] 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 [7]. Rather than returning a single maximal-munch token at a given input position, as in the classical algorithm , the PEGLL lexer returns (from the 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 8 in the parser is used to check if any tokens in the FIRST set of the expression [8] are also present in

Figure 8: Code for testSelect function

```
int testSelect(slot L, token set T, index c_I) {
        b = −1;
        if L is nullable { b = c_I; }
        for (t, r) in T {
                if r > b and t ∈ FIRST(L) { b = r; }
        }
        return b;
}
```

the token set returned from the lexer.

4.2 Syntactic Sugar

TODO write me.

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.

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

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

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