Computational Model for Parsing Expression Grammars

Alexander Rubtsov
HSE University, Moscow, Russia
MIPT, Moscow, Russia

Nikita Chudinov
Google, Zürich, Switzerland

Abstract

We present a computational model for Parsing Expression Grammars (PEGs). The predecessor of PEGs top-down parsing languages (TDPLs) were discovered by A. Birman and J. Ullman in the 1960-s, B. Ford showed in 2004 that both formalisms recognize the same class named Parsing Expression Languages (PELs). A. Birman and J. Ullman established such important properties like TDPLs generate any DCFL and some non-context-free languages like $a^nb^nc^n$, a linear-time parsing algorithm was constructed as well. But since this parsing algorithm was impractical in the 60-s TDPLs were abandoned and then upgraded by B. Ford to PEGs, so the parsing algorithm was improved (from the practical point of view) as well. Now PEGs are actively used in compilers (eg., Python replaced LL(1)-parser with a PEG one) so as for text processing as well. In this paper, we present a computational model for PEG, obtain structural properties of PELs, namely proof that PELs contain Boolean closure of regular closure of DCFLs and PELs are closed over left concatenation with regular closure of DCFLs. We present an extension of the PELs class based on the extension of our computational model. Our model is an upgrade of deterministic pushdown automata (DPDA) such that during the pop of a symbol it is allowed to return the head to the position of the push of the symbol. We provide a linear-time simulation algorithm for the 2-way version of this model, which is similar to the famous S. Cook linear-time simulation algorithm of 2-way DPDA.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases PEG, formal languages, pushdown automata, two-way pushdown automata

Digital Object Identifier 10.4230/LIPIcs.MFCS.2024.80

Related Version *Full Version*: `https://arxiv.org/abs/2406.14911` [13]

Funding *Alexander Rubtsov*: This paper was prepared within the framework of the HSE University Basic Research Program. Supported in part by RFBR grant 20–01–00645.

Acknowledgements Authors thank Mikhail Iumanov for helpful discussions and anonymous referees for the helpful comments.

1 Introduction

We present a computational model for Parsing Expression Grammars (PEGs) presented by B. Ford in [6]. The predecessor of PEGs top-down parsing languages (TDPLs) was discovered by A. Birman and J. Ullman in the 1960s (so as generalized TDPLs) [4]. While the PEGs formalism has more operations, it has the same power as TDPLs and generalized TDPLs which was shown by B. Ford in [7]. We refer to the class of languages generated by PEGs (and TDPLs, and generalized TDPLs) as *Parsing Expression Languages* (PELs).

Little is known about the structural properties of PELs. From the 60's it is known that PELs contain DCFLs as a subclass and some non-context-free languages like $a^nb^nc^n$ as well. A linear-time parsing algorithm (in RAM) had been constructed for TDPLs, but it was impractical in the 1960s since it required too much memory for memoization and TDPLs had been abandoned. B. Ford upgraded the TDPLs formalism to PEGs and presented a linear-time practical algorithm in 2002 [6]. Now PEGs are being actively used in compilers (eg., Python replaced an LL(1)-parser with a PEG one [16]) so as for text processing as well. In this paper, we present a computational model for PELs and obtain some interesting properties for this class, analyze (some of) its subclasses, and generalize the PELs class as well.

A computational model for PELs was presented in [12], but this model significantly differs from classical models of computations, so it is hard to clarify the place of PELs among known classes of formal languages, based on this model. So we present a simpler and more convenient model that discovers the place of PEGs in the variety of formal language classes. Namely, the computational model is a modified deterministic pushdown automaton (DPDA) that puts to the stack a symbol with the pointer of the head's position on the tape (from which the push has been performed). During the pop, the automaton has two options: either leave the head in the current position or move the head to the position stored in the pointer (retrieved during the pop of the symbol). We call this model a deterministic pointer pushdown automaton (DPPDA). This description of PELs from the automata point of view helped us to obtain other important results not only for the PELs but for the general area of formal languages as well. Namely, we prove that boolean closure of regular closure of DCFLs is linear-time recognizable (in RAM), what extends the nontrivial result by E. Bertsch and M.-J. Nederhof [3] that regular closure of DCFLs is linear-time recognizable.

To describe our results we shall mention the following important results in the area of formal languages and automata theory. Donald Knuth invented LR(k) grammars that describe DCFLs for $k \geq 1$ and were widely used in practice. It is easier to design an LL(k) grammar for practical purposes, so despite the power of LR, LL grammars are widely used for parsing (and some artificial modification of recursive descent parsing as well). Top-down parsing languages (TDPLs, predecessor of PEGs) cover LL(1) grammars and even contain DCFLs as a subclass, but their linear-time parsing algorithm was impractical in the 1970s, so TDPLs had been abandoned till B. Ford upgraded them to PEGs and presented a practically reasonable linear-time parser (Packrat). So, linear-time recognizable classes of formal languages are used in compilers, and LR (DCFLs) parsers now compete with PEGs which cover a wider class of formal languages that is almost undiscovered. There are no comprehensive results on the structure of PELs, so we make a contribution to this open question. Another wide linear-time recognizable class of formal languages is languages recognizable by two-way deterministic pushdown automata (2DPDA). S. Cook obtained in [5] a famous linear-time simulation algorithm for this model. There also was an amazing story about how D. Knuth used S. Cook's algorithm to discover the Knuth-Morris-Pratt algorithm ([10], Section 7).

We modify 2DPDA in the same way as we did for DPDA: we add symbols to stack with a pointer that allows returning the head to the cell from which the push had been performed. S. Cook's linear-time simulation algorithm applies to this model as well (with a little modification). So we extend the important class of formal languages (recognizable by 2DPDAs) preserving linear-time parsing. This extension can be used to generalize PEGs. Also, this algorithm provides another approach to linear time recognition of languages generated by PEGs described via DPPDAs. Note that there are not many structural results about PELs. Moreover, even equivalence of TDPLs and generalized TDPLs (with PEGs) had been proved by B. Ford [7] decades after these classes had been invented. In our opinion, one of the reasons for that is that TDPL-based formalisms are hard. So even the proof of inclusion DCFLs in PELs [4] is complicated, while it directly follows from the equivalence of PEGs with our model.

So we hope that our model will raise interest in investigations of PELs and will help with these investigations as well. Our results also clarify the place of another interesting result (we also improved it, as described below). It was shown by E. Bertsch and M.-J. Nederhof [3] that regular closure of DCFLs is linear-time recognizable. We show that this class is recognizable by DPPDAs which simplifies the original proof [3] and shows the place of this class in the formal languages classes.

There are many linear-time recognizable classes of formal languages. Recently Rubtsov showed [14] that Hibbard's hierarchy (the subclass of CFLs) is linear-time recognizable. So there are many open questions related to the systematization of linear time recognizable classes of formal languages and particularly the relation of Hibbard's hierarchy with languages recognizable by 1-2 DPPDAs.

1.1 Results

In this paper, we present a new computational model DPPDA which is equivalent to PEGs. We also consider the two-way model 2DPPDA and provide a linear time simulation algorithm for this model following S. Cook's construction. Via DPPDA we show that the PEGs class is closed over left concatenation with regular closure of DCFLs, so PELs contain the regular closure of DCFLs as a subclass. With the linear-time simulation algorithm for 2DPPDA, we obtain another linear-time recognition algorithm for the regular closure of DCFLs and since PELs are closed over Boolean operation we prove that the Boolean closure of regular closure of DCFLs is linear-time recognizable. Note that the last result not only generalizes well known result of linear-time recognizability of regular closure of DCFLs [3], but also our proof is significantly simpler as well.

The full version of this paper is available on arXiv [13]. We put the reference to [13] when the proof is omitted due to the space limitations.

1.2 Basic Notation

We follow the notation from [9] on formal languages, especially on context-free grammars (CFGs) and pushdown automata. We denote the input alphabet as Σ and its elements (letters, terminals) are denoted by small letters $a, b, c, \ldots$, while letters w, x, y, z denote words. The empty word is denoted by ε. We denote nonterminals N by capital letters $A, B, C, \ldots$, and X, Y, Z can be used for both nonterminals and terminals. The axiom is denoted by $S \in N$. Words over the alphabet $N \cup \Sigma$ are called sentential forms and are denoted by small Greek letters.

1.3 Informal Description of PEGs

The formal definition of PEGs is not well intuitive, so we begin with an informal one that clarifies a simple idea behind this formal model. The intuition behind PEGs lies in recursive descent parsing.

One of the parsing methods for CF-grammars is a recursive descent parsing that is a process when the derivation tree is built top-down (starting from the axiom S) and then each nonterminal is substituted according to the associated function. A rollback is possible as well, where by rollback we mean the replacement of one production rule by another or even the replacement of the rule higher above the current node with the deletion of subtrees. This method is very general and we do not go deep into details. For our needs, we describe a recursive descent parsing of LL(1) grammars and its modification that defines PEGs.

For LL(1) grammar, the following assertion holds. Fix a leftmost derivation of a word $w\triangleleft = uav\triangleleft$ and let $uA\alpha\triangleleft$ be a derivation step (here $\triangleleft$ is a right end marker of the input). The next leftmost derivation step is determined by the nonterminal A and the terminal a, so the rule is the function $R(A, a)$. So, the recursive descent algorithm for an LL(1)-parser is as follows. An input $w\triangleleft$ is written in the one-way read-only tape called the *input tape*. The pointer in the (constructing) derivation tree points to the leftmost nonterminal node (without children), initially the axiom S. This node is replaced according to the function R. In the fixed above derivation step $uA\alpha\triangleleft$ the pointer is over the nonterminal A, $R(A, a) = xB\beta$, where $A \to xB\beta$ is a grammar rule. So, $xB\beta$ is glued into A as a subtree, x is a prefix of av and the head of the input tape moves while scanning x. If $R(A, a)$ does not contain a nonterminal, then (after replacement) the tree is traversed via DFS until the next (leftmost!) nonterminal is met. Each terminal during this traversal shifts the head of the input tape. If the symbol under the head differs from the traversed terminal, the input word is rejected. We illustrated the described process in Fig. 1. Note that u, x, α, β are the subtrees and u, v, x, v' in fact occupies several cells of the input tape.

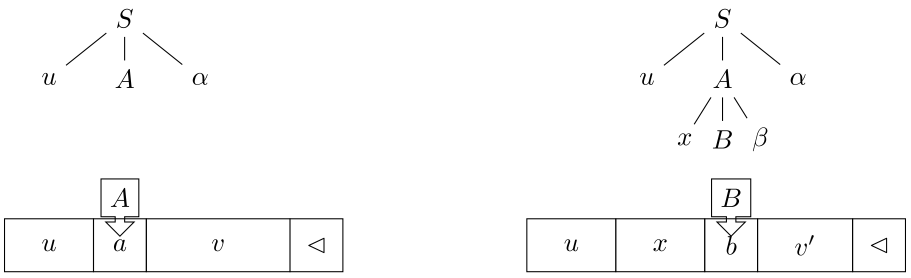

Figure 1 Example of LL(1) recursive descent parsing.

So now we move to the description of PEGs via modification of recursive descent parsing. In the first example, we will provide similar PEG and CFG (Fig. 2) and explain their similarity and differences.

PEG	**CFG**
$S \leftarrow AB \,/\, BC$	$S \to AB \mid BC$
$A \leftarrow aA \,/\, a$	$A \to aA \mid a$
$B \leftarrow abb \,/\, b$	$B \to abb \mid b$
$C \leftarrow cC \,/\, \varepsilon$	$C \to cC \mid \varepsilon$

Figure 2 PEG and CFG for comparison.

PEGs look similar to context-free grammars, but the meaning of almost all concepts are different, therefore the arrow $\leftarrow$ is used to separate the left part of a rule from the right part. The difference comes from the following approach to recursive descent parsing. We describe the PEG via the transformation of the CFG. Let us order all the rules of the CFG for each nonterminal. During recursive descent parsing, we will try each rule according to this order. If a failure happens, let us try the next rule in the order. If the last rule leads us to the failure too, propagate the failure to the parent and try using the next rule in the order on the previous tree level. So, that is the reason why all right-hand sides of the rules in PEG are separated by the delimiter /, but not by |. The order of rules in PEGs matters, unlike CFGs. Consider the parsing (Fig. 3) of the word aab by the PEG defined on Fig. 2.

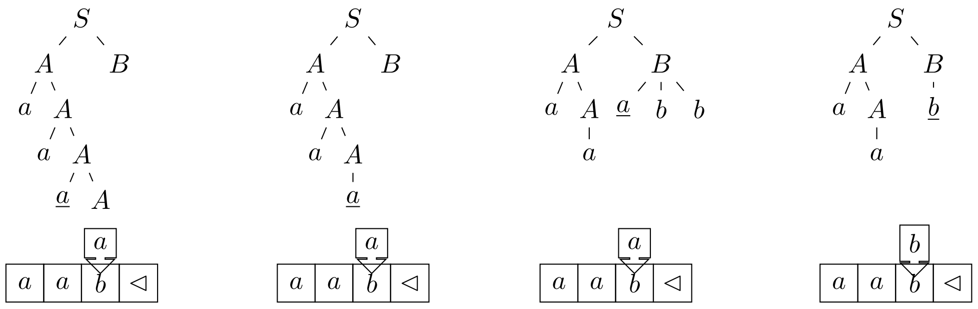

Figure 3 Parsing of aab by PEG.

The rule $A \leftarrow aA$ is applied while the content of the input tape matches the crown (the leafs) of the tree. So, when the last application is unsuccessful, it is replaced by the following rule $A \leftarrow a$ which is unsuccessful too. So failure signal goes to the level above and the second rule $A \leftarrow aA$ is replaced by $A \leftarrow a$. After that, the control goes to the nonterminal B for which firstly the rule $B \leftarrow abb$ is applied, but since it leads to the failure, finally the rule $B \leftarrow b$ is applied and it finishes the parsing since the whole word has been matched.

So PEGs are similar to CFGs since they share the idea of recursive descent parsing. But the difference is significant. Since all the rules for each nonterminal are ordered, the classical notion of concatenation does not apply to PEGs. We cannot say that if a word u is derived from A and v is derived from B, then uv is derived from AB as explained below. In the PEG example above, a word abb is never derived from B because A from AB will always parse all a's from the input. Note that the failure during the parsing occurs only because of a mismatch. So, the input $abbc$ will be parsed by the PEG as following. The prefix ab will be successfully parsed by AB and by S as well, but since the whole word has not been parsed, the input is rejected. Since there was no failure, the rule $S \leftarrow AB$ was not replaced by $S \leftarrow BC$. So the word $abbc$ is not accepted by the PEG while it is derived from the CFG.

▶ Remark 1. It is an open question, whether PELs are closed over concatenation.

Note that the patterns of iteration $A \leftarrow aA\,/\,a$ and $C \leftarrow cC\,/\,\varepsilon$ work in a greedy way. In the case of concatenation Ce (with an expression e), all c's from the prefix of the input would be parsed by C.

In the considered example we have not mentioned an important PEG's operation. There is a unary operator **!** that is applied as follows. In the case **!**e the following happens. Firstly the parsing goes to the expression e. If e parsed the following input successfully (i.e., a subtree for e that matches the prefix of the unprocessed part of the input has been constructed without a failure), then **!**e produces failure. If a failure happens, then **!**e is considered to parse the empty word ε and the parsing process continues. For example, consider the following PEG:

$$S \leftarrow A(\mathbf{!}C)\,/\,B \quad A \leftarrow aAb\,/\,\varepsilon \quad B \leftarrow aBc\,/\,\varepsilon \quad C \leftarrow a\,/\,b$$

($\mathbf{!}C$) guarantees that if $A(\mathbf{!}C)$ finished without failure, then it parsed the whole input. So, in the case of the input a^nb^n for $n \geq 0$, the input will be parsed by $A(\mathbf{!}C)$ and there will be no switch to the rule $S \leftarrow B$. For any other input, the parsing of $A(\mathbf{!}C)$ fails and the rule is switched to $S \leftarrow B$. So, this PEG generates the language $\{a^nb^n \mid n \geq 0\} \cup \{a^nc^n \mid n \geq 0\}$.

Another common use of the operator **!** is its double application that has its name: $\&e = \mathbf{!}(\mathbf{!}e)$. This construction checks whether the prefix of the (unprocessed part of the) input matches e: in the positive case, & parses an empty word and computation continues, in the negative case, it returns failure. So & acts similarly to **!**, but the conditions are flipped. Consider the following example:

$$S \leftarrow (\&(Ac))BC \quad A \leftarrow aAb \,/\, \varepsilon \quad B \leftarrow aB \,/\, a \quad C \leftarrow bCc \,/\, \varepsilon$$

This PEG checks that the input has the prefix $a^n b^n c$ and then parses the input if it has the form $a^+ b^n c^n$, so the PEG generates the language $\{a^n b^n c^n \mid n \geq 1\}$.

So it is known that PEGs generate non CFLs and it is still an open question whether PEGs generate all CFLs. The conditional answer is no: there exists a linear-time parsing algorithm for PEG, while the work of L. Lee [11] and Abboud et al. [1] proves that it is very unlikely for CFLs due to theoretical-complexity assumptions: any CFG parser with time complexity $O(gn^{3-\varepsilon})$, where g is the size of the grammar and n is the length of the input word, can be efficiently converted into an algorithm to multiply $m \times m$ Boolean matrices in time $O(m^{3-\varepsilon/3})$. Note that this conditional result shows that it is unlikely that 2DPPDAs recognize all CFLs as well.

2 Formal Definition of PEGs

Our definition slightly differs from the standard definition of PEG from [7] (Section 3) due to technical reasons. We discuss the difference after the formal definition.

▶ **Definition 2.** *A parsing expression grammar G is defined by a tuple (N, Σ, P, S), where N is a finite set of symbols called* nonterminals*, Σ is a finite input alphabet (a set of* terminals*), $N \cap \Sigma = \varnothing$, $S \in N$ is* the axiom*, and P is a set of* production rules *of the form $A \leftarrow e$ such that each nonterminal $A \in N$ has the only corresponding rule, and e is an expression that is defined recursively as follows. The empty word ε, a terminal $a \in \Sigma$, and a nonterminal $A \in N$ are expressions. If e and e' are expressions, then so are (e) which is equivalent to e, a* sequence ee'*, a* prioritized choice $e \,/\, e'$*, a* not predicate $\mathbf{!}e$*. We assume that* **!** *has the highest priority, the next priority has the sequence operation and the prioritized choice has the lowest one. We denote the set of all expressions over G by E_G or by E if the grammar is fixed.*

To define the language generated by a PEG G we define recursively a partial function $R : E \times \Sigma^* \to (\Sigma^* \cup \{\mathbf{F}\})$ that takes as input the expression e, the input word w, and if $R(e, w) = s \in \Sigma^*$, then s is the suffix of $w = ps$ such that the prefix p has been parsed by e during the processing of w; if $R(e, w) = \mathbf{F}$ it indicates a failure that happens during the parsing process. So, the function R is defined recursively as follows:

- $R(\varepsilon, w) = w$, $R(a, as) = s$, $R(a, bs) = \mathbf{F}$ (where $a \neq b$)
- $R(e_1e_2, w) = R(e_2, R(e_1, w))$ if $R(e_1, w) \neq \mathbf{F}$, otherwise $R(e_1e_2, w) = \mathbf{F}$
- $R(A, w) = R(e, w)$, where $A \leftarrow e \in P$
- $R(e_1 \,/\, e_2, w) = R(e_1, w)$ if $R(e_1, w) \neq \mathbf{F}$, otherwise $R(e_1 \,/\, e_2, w) = R(e_2, w)$
- $R(\mathbf{!}e, w) = w$ if $R(e, w) = \mathbf{F}$, otherwise $R(\mathbf{!}e, w) = \mathbf{F}$

Note that $R(e, w)$ is undefined if during the recursive computation, R comes to an infinite loop. In fact, we will never meet this case because for each PEG there exists an equivalent form for which R is a total function (see Subsection 2.1).

We say that a PEG G *generates* the language $L(G) = \{w \mid R(S, w) = \varepsilon\}$; if $R(S, w) = \varepsilon$ we say that w is *generated* by G.

2.1 Difference with other standard definitions and forms of PEGs

Note that our definition of $L(G)$ differs from [7] (Section 3). The difference is about the operations allowed in PEG and the acceptance condition as well. In this subsection, we explain the difference and provide an overview of different forms of PEGs.

In the case of practical parsing, it is convenient to have more operations in the definition of PEG, but theoretically, it is more convenient to have fewer operations for the sake of the proofs' simplicity. In [7] B. Ford investigated different forms of PEGs and proved their equivalence, so as the equivalence with (generalized) top-down parsing languages. We begin our overview with operations that are so easy to express via operations from our definitions that they can be considered (as programmers say) syntactic sugar:

- **Iterations:** e^* is equivalent to $A \leftarrow eA \,/\, \varepsilon$; $e^+ = ee^*$
- **Option expression:** $e?$ is equivalent to $A \leftarrow e \,/\, \varepsilon$
- **And predicate:** $\&e = \mathbf{!}(\mathbf{!}e)$
- **Any character:** $\bullet = a_1 \,/\, a_2 \,/\, \dots \,/\, a_k$ where $\Sigma = \{a_1, \dots, a_k\}$
- **Failure:** $\mathbf{F} = \mathbf{!}\varepsilon$ (we use the same notation as for the failure result)

We can use these constructions below. In this case, the reader can assume that they are reduced to the operations from Definition 2 as we have described.

So by adding to the definition (or removing) syntactic sugar operations, one obviously obtains an equivalent definition (in terms of recognizable languages' class). Now we move to the nontrivial cases proved in [7].

A PEG G is *complete* if for each $w \in \Sigma^*$ the function $R(S, w)$ is defined. A PEG G is *well-formed* if it does not contain directly or mutually left-recursive rules, such as $A \leftarrow Aa \,/\, a$. It is easy to see that a well-formed grammar is complete. It was proved in [7] that each PEG has an equivalent well-formed one and the algorithm of the transformation had been provided as well. So from now on we assume that each PEG in our constructions is well-formed. Note that most PEGs that are used in practice are well formed by construction.

Another interesting result from [7] is that each PEG has an equivalent one without predicate $\mathbf{!}$. Despite this fact, we decided to include $\mathbf{!}$ in our definition since unlike substitutions for syntactical sugar operations, removing $\mathbf{!}$ predicate requires significant transformations of the PEG. Since $\mathbf{!}$ predicate is widely used in practice and it does not affect our constructions, by including $\mathbf{!}$ in the definition we achieve the constructions that can be used in practice.

As we have already mentioned our condition of the input acceptance also differs from [7]. We used the provided approach since if $R(S, w) = \varepsilon$ we can reconstruct the parsing tree with the root S that generates w. We use this property for the transformation of PEG to the computational model and the inverse transformation as well. Firstly, in [7] there is no axiom in PEG, but there is a starting expression e_S. This difference is insignificant since one can state $e_S = S$ and $S \leftarrow e_S$ for the opposite direction. A PEG from [7] generates the input w if $R(e_S, w) \neq \mathbf{F}$, so $R(e_S, w) = y$, where $w = xy$. So to translate PEG from [7] to ours one needs to set $S \leftarrow e_S(\bullet)^*$. The transformation in the other direction is $e_S = S(\mathbf{!}\bullet)$.

3 Definition of the Computational Model

We call our model *deterministic pointer pushdown automata* (DPPDA). We consider a one-way model (1DPPDA or just DPPDA) as a restricted case of a two-way model (2DPPDA), so we define the two-way model only.

▶ **Definition 3.** *A 2-way deterministic pointer pushdown automata M is defined by a tuple*

$$\langle Q, \Sigma_{\rhd\lhd}, \Gamma, F, q_0, z_0, \delta \rangle$$

- *Q is the finite set of automaton states.*
- *$\Sigma_{\rhd\lhd} = \Sigma \cup \{\rhd, \lhd\}$, where Σ is the finite input alphabet and $\rhd, \lhd$ are the endmarkers. The input has the form $\rhd w \lhd$, $w \in \Sigma^*$.*
- *Γ is the alphabet of the pushdown storage.*
- *$F \subseteq Q$ is the set of the final states.*
- *$q_0 \in Q$ is the initial state.*
- *$Z_0 \in \Gamma$ is the initial symbol in the pushdown storage.*
- *δ is the partial transition function defined as $\delta : Q \times \Sigma_{\rhd\lhd} \times \Gamma \to Q \times \Gamma_\varepsilon^* \times \{\leftarrow, \downarrow, \uparrow, \rightarrow\}$, where $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. Moreover, if $\delta(q, a, z) = (q', \alpha, \uparrow)$, then $\alpha = \varepsilon$.*

To define and operate with configurations of automata we introduce some notation. We denoted by $\alpha \times \vec{i} = (Z_m, i_m), \ldots, (Z_0, i_0)$ the zip of the sequences α and $\vec{i}$, which are of the same length by the definition. A right associative operation $x : \vec{l}$ prepends an element x to the beginning of the vector $\vec{l}$ (we adopt this operator from Haskell programming language). E.g., if $\vec{l} = 1, 2, 3$ and $\vec{r} = 2, 3$, we write $\vec{l} = 1 : \vec{r}$.

A configuration of M on a word w is a quadruple $c \in Q \times (\Gamma \times I)^* \times I$, where $I = \{0, \ldots, |w| + 1\}$; we refer to $w_i, i \in I$ as the i-th input symbol; $w_0 = \rhd$, $w_{|w|+1} = \lhd$. A configuration $c = (q, \alpha \times \vec{i}, j)$ has the following meaning. The head of 2DPPDA M is over the symbol w_j in the state q; the pushdown contains $\alpha = Z_m Z_{m-1} \cdots Z_0$ (the stack grows from right to left) and there is also additional information vector $\vec{i} = i_m, i_{m-1}, \ldots, i_0$, $i_k \in I$ such that Z_k was pushed to the pushdown store when the head was over the i_k-th cell.

The automaton's move is defined via the relation $\vdash$ as follows. Let $\delta(q, a, Z_n) = (q', \beta, d)$. The relation

$$(q, Z_n\alpha \times i_n : \vec{i}, j) \vdash (q', \alpha' \times \vec{i}', j')$$

is defined according to the following case analysis.

- If $d \in \{\leftarrow, \downarrow, \rightarrow\}$, then $j' = j - 1$, $j' = j$, $j' = j + 1$ respectively. The cases $a = \rhd$, $d = \leftarrow$ and $a = \lhd$, $d = \rightarrow$ are forbidden.
- If $\beta = \varepsilon$ and $d \in \{\leftarrow, \downarrow, \rightarrow\}$, then $\alpha' = \alpha$, $\vec{i}' = \vec{i}$
- If $\beta = \varepsilon$ and $d = \uparrow$, then $\alpha' = \alpha$, $\vec{i}' = \vec{i}$, $j' = i_n$
- If $\beta = X_1 \cdots X_k$, $k > 0$, then $\alpha' = \beta Z_n \alpha$, $\vec{i}' = \underbrace{j' : j' : \cdots : j'}_{k} : i_n : \vec{i}$

The initial configuration is $(q_0, Z_0 \times 0, 0)$ and an accepting configuration is $(q_f, \varepsilon \times (), |w| + 1)$, where $q_f \in F$ and by $()$ we have denoted the empty sequence of integers. I.e., M reaches the right end marker $\lhd$ empties the stack and finishes the computation in an accepting state. Formally, a word w is accepted by M if there exists a computational path from the initial configuration to an accepting one.

In the case of 1DPPDA (or just DPPDA), the moves $\leftarrow$ are forbidden.

3.1 Properties of DPPDA

Now we discuss the properties of the model and provide some shortcuts for the following needs. Note that each move of a 2DPPDA is either push- or pop-move due to the sake of convenience in the proofs (induction invariants are simpler). At the same time, in constructions, it is convenient to have right, left, and even stay moves that do not change the stack. So we add moves $\hookrightarrow$, $\hookleftarrow$, and $\looparrowdownright$ that are syntactic sugar for such moves. So, when we write $\delta(q, a, z) = (p, \hookrightarrow)$, we mean the sequence of moves:

$$\delta(q, a, z) = (p', Z', \rightarrow); \forall \sigma \in \Sigma_{\rhd\lhd} : \delta(p', \sigma, Z') = (p, \varepsilon, \downarrow).$$

The construction for $\hookleftarrow$ and $\looparrowdownright$ are similar.

Due to the definition of δ, a DPPDA can move only if the stack is non-empty and since each move is either push or pop, we have that Z_0 lies at the bottom of the stack till the last move of a computation or even after the last move in the case of unsuccessful computation. In the case of a successful computation, Z_0 is popped at the last move.

4 Equivalence of DPPDAs and PEGs

In this section, we provide an algorithm that transforms a PEG into a DPPDA. The algorithm of the inverse transformation provided in [13] due to space limitations. Our construction is similar to the well-known proof of equivalence between CFGs and DPDA for CFLs, but since both DPPDAs and PEGs are more complicated than DPDAs and CFLs, our constructions are technically harder. We refer the reader to [15] for the detailed explanation of the proof idea, where it was provided for CFLs (so as the proof for CFLs as well).

In this section we assume that PEGs have a special form. We call it Chomsky's normal form since it is similar to such a form for CFGs.

▶ **Definition 4.** *A PEG G has a* Chomsky normal form *if the axiom S never occurs on the right side of the rules and the rules are of the following form:*

$$A \leftarrow B\,/\,C, \quad A \leftarrow BC, \quad A \leftarrow\, !B, \quad A \leftarrow a, \quad A \leftarrow \varepsilon.$$

▶ **Lemma 5** ([13]). *Each PEG G has an equivalent PEG G' in Chomsky's normal form which is complete if so was G.*

▶ **Theorem 6.** *For a PEG G there exists an equivalent DPPDA M.*

Proof. We assume that G is a well-formed PEG in a Chomsky normal form (by Lemma 5). We construct an equivalent DPPDA $M = \langle Q, \Sigma_{\rhd\lhd}, \Gamma, \{q_f\}, q_0, Z_0, \delta\rangle$ by the PEGs description. We formally describe δ on Fig. 4; we do not provide a full list of states Q and pushdown alphabet Γ since most of the states and symbols depend on rules listed in δ's construction and can be easily restored from it. Since the construction is straightforward, we describe here only the main details.

The DPPDA M simulates the parsing process of a PEG G on the input w. Firstly M performs a series of technical moves to come from the initial configuration to the initial simulation configuration:

$$(q_0, Z_0 \times 0, 0) \overset{*}{\vdash} (q, SZ_0 \times (1:0), 1),$$

where q is the *main work state* and S is the axiom of the PEG.

During the simulation the following invariants hold. Below A is a nonterminal of the PEG.

1. If the automaton is in the main work state q and on the top of the stack is the pair $A \times i$, then the head is over the cell i.
2. If the head is over the cell $r+1$ in a state $q_{A_\pm}$ (hereinafter $q_{A_\pm} \in \{q_{A_+}, q_{A_-}\}$) and the topmost symbol had been added at the position l, then it means the following.

   q_{A_+} A subword $s = w_l \cdots w_r$ would be parsed by the PEG from A (starting from the position l); when $r+1 = l$, we have $s = \varepsilon$. In the other direction: if the PEG parses $w_l \cdots w_r$ from A starting from the position l, then the DPPDA that starts computation from the position l in the main work state q with A on the top on the stack finishes at the position $r+1$ with (the same) A on the top of the stack, i.e.,

$$(q, A\alpha \times l : \vec{i}, l) \overset{*}{\vdash} (q_{A_+}, A\alpha \times l : \vec{i}, r+1).$$

q_{A_-} After the PEG started parsing from A from the position l, the computation ended up with a failure at some point in the case of q_{A_-}. In the other direction: if the PEG fails, then for some $r \geq l-1$:

$$(q, A\alpha \times l : \vec{i}, l) \overset{*}{\vdash} (q_{A_-}, A\alpha \times l : \vec{i}, r+1).$$

DPPDA M accepts the input only if the head reaches the symbol $\lhd$ in the state q_{S+} (note that the axiom does not occur on the right side of the rules). Formally, we add the rule

$$\delta(q_{S_+}, \lhd, Z_0) = (q_f, \varepsilon, \downarrow),$$

where q_f is the only final state of the DPPDA. So, from the invariant it follows that the DPPDA accepts the input iff the PEG parses the input.

The rest of the construction is the delta's description in Fig. 4. The proof is a straightforward induction on the recursion depth of the PEGs computation. So we describe the behavior of the automaton corresponding to formal construction in two main cases and check that the invariants hold (the remaining cases are simple).

Construction of δ

We denote by $Z \in \Gamma$ and $\sigma \in \Sigma_{\rhd\lhd}$ arbitrary symbols. The rules are grouped with respect to the PEG's operations. Note that the states q, $q_{A_\pm}$ are the same for all rules, while other states and stack symbols depend on the rule, i.e., stack symbols A_1's from different rules are different even if they correspond to the same nonterminal A. When we use nonterminals (and states) with signs $\pm$ or $\mp$, the signs have corresponding matching, i.e., if in a rule we have $A_\pm$ and $B_\mp$, then when $A_\pm = A_+$, $B_\mp$ equals to B_- and when $A_\pm = A_-$, $B_\mp$ equals to B_+.

0. General rules
- $\delta(q_0, \rhd, Z_0) = (q_0, \hookrightarrow)$; $\forall \sigma' \in \Sigma \cup \{\lhd\} : \delta(q_0, \sigma', Z_0) = (q, S, \downarrow)$;
- $\delta(q_{A_\pm}, \sigma, A) = (q_{A_\pm}, \varepsilon, \downarrow)$; $\delta(q_{S_+}, \lhd, Z_0) = (q_f, \varepsilon, \downarrow)$.

1. $A \leftarrow BC$
- $\delta(q, \sigma, A) = (q, BA_1, \downarrow)$
- $\delta(q_{B_+}, \sigma, A_1) = (q, CA_2, \downarrow)$;
- $\delta(q_{B_-}, \sigma, A_1) = (q_{A_-}, \varepsilon, \uparrow)$;
- $\delta(q_{C_+}, \sigma, A_2) = (q'_{A_2}, \varepsilon, \downarrow)$;
- $\delta(q'_{A_2}, \sigma, A_1) = (q_{A_+}, \varepsilon, \downarrow)$;
- $\delta(q_{C_-}, \sigma, A_2) = (q'_{A_{2-}}, \varepsilon, \uparrow)$;
- $\delta(q'_{A_{2-}}, \sigma, A_1) = (q_{A_-}, \varepsilon, \uparrow)$.

2. $A \leftarrow B \,/\, C$
- $\delta(q, \sigma, A) = (q, BA_1, \downarrow)$
- $\delta(q_{B_+}, \sigma, A_1) = (q_{A_+}, \varepsilon, \downarrow)$
- $\delta(q_{B_-}, \sigma, A_1) = (q_{A_2}, \varepsilon, \uparrow)$
- $\delta(q_{A_2}, \sigma, Z) = (q, CA_2, \downarrow)$
- $\delta(q_{C_+}, \sigma, A_2) = (q_{A_+}, \varepsilon, \downarrow)$
- $\delta(q_{C_-}, \sigma, A_2) = (q_{A_-}, \varepsilon, \uparrow)$;

3. $A \leftarrow \varepsilon$
- $\delta(q, \sigma, A) = (q_{A_+}, \lcurvearrowdown)$

4. $A \leftarrow \mathbf{!}B$
- $\delta(q, \sigma, A) = (q, BA_1, \downarrow)$;
- $\delta(q_{B_\pm}, \sigma, A_1) = (q_{A_\mp}, \varepsilon, \uparrow)$

5. $A \leftarrow a$
- $\delta(q, a, A) = (q_{A_+}, \hookrightarrow)$;
- $\delta(q, b, A) = (q_{A_-}, \lcurvearrowdown)$, here $b \neq a$;

Figure 4 Construction of δ by the PEG G.

Each rule is applied to a configuration of the form $(q, A\alpha \times l : \vec{i}, l)$. In the first case (of concatenation) the automaton pushes the auxiliary symbol A_1 at the same position that A has been pushed (since the invariant 1 holds) and then pushes B. If it reached a configuration of the form $(q_{B_+}, BA_1A\alpha \times l : l : l : \vec{i}, r' + 1)$, then B has successfully parsed the subword $w_l \cdots w_{r'}$ due to invariant 2, then B is popped due to General rules and DPPDA pushes C at the position $r' + 1$ and goes to the main work state q. If then the DPPDA reaches a configuration of the form $(q_{C_+}, CA_2A_1A\alpha \times (r'+1) : (r'+1) : l : l : \vec{i}, r+1)$ we have that the PEG parsed $w_{r'+1} \cdots w_r$ from C and after the sequences of technical pops the automaton comes to the configuration $(q_{A_+}, A\alpha \times l : \vec{i}, r + 1)$ that proves that invariant 2-q_{A_+} holds (the arguments for the other direction are similar).

In the case of reaching the configuration $(q_{C_-}, CA_2A_1A\alpha \times (r'+1) : (r'+1) : l : l : \vec{i}, r+1)$ or $(q_{B_-}, BA_1A\alpha \times l : l : l : \vec{i}, r' + 1)$ earlier, the sequence of pops lead the DPPDA to the configuration $(q_{A_-}, A\alpha \times l : \vec{i}, l)$ that proves that invariant 2-q_{A_-} holds (the arguments for the other direction are similar).

The case of the ordered choice is similar to the case of concatenation. The difference is, that in the case of a configuration $(q_{B_+}, BA_1A\alpha \times l : l : l : \vec{i}, r + 1)$, the automaton reaches the configuration $(q_{A_+}, A\beta \times l : \vec{i}, r + 1)$ via the technical moves, and in the case of $(q_{B_-}, BA_1A\alpha \times l : l : l : \vec{i}, r + 1)$ the automaton reaches the configuration $(q, CA_2A\alpha \times l : l : l : \vec{i}, l)$ after which

$$\text{either } (q, CA_2A\alpha \times l : l : l : \vec{i}, l) \vdash^* (q_{C_+}, CA_2A\alpha \times l : l : l : \vec{i}, r+1) \vdash (q_{A_+}, A\alpha \times l : \vec{i}, r+1),$$

$$\text{or } (q, CA_2A\alpha \times l : l : l : \vec{i}, l) \vdash^* (q_{C_-}, CA_2A\alpha \times l : l : l : \vec{i}, r+1) \vdash^* (q_{A_-}, A\alpha \times l : \vec{i}, l).$$

The analysis of the remaining cases directly follows from the definitions, so we omit it. ◀

5 Linear-Time Simulation of 2DPPDA

Our linear-time simulation algorithm for 2DPPDA is almost the same as S. Cook's algorithm for 2DPDA [5]. One can find the detailed exposition in [2] and [8].

▶ **Theorem 7** ([13]). *Let M be a 2DPPDA. The language $L(M)$ is recognizable in time $O(n)$ in the RAM model. Moreover, there exists an $O(|w|)$ (in RAM) simulation algorithm for M on the input w.*

6 Structural Results

We use the computational model to obtain new structural results about the PELs.

▶ **Lemma 8.** *Let X be a DCFL and Y be a PEL. Then XY is a PEL.*

Proof. We describe a DPPDA M recognizing XY that simulates a DPDA M_X recognizing X and a DPPDA M_Y recognizing Y, constructed by a (well-formed) PEG.

DPPDA M simulates M_X until it reaches an accepting state. Then it pushes the information of the state to the stack, then pushes Z_0 (of M_Y) and simulates M_Y. If M_Y accepts the rest of the input, then the whole input is accepted. Otherwise, M pops symbols from the stack until it reaches the info about the M_X state and continues the simulation until it reaches an accepting state again. This process is continued until either M_Y accepts, or M_X reaches the end of the input (and M_Y rejects ε).

The correctness easily follows from the construction. During the process M tests all the prefixes of the input from X and checks whether the corresponding suffixes belong to Y. ◀

We use the notation PEL, DCFL, and REG for the language classes in formulas (the last one denotes regular languages). Denote by $\Gamma_{\mathsf{REG}}(\mathsf{DCFL})$ the regular closure of DCFLs; this class is defined as follows. $L \in \Gamma_{\mathsf{REG}}(\mathsf{DCFL})$ if there exists a regular expression (RE) R over an alphabet $\Sigma_k = \{a_1, \dots, a_k\}$ and DCFLs $L_1, \dots, L_k$ such that if we replace a_i by L_i in R the resulting expression $\psi(R)$ describes L.

▶ **Lemma 9** ([13]). $\Gamma_{\mathsf{REG}}(\mathsf{DCFL}) \subseteq \mathsf{PEL}$.

We describe only the proof idea. We provided the proof of Lemma 8 to generalize it as follows. In the case of a single concatenation, we have a kind of linear order for an exhaustive search. In the case of $\Gamma_{\mathsf{REG}}(\mathsf{DCFL})$ we will perform an exhaustive search in the order corresponding to a (graph of) deterministic finite automaton (DFA) recognizing R. If a word w on the input belongs to $L \in \Gamma_{\mathsf{REG}}(\mathsf{DCFL})$, then it can be split into subwords $w_1 \cdots w_k = w$ such that there exists a word $\alpha_1 \cdots \alpha_k \in R$ such that $w_i \in L_{\alpha_i}$, where L_{α_i} is the DCFL from the substitution that maps α_i to L_{α_i}. So, the exhaustive search finds the split of w by considering $\alpha_1 \cdots \alpha_k$ in the length-lexicographic order and considering w's subwords $w_i \in L_{\alpha_i}$ ordered by the length. If a word $w_1 \in L_{\alpha_1}$ is the shortest prefix, the DPPDA tries to find the shortest $w_2 \in L_{\alpha_2}$ and so on. If at some point the DPPDA failed to find $w_{j+1} \in L_{\alpha_{j+1}}$, it rollbacks to α_j and tries to find a longer word $w_j \in L_{\alpha_j}$. If it fails, then it rollbacks to α_{j-1} and so on. During the search of w_j, the DPPDA simulates a DPDA M_j recognizing L_{α_j}.

Denote by $\Gamma_{\mathsf{Bool}}(\mathscr{L})$ the *Boolean closure* of the language's class $\mathscr{L}$, i.e., $\Gamma_{\mathsf{Bool}}(\mathscr{L})$ is a minimal class satisfying the conditions:

- $\mathscr{L} \subseteq \Gamma_{\mathsf{Bool}}(\mathscr{L})$
- $\forall A, B \in \Gamma_{\mathsf{Bool}}(\mathscr{L}) : A \cup B, A \cap B, \overline{A} \in \Gamma_{\mathsf{Bool}}(\mathscr{L})$

▶ **Theorem 10.** *The following assertions hold.*

1. $\Gamma_{\mathsf{Bool}}(\Gamma_{\mathsf{REG}}(\mathsf{DCFL})) \subseteq \mathsf{PEL}$.
2. $\Gamma_{\mathsf{REG}}(\mathsf{DCFL}) \cdot \mathsf{PEL} = \mathsf{PEL}$.

Proof. It was shown in [7] that $\Gamma_{\mathsf{Bool}}(\mathsf{PEL}) = \mathsf{PEL}$. We proved that $\Gamma_{\mathsf{REG}}(\mathsf{DCFL}) \subseteq \mathsf{PEL}$, so $\Gamma_{\mathsf{Bool}}(\Gamma_{\mathsf{REG}}(\mathsf{DCFL})) \subseteq \mathsf{PEL}$.

The inclusion $\Gamma_{\mathsf{REG}}(\mathsf{DCFL}) \cdot \mathsf{PEL} \supseteq \mathsf{PEL}$ is obvious ($\{\varepsilon\} \in \Gamma_{\mathsf{REG}}(\mathsf{DCFL})$). The inclusion $\Gamma_{\mathsf{REG}}(\mathsf{DCFL}) \cdot \mathsf{PEL} \subseteq \mathsf{PEL}$ follows from the modification of the simulation algorithm from the proof of Lemma 9 by the simulation step from the proof of Lemma 8: when M_j reaches an accepting state and the corresponding state of $\mathcal{A}$ is an accepting state, M simulates the DPPDA for the PEG. If it successfully parses the suffix, the input is accepted, otherwise, the simulation continues as in the proof of Lemma 9. Recall that it is not known, whether PELs are closed over concatenation (Remark Remark:ConcatClosure), so we have to use Lemma 9 to for left concatenation with $\Gamma_{\mathsf{REG}}(\mathsf{DCFL})$. ◀

▶ **Corollary 11.** *For each $L \in \Gamma_{\mathsf{Bool}}(\Gamma_{\mathsf{REG}}(\mathsf{DCFL}))$ there exists a RAM-machine M that decides, whether $w \in L$ in time $O(|w|)$. In other words, the class $\Gamma_{\mathsf{Bool}}(\Gamma_{\mathsf{REG}}(\mathsf{DCFL}))$ is linear-time recognizable.*

References

1 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. If the Current Clique Algorithms Are Optimal, So is Valiant's Parser. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, FOCS '15, pages 98–117, USA, 2015. IEEE Computer Society. doi:10.1109/FOCS.2015.16.

2 A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley Publishing Company, 1974.

3 Eberhard Bertsch and Mark-Jan Nederhof. Regular Closure of Deterministic Languages. *SIAM J. Comput.*, 29:81–102, 1999.

4 A. Birman and J. D. Ullman. Parsing algorithms with backtrack. In *11th Annual Symposium on Switching and Automata Theory (SWAT 1970)*, pages 153–174, 1970.

5 Stephen A Cook. *Linear time simulation of deterministic two-way pushdown automata*. Department of Computer Science, University of Toronto, 1970.

6 Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 36–47, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/581478.581483.

7 Bryan Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/964001.964011.

8 Robert Glück. Simulation of Two-Way Pushdown Automata Revisited. In *Electronic Proceedings in Theoretical Computer Science*, volume 129, pages 250–258. Open Publishing Association, September 2013.

9 John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979.

10 Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.

11 Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002.

12 Bruno Loff, Nelma Moreira, and Rogério Reis. The computational power of parsing expression grammars. In Mizuho Hoshi and Shinnosuke Seki, editors, *Developments in Language Theory - 22nd International Conference, DLT 2018, Tokyo, Japan, September 10-14, 2018, Proceedings*, volume 11088 of *Lecture Notes in Computer Science*, pages 491–502. Springer, 2018. doi:10.1007/978-3-319-98654-8_40.

13 Alexander Rubtsov and Nikita Chudinov. Computational Model for Parsing Expression Grammars, 2024. arXiv:2406.14911.

14 Alexander A. Rubtsov. A Linear-Time Simulation of Deterministic d-Limited Automata. In *Developments in Language Theory: 25th International Conference, DLT 2021, Porto, Portugal, August 16–20, 2021, Proceedings*, pages 342–354, Berlin, Heidelberg, 2021. Springer-Verlag. doi:10.1007/978-3-030-81508-0_28.

15 Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.

16 Guido van Rossum, Pablo Galindo, and Lysandros Nikolaou. New PEG parser for CPython, 2020. URL: https://peps.python.org/pep-0617/.