Linear Parsing Expression Grammars

Nariyoshi Chida(✉) and Kimio Kuramitsu

Yokohama National University, Yokohama, Japan
nariyoshi-chida-pg@ynu.jp, kimio@ynu.ac.jp

Abstract. PEGs were formalized by Ford in 2004, and have several pragmatic operators (such as ordered choice and unlimited lookahead) for better expressing modern programming language syntax. Since these operators are not explicitly defined in the classic formal language theory, it is significant and still challenging to argue PEGs' expressiveness in the context of formal language theory. Since PEGs are relatively new, there are several unsolved problems. One of the problems is revealing a subclass of PEGs that is equivalent to DFAs. This allows application of some techniques from the theory of regular grammar to PEGs. In this paper, we define Linear PEGs (LPEGs), a subclass of PEGs that is equivalent to DFAs. Surprisingly, LPEGs are formalized by only excluding some patterns of recursive nonterminal in PEGs, and include the full set of ordered choice, unlimited lookahead, and greedy repetition, which are characteristic of PEGs. Although the conversion judgement of parsing expressions into DFAs is undecidable in general, the formalism of LPEGs allows for a syntactical judgement of parsing expressions.

Keywords: Parsing expression grammars · Boolean finite automata · Packrat parsing

1 Introduction

Deterministic finite automata (DFAs) are a simple and fundamental theory in the classic formal language, which allows pattern matching on the input without backtracking. This positive aspect is applied to the implementation of many regular expression engines such as Google RE2 [8] and grep leading to significantly improved performance.

Similarly, the DFA nature is used for faster parsing. For example, a partial conversion of *context-free grammars* (CFGs) into DFAs is studied with ANTLR3/4 by Parr *et al.* [15,16]. In this study, Parr *et al.* achieve better performance of a parser based on CFG by using the conversion. Concretely, the parser decides a nonterminal that should be expanded by using the DFA. That is, DFA conversions remove backtracking while parsing.

In this way, DFAs are used for faster parsing. To the best of our knowledge, however, DFAs are not used for parsing a *parsing expression grammar* (PEG) [7] yet. PEGs are a relatively new and popular foundation for describing syntax, formalized by Ford in 2004. PEGs look very similar to some of the EBNFs or

© Springer International Publishing AG 2017
F. Drewes et al. (Eds.): LATA 2017, LNCS 10168, pp. 275–286, 2017.
DOI: 10.1007/978-3-319-53733-7_20

CFG-based grammar specifications, but differ significantly in that they have *unlimited lookahead* with syntactic predicates and deterministic behaviors with greedy repetition and prioritized choice. Due to these extended operators, PEGs can recognize highly nested languages such as $\{a^n\ b^n\ c^n \mid n > 0\}$, which is not possible in a CFG.

These extended operators raise an interesting and open question on the connection to the formal language theory. In particular, we have expected that a partial DFA conversion brings better performance benefits to the PEG-based parser generation as well as Parr *et al.* However, parsing expressions are obviously more expressive than DFAs, due to recursion which does not appear in regular expressions. Therefore, we require a subclass of PEGs that is equivalent to DFAs for applying DFA techniques to PEGs.

The main contribution of this paper is that we reveal a subclass of PEGs that is equivalent to DFAs. We formalize the subclass as *linear parsing expression grammars* (LPEGs). Surprisingly, LPEGs are formalized by excluding only some patterns of recursive nonterminal in PEGs, and include the full set of prioritized choice, unlimited lookahead, and greedy repetition, which are unique to PEGs. Furthermore, the formalism of LPEGs allows a partial conversion of a PEG into DFAs. Since converting into DFAs can eliminate backtracking, the partial conversion would lead to further optimization of the parser generator.

The rest of this paper proceeds as follows. Section 2 describes the formalism of LPEGs and shows the relationship between LPEGs and PEGs. Section 3 shows a regularity of LPEGs. Section 4 briefly reviews related work. Section 5 is the conclusion.

2 Linear PEG

In this section, we describe the formalism of *linear parsing expression grammars* (LPEGs). LPEGs are a subclass of PEGs equivalent to DFAs, and LPEGs are formalized by excluding patterns of recursive nonterminals that are followed by expressions. By the exclusion, the syntax of an LPEG is limited to right-linear. Thus, we can simply consider an LPEG as a PEG where the syntax is right-linear.

To begin with, we describe PEG operators in Sect. 2.1. Then, we show the formalism of LPEGs in Sect. 2.2. Finally, we describe language properties in Sect. 2.3.

2.1 PEG Operators

Table 1 shows the summary of PEG operators used throughout this paper.

The string ‘abc’ exactly matches the same input, while [abc] matches one of these terminals. The `.` operator matches any single terminal. The e`?`, e`*`, and e`+` expressions behave as in common regular expressions, except that they are greedy and match until the longest position. The $e_1\ e_2$ attempts two expressions e_1 and e_2 sequentially, backtracking the starting position if either expression

Table 1. PEG operators

PEG	Type	Proc.	Description
' '	Primary	5	Matches text
[]	Primary	5	Matches character class
.	Primary	5	Any character
A	Primary	5	Non-terminal application
(e)	Primary	5	Grouping
e?	Unary suffix	4	Option
e*	Unary suffix	4	Zero-or-more repetitions
e+	Unary suffix	4	One-or-more repetitions
&e	Unary prefix	3	And-predicate
!e	Unary prefix	3	Not-predicate
$e_1 e_2$	Binary	2	Sequence
e_1/e_2	Binary	1	Prioritized choice

fails. The choice e_1/e_2 first attempts e_1 and then attempts e_2 if e_1 fails. The expression &e attempts e without any terminal consuming. The expression !e fails if e succeeds, but succeeds if e fails.

We consider the any character . expression to be a choice of all single terminals $(a/b/\ldots/c)$ in Σ. As long as any special cases are not noted, we treat the any character as a syntax sugar of such a terminal choice.

Likewise, many convenient notations used in PEGs such as character class, option, and one or more repetition are treated as syntax sugars:

$$
\begin{array}{lll}
[abc] & = a/b/c & \text{character class} \\
e^{+} & = ee^{*} & \text{one or more repetition} \\
e? & = e/\varepsilon & \text{option} \\
\&e & = !!e & \text{and-predicate}
\end{array}
$$

2.2 Definition of LPEGs

Definition 1. *A linear parsing expression grammar (LPEG) is defined by a 4-tuple* $G = (N_G, \Sigma, P_G, e_s)$*, where* N_G *is a finite set of nonterminals,* Σ *is a finite set of terminals,* P_G *is a finite set of production rules, and* e_s *is a linear parsing expression termed the start expression. A linear parsing expression* e *is a parsing expression with the syntax according to BNF shown in Fig. 1.* p *in Fig. 1 is a nonterminal-free parsing expression (n-free parsing expression). An n-free parsing expression* p *is a parsing expression such that the expression doesn't contain nonterminals. Each rule in* P_G *is a mapping from a nonterminal* $A \in N_G$ *to a linear parsing expression* e*. We write* $P_G(A)$ *to denote an associated expression* e *such that* $A \leftarrow e \in P_G$*.*

$$
\begin{array}{rcll}
e & ::= & p & \\
& | & p\ A & \\
& | & p\ e & \\
& | & e/e & \\
& | & !e\ e & \\
p & ::= & \varepsilon & \text{empty} \\
& | & a & \text{character} \\
& | & . & \text{any character} \\
& | & p\ p & \text{sequence} \\
& | & p/p & \text{prioritized choice} \\
& | & p* & \text{zero or more repetition} \\
& | & !p & \text{not-predicate}
\end{array}
$$

Fig. 1. Syntax of a linear parsing expression

We show two examples of an LPEG and an example of a PEG but not an LPEG.

Example 2. $G = (\{A, B\}, \{a, b, c\}, \{A \leftarrow aA/bB/c, B \leftarrow aB/bA/c\}, A)$ is an LPEG.

Example 3. $G = (\{A\}, \{a, b\}, \{A \leftarrow !(aA)aA/b\}, A)$ is an LPEG.

Example 4. $G = (\{A, B\}, \{a, b\}, \{A \leftarrow aAa/B*, B \leftarrow aB/b\}, A)$ is not an LPEG. Note that aAa and $B*$ are not derived from the above syntax.

All subsequent use of the unqualified term "grammar" refers specifically to linear parsing expression grammars as defined here, and the unqualified term "expression" refers to linear parsing expressions. We use the variables $a, b, c \in \Sigma$, $A, B, C \in N_G$, $x, y, z \in \Sigma^*$, and e for linear parsing expressions.

2.3 Language Properties

In this section, we define a language recognized by LPEGs. We use a function *consume* to define the language. $consume(e, x) = y$ denotes that the expression e succeeds on the input string x and consumes y. $consume(e, x) = f$ denotes that the expression e fails on the input string x.

Definition 5. *Let $G = (N_G, \Sigma, P_G, e_s)$ be an LPEG, let e be an expression. The language generated by e is a set of all strings over Σ:*

$$L_G(e) = \{x \mid x \in \Sigma^*,\ y \textit{ is a prefix of } x,\ consume(e, x) = y\}.$$

Definition 6. *Let $G = (N_G, \Sigma, P_G, e_s)$ be an LPEG. The language generated by a grammar G is a set of all strings over Σ:*

$$L(G) = L_G(e_s).$$

3 Regularity

In this section, we prove that LPEGs are a class that is equivalent to DFAs. To prove this, we show that for any LPEG G there exists a DFA D such that $L(G) = L(D)$ and for any DFA D there exists an LPEG G such that $L(D) = L(G)$. We show the former in Sect. 3.1 and the latter in Sect. 3.2.

3.1 From LPEGs to DFAs

We show that for any LPEG G there exists a DFA D such that $L(G) = L(D)$. This can be proved by translating LPEGs into *boolean finite automata* (BFAs) [4].

A BFA is a generalized *nondeterministic finite automaton* (NFA). The difference between NFAs and BFAs is a representation of a state under transition. The state under transition on NFAs can be represented as a boolean function consisting of logical OR and boolean variables. On the other hand, the state under transition on BFAs can be represented as a boolean function consisting of logical AND, logical OR, logical NOT, constant values (i.e. *true* and $false$), and boolean variables.

There are two reasons for using BFAs. One is to handle not-predicates. We can represent these predicates as a boolean function by using logical AND and logical NOT. Another reason is that BFAs can be converted into DFAs ([4], Theorem 2). Thus, LPEGs can be converted into DFAs if we can convert LPEGs into BFAs.

In the next section we describe basic definitions and notations of BFAs. In Sect. 3.1, we show that LPEGs can be converted into BFAs.

Boolean Finite Automata

Definition 7. *A* boolean finite automaton *(BFA) is a 5-tuple $B = (Q, \Sigma, \delta, f^0, F)$. $Q = \{q_1, q_2, \ldots, q_n\}$ is a finite non-empty set of states. Σ is a finite set of terminals. $\delta : Q \times \Sigma \to V_Q$ is a transition function that maps a state and a terminal into a boolean function of boolean variables that correspond to the states $q_1, q_2, \ldots, q_n$ in the set of boolean functions V_Q. $f^0 \in V_Q$ is an initial boolean function. F is a finite set of accepting states. We use q_i as a boolean variable that corresponds to a state $q_i \in Q$.*

Let f be a boolean function in V_Q. The transition function δ is extended to $V_Q \times \Sigma^*$ as follows:

$$\begin{aligned}
\delta(f, \epsilon) &= f \\
\delta(f, a) &= f(\delta(q_1, a), \ldots, \delta(q_n, a)) \\
\delta(f, aw) &= \delta(\delta(f, a), w)
\end{aligned}$$

A language accepted by a BFA is defined as follows:

Definition 8. *Let B be a BFA and $x \in \Sigma^*$. $x \in L(B)$ iff $\delta(f^0, x)(c_1, \ldots, c_n) = true$, where $c_i = true$ if $q_i \in F$, otherwise $false$.*

From LPEGs to BFAs. We show a conversion from an LPEG into a BFA. The conversion consists of three steps. In the first step, we modify an LPEG in order to simplify the conversion. In the second step, we convert a modified LPEG into a BFA. However, the BFA is incomplete in this step, since the conversion handles nonterminals as temporary boolean variables to avoid an infinite loop by recursions. In the final step, we replace the temporary boolean variables in a BFA with initial functions of the nonterminals.

First, we modify an LPEG by applying a modification function C_G shown in Definition 9. By applying the modification, new production rules for nonterminals in not-predicates are added to the LPEG. We apply this modification to LPEGs, because we consider a nonterminal A in a not-predicate and a nonterminal A that is not in a not-predicate as distinct.

Definition 9. *Let* $G = (N_G, \Sigma, P_G, e_s)$ *be an LPEG.* $C_G(G) = (N_G \cup N_{G'}, \Sigma, P_{G_1} \cup P_{G_2}, e_{s'})$*, where* $P_{G_1} = \{A \leftarrow C_n(e_A) \mid e_A \in P_G\}$*,* $P_{G_2} = \{A' \leftarrow e_{A'} \mid e_{A'} = \mathit{copy}(e_A), e_A \in P_G\}$*,* $N_{G'} = \{A' \mid e_{A'} \in P_{G_2}\}$*,* $e_{s'} = C_n(e_s)$*.*

In the modification function, we use an auxiliary function C_n. C_n is a function for modification of a production rule. We show the definition of C_n in Definition 10. In the following definition, we use a function *copy*. $\mathit{copy}(e) = e'$ denotes that a nonterminal A is renamed as A' if the nonterminal A is not already A' and the other expressions are same. We assume that there does not exist A' in an LPEG before the modification.

For example, $\mathit{copy}(aA/!(bB)b/c) = aA'/!(bB')b/c$ and $\mathit{copy}(\mathit{copy}(!(aA))) = \mathit{copy}(!(aA')) = !(aA')$.

Definition 10.

$$\begin{aligned}
C_n(p) &= p \\
C_n(p\ A) &= p\ A \\
C_n(p\ e) &= p\ C_n(e) \\
C_n(e/e) &= C_n(e)/C_n(e) \\
C_n(!e\ e) &= !(\mathit{copy}(e))\ C_n(e)
\end{aligned}$$

We show an example of the modification as follows:

Example 11. $G = (\{A, B\}, \{a, b, c, d\}, \{A \leftarrow !(aA/bB/c)d, B \leftarrow aB/b\}, A)$ is an LPEG. Then, $C_G(G) = (\{A, B, A', B'\}, \{a, b, c, d\}, P_{G_1} \cup P_{G_2}, A)$, where P_{G_1} consists of the following rules:

$$\begin{aligned}
A &\leftarrow !(aA'/bB'/c)d \\
B &\leftarrow aB/b
\end{aligned}$$

P_{G_2} consists of the following rules:

$$\begin{aligned}
A' &\leftarrow !(aA'/bB'/c)d \\
B' &\leftarrow aB'/b
\end{aligned}$$

Secondly, we describe the conversion from modified LPEGs to BFAs with temporary boolean variables. This is inspired by [13].

In this function, a set of accepting states is divided into two sets, F and P, in order to simplify the construction of BFAs. The set P is a set of accepting states for not-predicates. We assume that the names of boolean variables are distinct in the conversion. We write a temporary boolean variable of a nonterminal A as f_{tmp_A}. A function $\phi(f_1, f_2, F)$ converts the boolean function f_1 by replacing a boolean variable s in f_1 with $s \vee f_2$ if $s \in F$. For example, let $f_1 = (q_1 \wedge q_2) \vee q_3$, $f_2 = q_4$ and $F = \{q_2, q_3\}$, where q_1, q_2, q_3 and q_4 are boolean variables. Then, $\phi(f_1, f_2, F) = (q_1 \wedge (q_2 \vee q_4)) \vee (q_3 \vee q_4)$. A function *copy* used in $T_B(\texttt{!}e)$ is the same definition as in Definition 10. Note that the BFA converted by the following function accepts the full match of the expressions. A BFA that accepts the same language with the LPEG is written as $T_B(e_s\texttt{.*})$.

$$
\begin{aligned}
T(G) &= (Q, \Sigma, \delta', f^0, F \cup P) \\
&\qquad \textit{where} \quad (Q, \Sigma, \delta, f^0, F, P) = T_B(e_s) \\
&\qquad \textit{and} \quad \delta' = \{((s, \texttt{.}), s) \mid s \in P\} \cup \delta \\
T_B(\epsilon) &= (\{s\}, \Sigma, \{\}, s, \{s\}, \{\}) \\
T_B(a) &= (\{s, t\}, \Sigma, \{((s, a), t)\}, s, \{t\}, \{\}) \\
T_B(\texttt{!}e) &= (Q \cup \{s\}, \Sigma, \delta, s \wedge \overline{f^0}, \{s\}, F \cup P) \\
&\qquad \textit{where} \quad (Q, \Sigma, \delta, f^0, F, P) = T_B(\textit{copy}(e)) \\
T_B(e_1 e_2) &= (Q_1 \cup Q_2, \Sigma, \delta, \phi(f_1^0, f_2^0, F_1), F_2, P_1 \cup P_2) \\
&\qquad \textit{where} \quad (Q_1, \Sigma, \delta_1, f_1^0, F_1, P_1) = T_B(e_1), \\
&\qquad (Q_2, \Sigma, \delta_2, f_2^0, F_2, P_2) = T_B(e_2) \\
&\qquad \textit{and} \quad \delta = \{((s, a), \phi(t, f_2^0, F_1)) \mid ((s, a), t) \in \delta_1\} \cup \delta_2 \\
T_B(e_1 / e_2) &= T_B(e_1 \mid \texttt{!}e_1 e_2) \\
T_B(e_1 \mid e_2) &= (Q_1 \cup Q_2, \Sigma, \delta_1 \cup \delta_2, f_1^0 \vee f_2^0, F_1 \cup F_2, P_1 \cup P_2) \\
&\qquad \textit{where} \quad (Q_1, \Sigma, \delta_1, f_1^0, F_1, P_1) = T_B(e_1) \\
&\qquad \textit{and} \quad (Q_2, \Sigma, \delta_2, f_2^0, F_2, P_2) = T_B(e_2) \\
T_B(e\texttt{*}) &= T_B(e^\star \texttt{!}e) \\
T_B(e^\star) &= (Q \cup \{s\}, \Sigma, \delta', s \vee f^0, F \cup \{s\}, P) \\
&\qquad \textit{where} \quad (Q, \Sigma, \delta, f^0, F, P) = T_B(e) \\
&\qquad \textit{and} \quad \delta' = \{((s, a), \phi(t, f^0, F)) \mid ((s, a), t) \in \delta\} \\
T_B(A) &= \begin{cases} T_B(P_G(A)) \\ (\texttt{first time to apply the function to a nonterminal A}) \\ (\{\}, \Sigma, \{\}, f_{tmp_A}, \{\}, \{\}) \\ (\texttt{otherwise}) \end{cases}
\end{aligned}
$$

Basically, this conversion is based on Thompson's construction [17]. In this conversion, the construction of prioritized choices `/` and zero-or-more repetitions `*` is reduced to the construction of alternations $|$ and zero-or-more repetitions $\star$ in a regular expression. We rewrite them precisely as follows:

$$e_1/e_2 \Rightarrow e_1 \mid !e_1 e_2$$
$$e* \Rightarrow e^{\star}!e$$

$\alpha \Rightarrow \beta$ denotes rewriting α as β. For example, a/aa is rewritten as $a \mid (!a)aa$ and $a*a$ is rewritten as $a^{\star}\,(!a)a$.

In the rewriting of prioritized choices, the not-predicate $!e_1$ is added to the front of e_2. By the rewriting, e_2 does not match an input string when the input string matches e_1.

Furthermore, in the rewriting of zero-or-more repetitions, the not-predicate $!e$ is added to the back of $e^{\star}$. By this rewriting, we can eliminate the ambiguity of $e^{\star}$ and the repetition becomes a longest match.

The function T handles the nonterminals in the same way as a conversion from a right-linear grammar to an NFA [11]. In the conversion from a right-linear grammar to an NFA, a nonterminal is handled as an initial state of the NFA. In the same way, in the function T, a nonterminal is handled as an initial function of the BFA.

Finally, we replace temporary variables with the initial functions of the nonterminals. We show the replacement in the following example of conversion from an LPEG to a BFA.

Example 12. Let $G = (\{A\}, \{a, b\}, \{A \leftarrow aA/b\}, A)$ be an LPEG. The language of the LPEG $L(G) = \{a^i b \mid i \geq 0\}$.

First, we modify the LPEG G as follows: $G = C_G(G)$, where $C_G(G) = (\{A, A'\}, \{a, b\}, \{A \leftarrow aA/b, A' \leftarrow aA'/b\}, A)$.

Secondly, we convert the LPEG G to a BFA B with temporary boolean variables. As a result of the conversion, we get the BFA $B = (Q, \{a, b\}, \delta, q_0 \vee ((q_{11} \vee q_{12}) \wedge \overline{q_2}), \{q_{10}, q_{13}\})$, where δ is shown in Table 2. For simplicity, we consider transitions that are not in Table 2 return $false$.

Table 2. The transition function δ with temporary boolean variables

	$\delta(state, terminal)$	TERMINAL	
		a	b
STATE	q_0	$q_1 \vee f_{tmp_A}$	$false$
	q_2	$q_3 \vee (q_4 \vee ((q_8 \vee q_9) \wedge \overline{q_6}))$	$false$
	q_4	$q_5 \vee f_{tmp_{A'}}$	$false$
	q_6	$q_7 \vee f_{tmp_{A'}}$	$false$
	q_9	$false$	q_{10}
	q_{10}	q_{10}	q_{10}
	q_{12}	$false$	q_{13}

Finally, we replace temporary boolean variables with the initial functions. In this BFA, there are two temporary boolean variables, f_{tmp_A} and $f_{tmp_{A'}}$. f_{tmp_A} is replaced by $q_0 \vee ((q_{11} \vee q_{12}) \wedge \overline{q_2})$. $f_{tmp_{A'}}$ is replaced by $q_4 \vee ((q_8 \vee q_9) \wedge \overline{q_6})$. The transition function δ is shown in Table 3.

Table 3. The transition function δ

	$\delta(state, terminal)$	Terminal	
		a	b
State	q_0	$q_1 \vee q_0 \vee ((q_{11} \vee q_{12}) \wedge \overline{q_2})$	$false$
	q_2	$q_3 \vee (q_4 \vee ((q_8 \vee q_9) \wedge \overline{q_6}))$	$false$
	q_4	$q_5 \vee q_4 \vee ((q_8 \vee q_9) \wedge \overline{q_6})$	$false$
	q_6	$q_7 \vee q_4 \vee ((q_8 \vee q_9) \wedge \overline{q_6})$	$false$
	q_9	$false$	q_{10}
	q_{10}	q_{10}	q_{10}
	q_{12}	$false$	q_{13}

The BFA B accepts an input string b.

$$\begin{aligned}\delta(q_0 \vee ((q_{11} \vee q_{12}) \wedge \overline{q_2}), b) &= false \vee ((false \vee q_{13}) \wedge \overline{false}) \\ &= false \vee ((false \vee true) \wedge \overline{false}) \\ &= true\end{aligned}$$

In the same way, we can check that the BFA B rejects an input string a.

In order to define a function *consume* for BFAs, we define two evaluate function, $eval_F$ and $eval_P$. $eval_F$ takes a boolean function f and a set of accepting state F and returns a boolean function f' that replaced boolean variables $q_i \in F$ in f with *true*. For example, let $f = q_0 \wedge (q_1 \vee q_2)$ and $F = \{q_1\}$. Then $eval_F(f, F) = q_0 \wedge (true \vee q_2) = q_0 \wedge q_2$. $eval_P$ takes a boolean function f and a set of accepting state of not-predicates P and returns a boolean value that is a result of a replacement of a boolean variables $q_i \in P$ in f with *true*, otherwise *false*. For example, let $f = q_0 \wedge \overline{q_1}$ and $P = \{q_0\}$. Then, $eval_P(f, P) = true \wedge \overline{false} = true$.

We define a function *consume* for a BFA B. $consume(B, w) = x$ denotes that $eval_P(\delta(eval_F(\delta(f^0, x), F), y), P) = true$ for an input string $w = xyz$. $consume(B, w) = f$ if there is no such x.

Theorem 13. *Let $G = (N_G, \Sigma, P_G, e_s)$ be an LPEG modified by C_G in Definition 9. Let $B = T((N_G, \Sigma, P_G, e_s \,.\, *))$ and B has already replaced the temporary variables with initial functions. Then, $L(G) = L(B)$.*

Proof. The proof is by induction on the structure of linear parsing expression e. We assume that $T_B(e)$ is a BFA such that $consume(T_B(e), x) = consume(e, x)$, where $x \in \Sigma^*$.

Theorem 14. *For any LPEG G there exists a DFA D such that $L(G) = L(D)$.*

Proof. By Theorem 13, LPEGs can be converted into BFAs. BFAs can be converted into DFAs.

3.2 From a DFA to an LPEG

An arbitrary regular expression can be converted into a PEG [12,14]. In this section, we say that for any DFA D there exists an LPEG G such that $L(D) = L(G)$. To prove this, we show that a PEG converted from a regular expression by [12] is an LPEG, since DFAs can be converted into equivalent regular expressions [9].

Medeiros et al. studied the conversion and they showed the conversion function as a function Π [12]. The definition of the function Π is shown in Definition 15. The function $\Pi(r, G)$ takes a regular expression r and a continuation grammar $G = (N_G, \Sigma, P_G, e_s)$, and returns a PEG. The continuation grammar is defined by a PEG $G_0 = (\{\}, \Sigma, \{\}, \varepsilon)$ for the first application.

Definition 15 (in [12]).

$$\begin{aligned}
\Pi(\epsilon, G) &= G \\
\Pi(a, G) &= (N_G, \Sigma, P_G, ae_s) \\
\Pi(r_1 r_2, G) &= \Pi(r_1, \Pi(r_2, G)) \\
\Pi(r_1 \mid r_2, G) &= (N_G{''}, \Sigma, P_G{''}, e_s{'} / e_s{''}) \\
&\qquad \textit{where} \quad (N_G{''}, \Sigma, P_G{''}, e_s{''}) = \Pi(r_2, (N_G{'}, \Sigma, P_G{'}, e_s)) \\
&\qquad \textit{and} \quad (N_G{'}, \Sigma, P_G{'}, e_s{'}) = \Pi(r_1, G) \\
\Pi(r^\star, G) &= (N_G{'}, \Sigma, P_G{'} \cup \{A \leftarrow e_s{'} / e_s\}, A) \ \textit{with} \ A \notin N_G \\
&\qquad \textit{and} \quad (N_G{'}, \Sigma, P_G{'}, e_s{'}) = \Pi(r, (N_G \cup \{A\}, \Sigma, P_G, A))
\end{aligned}$$

Theorem 16. *Let r be a regular expression and $\Pi(r, G_0) = G$. The PEG G is an LPEG.*

Proof. We assume that if G is an LPEG, then $\Pi(r, G)$ is also an LPEG. For any regular expression r, we check whether the assumption is correct. If so, $\Pi(r, G_0)$ is an LPEG since G_0 is obviously an LPEG.

1. Case $r = \epsilon$
   By induction hypothesis, G is an LPEG.
2. Case $r = a$
   By induction hypothesis, e_s is a linear parsing expression. Since $ae_s = pe$, (N_G, Σ, P_G, ae_s) is an LPEG.
3. Case $r = r_1 r_2$
   Since G is an LPEG, $\Pi(r_2, G)$ is an LPEG. Therefore, $\Pi(r_1, \Pi(r_2, G))$ is also an LPEG.
4. Case $r = r_1 \mid r_2$
   $\Pi(r_1, G)$ is an LPEG. Since e_s is a linear parsing expression, $\Pi(r_2, (N_G{'}, \Sigma, P_G{'}, e_s))$ is also an LPEG. Therefore, $e_s{'}$ and $e_s{''}$ are a linear parsing expression. Since $e_s{'} / e_s{''} = e / e$, $(N_G{''}, \Sigma, P_G{''}, e_s{'} / e_s{''})$ is an LPEG.
5. Case $r = r^\star$
   Since a nonterminal $A (A = pA)$ is a linear parsing expression, $(N_G \cup \{A\}, \Sigma, P_G, A)$ is an LPEG and $\Pi(r, (N_G \cup \{A\}, \Sigma, P_G, A))$ is also an LPEG. Since $e_s{'} / e_s = e / e$, $(N_G{'}, \Sigma, P'_G \cup \{A \leftarrow e_s{'} / e_s\}, A)$ is an LPEG.

Hence, $\Pi(r, G_0)$ is an LPEG.

Theorem 17. *For any DFA D there exists an LPEG G such that $L(D) = L(G)$.*

Proof. A DFA D can be converted into a regular expression r. By Theorem 16, r can be converted into an LPEG.

Consequently, we derive the following theorem.

Theorem 18. *LPEGs are a class that is equivalent to DFAs.*

Proof. By Theorem 14, for any LPEG G there exists a DFA D such that $L(G) = L(D)$. In addition, by Theorem 17, for any DFA D there exists an LPEG G such that $L(D) = L(G)$. Hence, LPEGs are a class that is equivalent to DFAs.

4 Related Work

Birman and Ullman showed formalism of recognition schemes as TS and gTS [2,3]. TS and gTS were introduced in [1] as TDPL and GTDPL, respectively. A PEG is a development of GTDPL. In this paper, we showed a subclass of PEGs that is equivalent to DFAs, which would lead to more optimized PEG-based parser generator such as [10].

Morihata showed a translation of regular expression with positive and negative lookaheads into finite state automata [13]. He used a *boolean finite automata* (BFAs) [4], that is, *alternating finite automata* [5,6], to represent positive and negative lookaheads of regular expressions as finite automata. We showed that a PEG excluding nonterminals can be converted into an equivalent regular expression with positive and negative lookaheads. Therefore, these PEGs can be converted into a DFA by applying Morihata's translation. However, nonterminals were not covered. In this paper, we showed a translation to handle nonterminals on Morihata's translation.

5 Conclusion

In this study, we formalized a subclass of PEGs that is equivalent to DFAs. In the process of proving the equivalence of the class and DFAs, we showed the conversion from LPEGs into BFAs. Since BFAs can be converted into DFAs, we can convert these LPEGs into DFAs.

One of our motivations is to achieve speed up of runtime by processing a part of a PEG such that the part is regular by using DFAs. To achieve this, we have to check whether the part of a PEG is regular. However, this is undecidable. On the other hand, it is decidable whether a PEG is an LPEG. Thus, we can check whether the part of a PEG is an LPEG and convert the part into DFAs. Since DFAs eliminate backtracking, it would lead to further optimizations of the parser generator.

As a future study, we aim to propose an algorithm for detecting a part of a PEG such that backtracking becomes necessary.

References

1. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling. Prentice-Hall Inc., Upper Saddle River (1972)
2. Birman, A.: The TMG recognition schema. Ph.D. thesis aAI7101582, Princeton, NJ, USA (1970)
3. Birman, A., Ullman, J.D.: Parsing algorithms with backtrack. Inf. Control **23**(1), 1–34 (1973). http://www.sciencedirect.com/science/article/pii/S0019995873908516
4. Brzozowski, J., Leiss, E.: On equations for regular languages, finite automata, and sequential networks. Theoret. Comput. Sci. **10**(1), 19–35 (1980). http://www.sciencedirect.com/science/article/pii/0304397580900699
5. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. J. ACM **28**(1), 114–133 (1981). http://doi.acm.org/10.1145/322234.322243
6. Fellah, A., Jürgensen, H., Yu, S.: Constructions for alternating finite automata. Int. J. Comput. Math. **35**(1–4), 117–132 (1990). http://dx.doi.org/10.1080/00207169008803893
7. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Proceedings of 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 111–122. ACM, New York (2004). http://doi.acm.org/10.1145/964001.964011
8. Google: Re2. https://github.com/google/re2
9. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2006)
10. Kuramitsu, K.: Nez: practical open grammar language. In: Proceedings of 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, pp. 29–42. ACM, New York (2016). http://doi.acm.org/10.1145/2986012.2986019
11. Linz, P.: An Introduction to Formal Language and Automata. Jones and Bartlett Publishers Inc., USA (2006)
12. Medeiros, S., Mascarenhas, F., Ierusalimschy, R.: From regexes to parsing expression grammars. Sci. Comput. Program. Part A **93**, 3–18 (2014). http://www.sciencedirect.com/science/article/pii/S0167642312002171
13. Morihata, A.: Translation of regular expression with lookahead into finite state automaton. Comput. Softw. **29**(1), 1_147–1_158 (2012)
14. Oikawa, M., Ierusalimschy, R., Moura, A.L.D.: Converting regexes to parsing expression grammars
15. Parr, T., Fisher, K.: Ll(*): the foundation of the ANTLR parser generator. In: Proceedings of 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 425–436. ACM, New York (2011). http://doi.acm.org/10.1145/1993498.1993548
16. Parr, T., Harwell, S., Fisher, K.: Adaptive ll(*) parsing: the power of dynamic analysis. SIGPLAN Not. **49**(10), 579–598 (2014). http://doi.acm.org/10.1145/2714064.2660202
17. Thompson, K.: Programming techniques: regular expression search algorithm. Commun. ACM **11**(6), 419–422 (1968). http://doi.acm.org/10.1145/363347.363387