Journal of Information Processing Vol.25 975–982 (Dec. 2017)

[DOI: 10.2197/ipsjjip.25.975]

Regular Paper

Parsing Expression Grammars with Unordered Choices

Nariyoshi Chida[1,†1,a)] Kimio Kuramitsu[1,b)]

Received: February 6, 2017, Accepted: June 23, 2017

Abstract: *Parsing expression grammars* (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. In addition, PEGs can be parsed in a linear time by using recursive-descent parsing and memoization. In this way, PEGs have a lot of positive aspects. On the other hand, it is known that ordered choices defy intuition. They may cause bugs. This is due to a priority of an ordered choice. To avoid this, unordered choices are required. In this paper, we define a *parsing expression grammar with unordered choices* (PEGwUC), an extension of a PEG with unordered choices. By the extension, it is expected that a PEGwUC includes both a PEG and a *context-free grammar* (CFG), and this allows us to write a grammar more intuitively. Furthermore, we show an algorithm for parsing a PEGwUC. The algorithm runs in a linear time when a PEGwUC does not include unordered choice and in a cubic time in worst-case running time.

Keywords: parsing expression grammars, regular expressions, packrat parsing

1. Introduction

In 2004, a new formal grammar, called *parsing expression grammar* (PEG), was introduced by Bryan Ford [7]. PEGs are foundations for describing syntax and have several pragmatic operators (such as ordered choice and unlimited lookahead) for better expressing modern programming language syntax. In addition, PEGs look very similar to some of the *context-free grammar* (CFG)-based grammar specifications, but differ significantly, in that they have unlimited lookahead with syntactic predicates and deterministic behaviors with greedy repetition and ordered 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.

Behavior of a choice operator in a PEG is deterministic. That is, the choice operator attempts to match in the order of the alternates and finishes the matching immediately if the alternate succeeds. For example, let $/$ be a choice operator in a PEG. An expression aa/a matches only aa for an input string aa since the alternate aa succeeds.

The deterministic behavior of the choice operator has positive aspects. One of the positive aspects is that the behavior does not cause problems due to ambiguity of a choice operator. For example, it does not yield the classic dangling else problem.

The deterministic behavior of the choice operator, however, may cause a bug. This is because the choice operator defies intuition [12]. A parsing expression a/aa is one of the examples. In a regular expression, $a \mid aa$ matches a and aa. However, in a PEG, the expression only matches a because the choice first attempts a and then attempts aa if a fails. In addition, an expression that includes a recursion is more complexity, for example, the expression $A \leftarrow aAa/aa$. A is a nonterminal and this is a placeholder for patterns of terminals in common with CFGs. Let a^{10} be an input string. Then, intuitively, we consider the expression consumes a^{10}. However, contrary to our intuition, the expression consumes a^4.

In this paper, we show an approach to avoid the bugs due to the non-intuitive behaviors of ordered choice operators. The non-intuitive behaviors are caused by finishing the matching of the ordered choice at an unintended place for us. That is, the ordered choice may not attempt all alternates of the choice. Therefore, we address extending a PEG by adding another choice operator that attempts all alternates of the choice, that is, an unordered choice operator $\mid$. By the extension, we can avoid the bugs by using an unordered choice instead of an ordered choice.

The main contribution of this paper is that we formalize the extended PEG and present the parsing algorithm. We introduce the extended PEG as a *parsing expression grammar with unordered choices* (PEGwUC). By the extension, it is expected that a PEGwUC includes both a PEG and a CFG. Furthermore, the parsing algorithm allows us to parse a PEGwUC in a linear time if the PEGwUC does not include unordered choice and in a cubic time in worst-case running time. We show the implementation and the performance to check the runtime.

The rest of this paper is organized as follows. In Section 2, we describe the formalism of PEGwUC and the semantics and properties. In Section 3, we give an algorithm for generating a PEGwUC parser. In Section 4, we consider the time complexity of a PEGwUC parser. In Section 5, we show experimental results of PEGwUC parsers generated by the algorithm in Section 3. In Section 6, we briefly review related work. Section 7 provides the conclusion.

1 Graduate School of Electronic and Computer Engineering, Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
†1 Presently with NTT secure platform laboratories
This work has been done when the first author was at the Yokohama National University.
a) chida-nariyoshi-pg@ynu.jp
b) kimio@ynu.ac.jp

2. PEGwUC

In this section, we describe the formalism of *parsing expression grammars with unordered choices* (PEGwUC). A PEGwUC is an extension of a PEG with unordered choices. The extension allows us to write a grammar more intuitively.

2.1 Grammars

A PEGwUC is an extended formalism of a PEG, defined in Ref. [7]. Most grammar constructs come from those of PEG, while our major extension is the introduction of an unordered choice operator, which we denote $\mid$ in this paper.

We start by defining a grammar tuple for a PEGwUC.

Definition 1 (PEGwUC)**.** *A parsing expression grammar with unordered choices (PEGwUC) is defined by a 4-tuple* $G = (N, \Sigma, R, e_s)$*, where* N *is a finite set of nonterminals,* Σ *is a finite set of terminals,* R *is a finite set of rules, and* e_s *is a parsing expression with unordered choices termed start expression.*

We use $A \leftarrow e$ for a rule in R, which is a mapping from a nonterminal $A \in N$ to a parsing expression with unordered choices e. We write $R(A)$ to represent an expression e, which is associated by $A \leftarrow e$.

A *parsing expression with unordered choices* e is a main specification of describing syntactic constructs. **Figure 1** shows the syntax of a parsing expression with unordered choices.

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

The interpretation of PEGwUC operators comes exactly from PEG operators. That is, the empty ε matches the empty string. The terminal a exactly matches the same input character a. The any character $.$ matches any single terminal. The nonterminal A attempts the expression $R(A)$. The sequence $e_1\ e_2$ attempts two expressions e_1 and e_2 sequentially, backtracking the starting position if either expression fails. The ordered choice e_1/e_2 first attempts e_1 and then attempts e_2 if e_1 fails. The zero-or-more repetition $e*$ behave as in common regular expressions, except that they are greedy and match until the longest position. The not-predicate $!e$ attempts e without any terminal consuming and it fails if e succeeds, but succeeds if e fails.

An important extension to PEG operators is the unordered choice, $e_1 \mid e_2$. Intuitively, this unordered choice works as the alternation of regular expression. That is, the unordered choice attempts both e_1 and e_2. If the unordered choice matches both e_1 and e_2, it causes two possibilities and the subsequent behavior becomes non-deterministic.

e	$::=$	ε	empty
	$\mid$	a	character
	$\mid$	$.$	any character
	$\mid$	A	nonterminal
	$\mid$	$e\ e$	sequence
	$\mid$	e/e	ordered choice
	$\mid$	$e \mid e$	unordered choice
	$\mid$	$e*$	zero-or-more repetition
	$\mid$	$!e$	not-predicate

Fig. 1 Syntax of a parsing expression with unordered choices.

The precedence of the unordered choice is the lowest of all operators. For example, $e_1/e_2 \mid e_3$ is the same as $(e_1/e_2) \mid e_3$.

By the extension, it is expected that a PEGwUC includes both a PEG and a CFG. Obviously, a PEGwUC includes a PEG because a PEGwUC is an extension of a PEG. In addition, informally, we will define a PEGwUC that is equivalent to a CFG as follows: Let $G' = (N, \Sigma, R, S)$ be a CFG. N is a finite set of nonterminals. Σ is a finite set of terminals. R is a finite set of rules. $S \in N$ is a start symbol. We assume without loss of generality that the CFG does not have left recursion since we can eliminate left recursion in a CFG [1]. Then, we can define a PEGwUC G that is equivalent to the CFG G' as a 4-tuple $(N, \Sigma, R', S\,!.)$, where R' is a finite set of rules that is basically the same as the rule R, but differs in the restriction. That is, there exists exactly one expression e, such that $A \leftarrow e \in R'$ for a nonterminal A. In order to satisfy the restriction, we concatenate the expressions of the rules by using unordered choices if more than one rule exists for a nonterminal A. For example, let a CFG G' be a 4-tuple $(\{S\}, \{a, b\}, \{S \leftarrow a, S \leftarrow b\}, S)$. Then, we can define a PEGwUC that is equivalent to the CFG as a 4-tuple $(\{S\}, \{a, b\}, \{S \leftarrow a \mid b\}, S\,!.)$.

2.2 Syntactic Sugar

We consider the any character $.$ expression to be an ordered choice of all single terminals $(a/b/\ldots/c)$ in Σ. 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, one or more repetition, option, and and-predicate are treated as syntax sugars:

$[abc]$	$=$	$a/b/c$	character class
e^+	$=$	ee^*	one or more repetition
$e?$	$=$	e/ε	option
$\&e$	$=$	$!!e$	and-predicate

2.3 Semantics

Medeiros et al. presented a formalization of regular expressions and PEGs, using the framework of natural semantics [10]. In this section, we present a formalization of PEGwUC, based on their work.

First, we define some notations which are used later.

We use $\overset{PEGwUC}{\leadsto}$ to denote a matching relation in PEGwUC. Let $G[e]$ be a PEGwUC whose start expression is replaced with e in G. The matching relation $G[e]x \overset{PEGwUC}{\leadsto} y$ might be read thus: when $G[e]$ parser parses the input string x, the string y remains an unconsumed string. We use `fail` for representing a failure of the matching. That is, $G[e]x \overset{PEGwUC}{\leadsto} \mathtt{fail}$ means that $G[e]$ parser cannot parse the input string x.

As described before, the unordered choice yields some parsing results. Such results are represented by two possible relations over $\overset{PEGwUC}{\leadsto}$. To denote this, we use $x_1 \vee \ldots \vee x_n$. For example, $G[a \mid aa]\,aaa \overset{PEGwUC}{\leadsto} aa \vee a$. We consider that $x_1 \vee \ldots \vee x_n = x_1$ if $n = 1$.

The parsing results have the following relationship:

$$x \vee y \equiv y \vee x \qquad (1)$$

$$\frac{}{G[\varepsilon]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x}\ \textbf{(empty.1)}$$

$$\frac{}{G[a]\ ax \overset{\mathit{PEGwUC}}{\rightsquigarrow} x}\ \textbf{(char.1)}$$

$$\frac{}{G[b]\ ax \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail}}\ \textbf{(char.2)}$$

$$\frac{}{G[a]\ \varepsilon \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail}}\ \textbf{(char.3)}$$

$$\frac{G[R(A)]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} X}{G[A]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} X}\ \textbf{(var.1)}$$

$$\frac{G[e_1]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail}}{G[e_1 e_2]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail}}\ \textbf{(seq.1)}$$

$$\frac{G[e_1]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x_1 \vee ... \vee x_n \qquad G[e_2]\ x_1 \vee ... \vee x_n \overset{\mathit{PEGwUC}}{\rightsquigarrow} X}{G[e_1 e_2]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} X}\ \textbf{(seq.2)}$$

$$\frac{G[e_1]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail} \vee x_1 \vee ... \vee x_n \qquad G[e_2]\ x_1 \vee ... \vee x_n \overset{\mathit{PEGwUC}}{\rightsquigarrow} X}{G[e_1 e_2]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail} \vee X}\ \textbf{(seq.3)}$$

$$\frac{G[e_1]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x_1 \vee ... \vee x_n}{G[e_1 / e_2]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x_1 \vee ... \vee x_n}\ \textbf{(order.1)}$$

$$\frac{G[e_1]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail} \qquad G[e_2]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} X}{G[e_1 / e_2]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} X}\ \textbf{(order.2)}$$

$$\frac{G[e_1]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail} \vee x_1 \vee ... \vee x_n \qquad G[e_2]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} X}{G[e_1 / e_2]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x_1 \vee ... \vee x_n \vee X}\ \textbf{(order.3)}$$

$$\frac{G[e_1]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} X \qquad G[e_2]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} X'}{G[e_1 \mid e_2]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} X \vee X'}\ \textbf{(unorder.1)}$$

$$\frac{G[e]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail}}{G[e^*]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x}\ \textbf{(rep.1)}$$

$$\frac{G[e]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail} \vee x_1 \vee ... \vee x_n \qquad G[e^*]\ x_1 \vee ... \vee x_n \overset{\mathit{PEGwUC}}{\rightsquigarrow} x_1{}' \vee ... \vee x_n{}'}{G[e^*]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x \vee x_1{}' \vee ... \vee x_n{}'}\ \textbf{(rep.2)}$$

$$\frac{G[e]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x_1 \vee ... \vee x_n \qquad G[e^*]\ x_1 \vee ... \vee x_n \overset{\mathit{PEGwUC}}{\rightsquigarrow} x_1{}' \vee ... \vee x_n{}'}{G[e^*]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x_1{}' \vee ... \vee x_n{}'}\ \textbf{(rep.3)}$$

$$\frac{G[e]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail}}{G[!e]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x}\ \textbf{(not.1)}$$

$$\frac{G[e]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail} \vee x_1 \vee ... \vee x_n}{G[!e]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail} \vee x}\ \textbf{(not.2)}$$

$$\frac{G[e]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x_1 \vee ... \vee x_n}{G[!e]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} \mathtt{fail}}\ \textbf{(not.3)}$$

$$\frac{n \geq 2 \qquad G[e]\ x_1 \overset{\mathit{PEGwUC}}{\rightsquigarrow} X \qquad G[e]\ x_2 \vee ... \vee x_n \overset{\mathit{PEGwUC}}{\rightsquigarrow} X'}{G[e]\ x_1 \vee ... \vee x_n \overset{\mathit{PEGwUC}}{\rightsquigarrow} X \vee X'}\ \textbf{(split.1)}$$

Fig. 2 A matching relation $\overset{\mathit{PEGwUC}}{\rightsquigarrow}$ via natural semantics.

$$\mathtt{fail} \vee x \equiv x \vee \mathtt{fail} \tag{2}$$

$$x \vee y \equiv x \qquad (\text{if } x = y) \tag{3}$$

$$\mathtt{fail} \vee \mathtt{fail} \equiv \mathtt{fail} \tag{4}$$

Equations (1) and (2) means that $\vee$ is commutative. For example, $G[a \mid aa]\ aaa \overset{\mathit{PEGwUC}}{\rightsquigarrow} aa \vee a$ is the same with $G[a \mid aa]\ aaa \overset{\mathit{PEGwUC}}{\rightsquigarrow} a \vee aa$. (3) and (4) means that we can eliminate duplication. For example, $G[a \mid a]\ aa \overset{\mathit{PEGwUC}}{\rightsquigarrow} a \vee a$ is the same with $G[a \mid a]\ aa \overset{\mathit{PEGwUC}}{\rightsquigarrow} a$.

For simplicity's sake, we use X as an arbitrary result. That is, X is the same with one of the results: $x_1 \vee \ldots \vee x_n$, $\mathtt{fail}$ or $\mathtt{fail} \vee x_1 \vee \ldots \vee x_n$.

The semantics of PEGwUC shown in **Fig. 2** is similar to that of PEGs. The difference comes from unordered choices and the parsing results. More precisely, we add a new rule **split.1** in order to split some parsing results of the unordered choices. The rules are applied if a PEGwUC has some parsing results, that is, if the input for the PEGwUC parser is $x_1 \vee \ldots \vee x_n$ ($n \geq 2$). When the rule is applied, the rule splits the input and merges the results. For example, $G[e]\ x \vee y$ is split into $G[e]\ x$ and $G[e]\ y$. Let $G[e]\ x \overset{\mathit{PEGwUC}}{\rightsquigarrow} x'$ and $G[e]\ y \overset{\mathit{PEGwUC}}{\rightsquigarrow} y'$. Then, the results are merged into a new result $x' \vee y'$. In addition, we extended the other rules in order to handle some parsing results.

The meanings of the other rules are as follows. **empty.1** says that ϵ matches an empty string. **char.1**, **char.2** and **char.3** say that the expression a attempts to match the prefix of the input string. **var.1** says that the result of the matching of the nonterminal A is a result of the expression $R(A)$. **seq.1**, **seq.2** and **seq.3** say that $G[e_2]$ tries to match by using the result of the matching $G[e_1]\ x$. **order.1**, **order.2** and **order.3** say that an ordered choice attempts to match in the order of the choices and continues to the next matching if the result of the current matching has `fail`. **unorder.1** says that an unordered choice attempts to match both of the alternates of the choice. **rep.1**, **rep.2** and **rep.3** say that a zero-or-more repetition iterates the matching until the matching fails completely. **not.1**, **not.2** and **not.3** say that a not-predicate $!e$ matches empty string if the matching of e succeeds, otherwise the matching of the not-predicate fails. **split.1** says that the rules split the input in order to handle some parsing results and merge the results.

Finally, as with PEGs, left recursion is unavailable in PEGwUC. For example, $A \leftarrow Aa/b$ is unavailable in PEGwUC because it causes a degenerate loop. Thus, we assume that all subsequent PEGwUC do not have left recursion, regardless whether the recursion is direct or indirect.

2.4 Language Properties

In this section, we define a *language* recognized by PEGwUC and discuss the properties of the languages.

A language recognized by a PEGwUC is defined as follows:

Definition 2. *Let $G = (N, \Sigma, R, e_s)$ be a PEGwUC. The language $L(G)$ is the set of strings $x \in \Sigma^*$ for which the start expression e_s matches x.*

As with in Ref. [7], "match" means that the start expression e_s does not fail on the string x, that is, e_s matches x if e_s succeeds on any x-prefix strings. In addition, as with the language definition in PEG, any x-prefix strings are included in $L(G)$ if $x \in L(G)$.

Example 1. *Let $G = (\{\}, \{a\}, \{\}, a)$ be a PEGwUC. Then, the language $L(G) = \{ax \mid x \in \Sigma^*\}$.*

A language $\mathcal{L}$ over a terminal Σ is a *language of a parsing expression with unordered choices* iff there exists a PEGwUC G whose language is L.

Then, we show that PEGwUC have the same properties as PEGs.

Theorem 1. *If $\mathcal{L}$ and $\mathcal{M}$ are the languages of PEGwUC, then $\mathcal{L} \cup \mathcal{M}$ is also a language of PEGwUC.*

Proof Since $\mathcal{L}$ and $\mathcal{M}$ are the languages of PEGwUC, there exists PEGwUC $G_{\mathcal{L}} = (N_{\mathcal{L}}, \Sigma, R_{\mathcal{L}}, e_{S\mathcal{L}})$ and $G_{\mathcal{M}} = (N_{\mathcal{M}}, \Sigma, R_{\mathcal{M}}, e_{S\mathcal{M}})$ whose languages are $\mathcal{L}$ and $\mathcal{M}$. Then, $\mathcal{L} \cup \mathcal{M} = L(e_{S\mathcal{L}} \mid e_{S\mathcal{M}})$. □

Theorem 2. *If $\mathcal{L}$ over Σ is a language of PEGwUC, then $\overline{\mathcal{L}} = \Sigma^* - \mathcal{L}$ is also a language of PEGwUC.*

Proof Since $\mathcal{L}$ is a language of PEGwUC, there exists a

PEGwUC $G_{\mathcal{L}} = (N_{\mathcal{L}}, \Sigma, R_{\mathcal{L}}, e_{S\mathcal{L}})$ whose language is $\mathcal{L}$. Then, $\overline{\mathcal{L}} = L(!e_{S\mathcal{L}})$. □

Theorem 3. *If $\mathcal{L}$ and $\mathcal{M}$ are languages of PEGwUC, then $\mathcal{L} \cap \mathcal{M}$ is also a language of PEGwUC.*

Proof By DeMorgan's laws, $\mathcal{L} \cap \mathcal{M} = \overline{\overline{\mathcal{L}} \cup \overline{\mathcal{M}}}$ □

Theorem 4. *It is undecidable whether the language $L(G)$ of an arbitrary PEGwUC G is empty.*

Proof It is undecidable whether the language of an arbitrary PEG is empty [7]. If it is decidable whether the language L(G) of an arbitrary PEGwUC G is empty, the language of an arbitrary PEG being empty can also be decidable, since GPEGs include PEGs. Hence, it is undecidable. □

Theorem 5. *It is undecidable whether a PEGwUC G_1 and a PEGwUC G_2 are equivalent.*

Proof The equivalence of two arbitrary PEGs is undecidable [7]. Hence, it is also undecidable in a PEGwUC. □

3. Parsing Algorithm

In this section, we describe an algorithm for generating a PEGwUC parser. A PEGwUC parser is an extension of a *packrat parser* used for parsing PEGs. Furthermore, a PEGwUC parser inherits the benefits of a packrat parser in terms of time complexity. That is, a PEGwUC parser runs in a linear time when the PEGwUC does not include unordered choice. A PEGwUC parser consists of functions for parsing a nonterminal and an expression and a main function. In order to define a function for parsing a nonterminal A and an expression e, we write parse_A and parse_e, respectively. We assume that the names of the functions are distinct. **Figure 3** shows the pseudocode.

In Fig. 3, # denotes a comment line and code(e) denotes a placeholder for an expression e. We can replace code(e) with a code for parsing the expression e. The details of code(e) are given in the rest of this section. The function takes an input position i as an argument. In this function, we use three sets Curr, Next, and Temp in order to handle parsing results. Curr and Next are used for storing current and next input positions, respectively. Furthermore, Temp is used for storing input positions temporarily. Basically, elements of a set Next are results of the parsing of a PEGwUC operator. In this parser, we assume that an input string is stored in a variable I and the variable I is an array. I[i] corresponds to x in the semantics shown in Fig. 2. fail is the same in Section 2.3; that is, fail means that a matching fails. In addition, for simplicity, we do not write a code of memoization in each pseudocode. The rest of this section proceeds as follows. To begin with, we describe the algorithm for PEGwUC operators from Section 3.1 to Section 3.7 Finally, we describe a main function of a PEGwUC parser and show some examples in Section 3.8.

3.1 Nonterminals

A code code(A) for parsing a nonterminal A is shown in

```
parse_A(i) # or parse_$e$(i)
  Curr = {i}
  code($e$)
  return Curr
```

Fig. 3 A parse function of a nonterminal $A \leftarrow e$ and an expression e.

Fig. 4. The code corresponds to **(var.1)** shown in Fig. 2. More specifically, parse_A(i) corresponds to $G[R(A)]\ x$. In this code, a PEGwUC parser stores the parsing result of parse_A(i) in the set Next. This means $G[R(A)]\ x \overset{PEGwUC}{\rightsquigarrow} X$ since the set Next is the parsing result of the nonterminal A and the behavior corresponds to **(var.1)**.

3.2 Terminals

A code code(a) for parsing a terminal a is shown in **Fig. 5**. The code corresponds to **(char.1)**, **(char.2)** and **(char.3)** shown in Fig. 2. When I[i] is a, a PEGwUC parser consumes I[i] as with **(char.1)**. In the other cases, the matching fails as with **(char.2)** and **(char.3)**.

3.3 Sequences

A code code($e_1 e_2$) for a sequence $e_1 e_2$ is shown in **Fig. 6**. The code corresponds to **(seq.1)**, **(seq.2)** and **(seq.3)** shown in Fig. 2. When the parsing result of the code code(e_1) is fail, the parsing results of the code code($e_1 e_2$) is also fail as with **(seq.1)**. In the other cases, a PEGwUC parser parses code(e_2) as with **(seq.2)** and **(seq.3)**.

3.4 Ordered Choices

A code code(e_1 / e_2) for an ordered choice is shown in **Fig. 7**. The code corresponds to **(order.1)**, **(order.2)** and **(order.3)** shown in Fig. 2. When the parsing results of parse_e_j(i) do not have fail, a PEGwUC parser finishes parsing the expression e_j at the input position i as with **(order.1)**, since the results are stored in the set Next. The set Next is the parsing result of the ordered choice and it is not used in the parsing. In the other cases, the parser parses the next expression as with **(order.2)** and **(order.3)**.

3.5 Unordered Choices

A code code($e_1 \mid e_2$) for an unordered choice is shown in

```
Next = $\emptyset$
foreach i $\in$ Curr
  if i == fail
    Next = Next $\cup$ {fail}
  else
    Next = Next $\cup$ parse_A(i)
Curr = Next
```

Fig. 4 code(A) : A code for a nonterminal A.

```
Next = $\emptyset$
foreach i $\in$ Curr
  if i == fail || I[i] $\neq a$
    Next = Next $\cup$ {fail}
  else
    Next = Next $\cup$ {i+1}
Curr = Next
```

Fig. 5 code(a) : A code for a character a.

```
if Curr $\neq$ {fail}
  code($e_1$)
if Curr $\neq$ {fail}
  code($e_2$)
```

Fig. 6 code($e_1 e_2$) : A code for a sequence $e_1 e_2$.

```
Next = ∅
Temp = ∅
for j = 1 to 2
  Temp = ∅
  foreach i ∈ Curr
    foreach k ∈ parse_$e_j$(i)
      if k == fail
        Temp = Temp ∪ {i}
      else
        Next = Next ∪ {k}
  Curr = Temp
if Curr == ∅
  Next = {fail}
Curr = Next
```

Fig. 7 code(e_1 / e_2) : A code for an ordered choice e_1 / e_2.

```
Next = ∅
foreach i ∈ Curr
  if i == fail
    Next = Next ∪ {fail}
  else
    Next = Next ∪ parse_$e_1$(i) ∪ parse_$e_2$(i)
Curr = Next
```

Fig. 8 code($e_1 \mid e_2$) : A code for an unordered choice $e_1 \mid e_2$.

```
Next = ∅
Temp = ∅
while Curr ≠ {fail}
  Temp = ∅
  foreach i ∈ Curr
    if parse_$e$(i) == {fail}
      Next = Next ∪ {i}
    Temp = Temp ∪ parse_$e$(i)
  Curr = Temp
Curr = Next
```

Fig. 9 code(e*) : A code for a zero-or-more repetition $e*$.

Fig. 8. The code corresponds to **(unorder.1)** shown in Fig. 2. More specifically, parse_e_j(i) corresponds to $G[e_j]\ x$. This means that a PEGwUC parser attempts every alternative of the unordered choice and unites the results.

3.6 Zero-or-more Repetitions

A code code($e*$) for a zero-or-more repetition $e*$ is shown in **Fig. 9**. The code corresponds to **(rep.1)**, **(rep.2)** and **(rep.3)** shown in Fig. 2. parse_e(i) in the code corresponds to $G[e]\ x$. When the parsing result of parse_e(i) is fail, that is, $G[e]\ x \overset{PEGwUC}{\rightsquigarrow}$ fail, a PEGwUC parser stores the input position i in the set Next. Since the set Next is the parsing result of the repetition $e*$, the element i of the set Next corresponds to x of $G[e*]\ x \overset{PEGwUC}{\rightsquigarrow} x$ and the behavior corresponds to **(rep.1)**. When the result has an element that is not fail, that is, $G[e]\ x \overset{PEGwUC}{\rightsquigarrow}$ fail$\vee x_1 \vee \ldots \vee x_n$ and $G[e]\ x \overset{PEGwUC}{\rightsquigarrow} x_1 \vee \ldots \vee x_n$, the results are stored in the set Temp and the parser continues to parse the repetition $e*$ with the set of the next input positions Temp. The elements of the set Temp corresponds to $x_1 \vee \ldots \vee x_n$ except for fail and the behavior corresponds to **(rep.2)** and **(rep.3)**.

3.7 Not-predicates

A code code($!e$) for a not-predicate $!e$ is shown in **Fig. 10**. The code corresponds to **(not.1)**, **(not.2)** and **(not.3)** shown in Fig. 2. parse_e(i) in the code corresponds to $G[e]\ x$. When

```
Next = ∅
foreach i ∈ Curr
  if {fail} ∈ parse_$e$(i)
    Next = Next ∪ {i}
  if j ∈ parse_$e$(i)
    Next = Next ∪ {fail}
Curr = Next
```

Fig. 10 code($!e$) : A code for a not-predicate $!e$.

```
main()
  read an input string I
  Succ = parse_$e_S$(0)
  if Succ == {fail}
    the matching failed
  else
    the matching succeeded
```

Fig. 11 A main function of a PEGwUC parser.

the parsing result of parse_e(i) is fail, that is, $G[e]\ x \overset{PEGwUC}{\rightsquigarrow}$ fail, a PEGwUC parser stores the input position i in the set Next. Since the set Next is the parsing result of the expression $!e$, the element i of the set Next corresponds to x of $G[!e]\ x \overset{PEGwUC}{\rightsquigarrow} x$ and the behavior corresponds to **(not.1)**. When the parsing result of parse_e(i) has both fail and an input position i, that is, $G[e]\ x \overset{PEGwUC}{\rightsquigarrow}$ fail $\vee x_1 \vee \ldots \vee x_n$, the parser stores both fail and i in the set Next. This means that $G[!e]\ x \overset{PEGwUC}{\rightsquigarrow}$ fail$\vee x$ and the behavior corresponds to **(not.2)**. When the parsing result of parse_e(i) does not have fail, that is, $G[e]\ x \overset{PEGwUC}{\rightsquigarrow} x_1 \vee \ldots \vee x_n$, the parser stores fail in the set Next. This means that $G[!e]\ x \overset{PEGwUC}{\rightsquigarrow}$ fail and the behavior corresponds to **(not.3)**.

3.8 Building a PEGwUC Parser

A main function of a PEGwUC parser is shown in **Fig. 11**. In Fig. 11, Succ denotes a set of input positions that the matching succeeded except for fail.

A procedure for building a PEGwUC parser as follows:
(1) Writing parse functions for each nonterminal in a PEGwUC
(2) Writing a main function
We show two examples of the algorithm.

Example 2. *Let a PEGwUC $G = (\{A\}, \{a\}, \{A \leftarrow a\}, A)$. We first write a parse function* parse_A(i).

```
parse_A(i)
  Curr = {i}
  code($a$)
  return Curr
```

Then, we replace a code code(a) *with a code for parsing a terminal a.*

```
parse_A(i)
  Curr = {i}
  Next = ∅
  foreach i ∈ Curr
    if i == fail || I[i] ≠ $a$
      Next = Next ∪ {fail}
    else
      Next = Next ∪ {i+1}
  Curr = Next
  return Curr
```

```
parse_A(i)
  Curr = {i}
  Next = ∅
  foreach i ∈ Curr
    if i == fail || I[i] ≠ $a$
      Next = Next ∪ {fail}
    else
      Next = Next ∪ {i+1}
  Curr = Next
  return Curr

main()
  read an input string I
  Succ = parse_A(0)
  if Succ == {fail}
    the matching failed
  else
    the matching succeeded
```

Fig. 12 A PEGwUC parser for a PEGwUC $G = (\{A\}, \{a\}, \{A \leftarrow a\}, A)$.

```
parse_A(i) # parse function for $a$ | $aa$
  Curr = {i}
  Next = ∅
  for j = 1 to 2  # $e_1 = a, e_2 = aa$
    foreach i ∈ Curr
      if i == fail
        Next = Next ∪ {fail}
      else
        Next = Next ∪ parse_$e_j$(i)
  Curr = Next
  return Curr

parse_$e_1$(i) # parse function for $a$
  Curr = {i}
  Next = ∅ # code($a$)
  foreach i ∈ Curr
    if i == fail || I[i] ≠ $a$
      Next = Next ∪ {fail}
    else
      Next = Next ∪ {i+1}
  Curr = Next
  return Curr

parse_$e_2$($i$) # parse function for $aa$
  Curr = {i}
  if Curr ≠ {fail} # code($aa$)
      Next = ∅ # code($a$)
      foreach i ∈ Curr
        if i == fail || I[i] ≠ $a$
          Next = Next ∪ {fail}
        else
          Next = Next ∪ {i+1}
      Curr = Next
  if Curr ≠ {fail}  # code($aa$)
      Next = ∅ # code($a$)
       foreach i ∈ Curr
         if i == fail || I[i] ≠ $a$
           Next = Next ∪ {fail}
         else
           Next = Next ∪ {i+1}
       Curr = Next
  return Curr

main()
  read an input string I
  Succ = parse_A(0)
  if Succ == {fail}
    the matching failed
  else
    the matching succeeded
```

Fig. 13 A PEGwUC parser for a PEGwUC $G = (\{A\}, \{a\}, \{A \leftarrow a \mid aa\}, A)$.

Finally, we write a main function. The result are shown in **Fig. 12**. *When* I = *aaa, a result of a parsing is a set* Succ = {1}. *This means that the PEGwUC consumed the prefix a of the input string* I.

Then, we show a more complex example.

Example 3. *Let a PEGwUC* $G = (\{A\}, \{a\}, \{A \leftarrow a \mid aa\}, A)$. *A PEGwUC parser for a PEGwUC G is shown in* **Fig. 13**. *When* I = *aaa, a result of a parsing is a set* Succ = {1, 2}. *This means that the PEGwUC consumed the prefix a and aa of the input string* I.

4. The Complexity of a PEGwUC Parser

In this section, we show the time complexity of a PEGwUC parser. We first show that a PEGwUC parser runs in a linear time if the PEGwUC does not include unordered choice. That is, the size of the set returned by the parse function should be 1. We show this as Lemma 1.

Lemma 1. *Let G be a PEGwUC. If G does not include unordered choice, sizes of the sets returned by each parse function in a PEGwUC parser generated by the algorithm is 1.*

Proof We assume that sizes of the sets returned by each parse functions and code(e) is 1. Then, we check the size of the set Curr is 1 for each code(e).

1. Case code(a)
   Obviously, the size of the set Curr is 1 because if I[i] is a, then Curr = {i + 1}, otherwise Curr = {fail}.
2. Case code($e_1 e_2$)
   By the assumption, the size of the set Curr is 1.
3. Case code(e_1 / e_2)
   By the assumption, the number of iterations of foreach i ∈ Curr and foreach i ∈ parse_e_j(i) is 1. Thus, the size of the set Curr is 1.
4. Case code($e*$)
   By the assumption, the number of iterations of foreach i ∈ Curr is 1. In addition, a set returned by parse_e(i) is also 1. Thus, the size of the set Curr is 1.
5. Case code($!e$)
   This is same with the case code($e*$).

□

Then, we show that the parser runs in a linear time.

Theorem 6. *Let G be a PEGwUC. A parser for the PEGwUC G generated by the algorithm runs in a linear time when the PEGwUC G does not include unordered choice.*

Proof By Lemma 1, sizes of the sets returned by each parse function are 1. Thus, the number of iterations of foreach in each code(e) are also 1. Therefore, we can prove this by induction on the structure of a parsing expression with unordered choices e. □

Next, we show the time complexity in worst-case.

Theorem 7. *Let G be a PEGwUC. A parser for the PEGwUC G generated by the algorithm runs in a cubic time in worst-case.*

Proof Let n be a length of an input string. By the memoization, a number of calls of each parse function is O(n). In each parse function, foreach iterates at most n times since the size of the set Curr is at most n and the function take O(n) to copy the result of the parsing in the iteration. Thus, a PEGwUC parser runs in a cubic time in worst-case. □

Corollary 1. *A PEGwUC parser generated by the algorithm runs in a linear time when the PEGwUC does not include unordered choice and in a cubic time in worst-case.*

5. Experimental Results

In order to check how PEGwUC parsers generated by the algorithm described in Section 3 run in practice, we implemented the algorithm and measured the runtimes. This section reports the experimental results. In this experiment, we use three deterministic grammars and a non-deterministic grammar in a PEGwUC. The deterministic grammars are the grammar of XML, Java, and G_1 such that the G_1 accepts the language $\{a^n b^n c^n \mid n > 0\}$. The non-deterministic grammar G_2 is as shown in **Fig. 14**:

The non-deterministic grammar accepts the language $\{a^n \mid n > 0\}$ and derived from a highly-ambiguous grammar $S \leftarrow SS|SSS|a$ in a CFG [4]. We expect that our PEGwUC parser runs in a cubic-time for this grammar. The parsers used in the experiments are generated by our parser generator based on the algorithm. The parser generator and the grammars are available online at https://github.com/NariyoshiChida/GPEG. All tests in this section are measured on DELL XPS-8700 with 3.4 GHz Intel Core i7-4700, 8 GB of DDR3 RAM, and running on Linux Ubuntu 14.04.3 LTS. We measured runtimes ten times in a row and calculated the averages of the runtimes other than the maximum runtime and the minimum runtime. We have chosen the following files as inputs:

- XML - xmark : a synthetic and scalable XML files that are provided by XMark benchmark program [13].
- Java - relatively large files of 60 KB or larger.
- G_1 - files written as $a^n b^n c^n$ ($n = 10^3, 10^4$ and 10^5).
- G_2 - files written as a^n ($n = 10^2, 10^3$ and 10^4).

The results are shown from **Table 1** to **Table 4**. In these tables, Size, Runtime, and Memo denote a size of an input file, an average of runtimes, and a number of elements in a memoization cache respectively.

```
S ← a T
T ← ( S | S S ) T | ϵ
```

Fig. 14 The nondeterministic grammar G_2.

Table 1 The result of a XML parser.

	Size (byte)	Runtime (sec)	Memo
Input 1	515,109	1.303	1,364,208
Input 2	1,033,034	3.011	2,710,554
Input 3	2,172,640	6.208	5,668,326

Table 2 The result of a Java parser.

	Size (byte)	Runtime (sec)	Memo
Input 1	62,283	0.943	611,713
Input 2	111,293	2.183	1,422,695
Input 3	229,982	5.257	3,203,204

Table 3 The result of a G_1 parser.

	Size (byte)	Runtime (sec)	Memo
Input 1	3,001	0.003	5,015
Input 2	30,001	0.090	50,015
Input 3	300,001	0.987	500,015

Table 4 The result of a G_2 parser.

	Size (byte)	Runtime (sec)	Memo
Input 1	101	0.013	25,452
Input 2	1,001	13.598	2,504,502
Input 3	10,001	15,555.653	250,045,002

By the experimental results, in this case, we can check the algorithm runs in a linear time when a PEGwUC does not include unordered choice and in a cubic time in worst-case.

6. Related Work

Birman and Ullman [2], [3] showed formalism of recognition schemes as TS and gTS. TS and gTS were introduced in Ref. [1] as *Top-Down Parsing Language* (TDPL) and *Generalized Top-Down Parsing Language* (GTDPL) respectively. A PEG is a development of GTDPL. In this paper, we defined a PEGwUC, an extension of a PEG with unordered choices. PEGs are widely used in parser generators. Robert Grimm developed *Rats!*, a PEG-based parser generator for Java [8]. In addition, David Majda developed *PEG.js*, a PEG-based parser generator for JavaScript [6]. Many other PEG-based parser generators were developed [5], [9], [11]. Scott and Johnstone [14], [15] showed GLL parsing, which is an algorithm for generating a generalized parser using LL parsing. GLL parsing is recursive descent-like parsing and handles all CFGs. Furthermore, it runs in a linear time on LL grammars and in a cubic time in worst-case. We showed a PEGwUC and the parsing algorithm. Our parsing algorithm is recursive descent parsing and handles PEGwUC. Furthermore, as with GLL parsing, it runs in a linear time on PEGs and in a cubic time in worst-case.

7. Conclusion

In this study, we formalized a PEGwUC, an extension of a PEG with unordered choices. By the extension, it is expected that a PEGwUC includes both a PEG and a CFG and the extension allows us to write a grammar more intuitively. Furthermore, we showed an algorithm for generating a PEGwUC parser and the implementation. A PEGwUC parser inherits the benefits of a packrat parser in terms of time complexity. That is, a PEGwUC parser runs in a linear time when the PEGwUC does not include unordered choice. In addition, we checked the benefits in our experiments.

References

[1] Aho, A.V. and Ullman, J.D.: *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1972).

[2] Birman, A.: The Tmg Recognition Schema, PhD Thesis, Princeton, NJ, USA (1970).

[3] Birman, A. and Ullman, J.D.: Parsing algorithms with backtrack, *Information and Control*, Vol.23, No.1, pp.1–34 (online), DOI: http://dx.doi.org/10.1016/S0019-9958(73)90851-6 (1973).

[4] Cohen, S.B. and Johnson, M.: The Effect of Non-tightness on Bayesian Estimation of PCFGs, *Proc. ACL* (2013).

[5] Colin, H. and Daniel, F.: Parsing Expression Grammar Template Library, available from ⟨https://code.google.com/p/pegtl⟩ (2014).

[6] David, M.: PEG.js, available from ⟨https://pegjs.org⟩. parser generator for JavaScript.

[7] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, ACM, pp.111–122 (online), DOI: 10.1145/964001.964011 (2004).

[8] Grimm, R.: Better Extensibility Through Modular Syntax, *Proc. 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, New York, NY, USA, ACM, pp.38–51 (online), DOI: 10.1145/1133981.1133987 (2006).

[9] Kuramitsu, K.: Nez: Practical Open Grammar Language, *Proc. 2016*

ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2016, New York, NY, USA, ACM, pp.29–42 (online), DOI: 10.1145/2986012.2986019 (2016).

[10] Medeiros, S., Mascarenhas, F. and Ierusalimschy, R.: From regexes to parsing expression grammars, *Science of Computer Programming*, Vol.93, Part A, pp.3–18 (online), available from ⟨http://dx.doi.org/10.1016/j.scico.2012.11.006⟩ (2014).

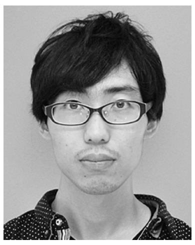
[11] Orlando, H.: Waxeye, available from ⟨http://waxeye.org/⟩.

[12] Redziejowski, R.R.: Some Aspects of Parsing Expression Grammar, *Fundam. Inform.*, Vol.85, No.1-4, pp.441–451 (2008) (online), available from ⟨http://content.iospress.com/articles/fundamenta-informaticae/fi85-1-4-30⟩.

[13] Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I. and Busse, R.: XMark: A Benchmark for XML Data Management, *Proc. 28th International Conference on Very Large Data Bases*, VLDB '02, VLDB Endowment, pp.974–985 (2002) (online), available from ⟨http://dl.acm.org/citation.cfm?id=1287369.1287455⟩.

[14] Scott, E. and Johnstone, A.: GLL Parsing., *Electr. Notes Theor. Comput. Sci.*, Vol.253, No.7, pp.177–189 (2010) (online), available from ⟨http://dblp.uni-trier.de/db/journals/entcs/entcs253.html#ScottJ10⟩.

[15] Scott, E. and Johnstone, A.: GLL Parse-tree Generation, *Sci. Comput. Program.*, Vol.78, No.10, pp.1828–1844 (online), DOI: 10.1016/j.scico.2012.03.005 (2013).



Nariyoshi Chida was born in 1993. He received his B.CS. and M.E. degrees from University of Aizu and Yokohama National University in 2015 and 2017, respectively. He joined NTT secure platform laboratories, Japan. His research interests include formal language and automata theory.

Kimio Kuramitsu is an Associate Professor, leading the Software Assurance research group at Yokohama National University. His research interests range from programming language design, software engineering to data engineering, ubiquitous and dependable computing. He has received the Yamashita Memorial Research Award at IPSJ. His pedagogical achievements include Konoha and Aspen, the first programming exercise environment for novice students. He earned his B.E. at the University of Tokyo, and his Ph.D. at the University of Tokyo under the supervision of Prof. Ken Sakamura.