Parsing Expression Grammars with Unordered Choices

NARIYOSHI CHIDA\textsuperscript{1,†1,a)} KIMIO KURAMITSU\textsuperscript{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\textsuperscript{[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\textsuperscript{[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 \( A \leftarrow aa/aa \). \( A \) is a nonterminal and this is a placeholder for patterns of terminals in common with CFGs. Let \( a^5 \) be an input string. Then, intuitively, we consider the expression consumes \( a^5 \). However, contrary to our intuition, the expression consumes \( a^3 \).

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.
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 \( \rightarrow \) 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 \( G = (N, \Sigma, R, e) \), where \( N \) is a finite set of nonterminals, \( \Sigma \) is a finite set of terminals, \( R \) is a finite set of rules, and \( e \) 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 non-terminal \( A \in N \) 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 \( e \) 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, \( e_1 \| e_2 \). 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.

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

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, \( \Sigma \) 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 \((\{S\}, \{a,b\}, \{\text{fail} \leadsto a, S \leftarrow b, S\})\). Then, we can define a PEGwUC that is equivalent to the CFG as a 4-tuple \((\{S\}, \{a,b\}, \{S \leftarrow \text{fail} \leadsto b\}, S\)).

2.2 Syntactic Sugar

We consider the any character \( . \) expression to be an ordered choice of all single terminals \((a/b/\ldots/c)\) in \( \Sigma \). 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:

\[
\begin{align*}
[abc] & = a/b/c \quad \text{character class} \\
\varepsilon & = \varepsilon^* \quad \text{one or more repetition} \\
& = e/e \quad \text{option} \\
& = !e \quad \text{and-predicate}
\end{align*}
\]

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 \( \leadsto \) 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 \( G[e]x \leadsto y \) might be read thus: when \( G[e] \) parses the input string \( x \), the string \( y \) remains an unconsumed string. We use \( \text{fail} \) for representing a failure of the matching. That is, \( G[e]x \leadsto \text{fail} \) 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 \( \leadsto \). To denote this, we use \( x \lor y \lor x_0 \). For example, \( G[a] \leadsto \text{fail} \lor aa \lor a \). We consider that \( x \lor \ldots \lor x_n = x_1 \) if \( n = 1 \).

The parsing results have the following relationship:

\[
x \lor y \equiv y \lor x
\]
The semantics of PEGwUC shown in Fig. 2 is similar to that of PEGs. The semantics 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 \( x_1 \lor \ldots \lor x_n (n \geq 2) \).

When the rule is applied, the rule splits the input and merges the results. For example, \( G[e] x \lor y \) is split into \( G[e] x \) and \( G[e] y \). Let \( G[e] x \sim_{\text{PEGwUC}} x' \) and \( G[e] y \sim_{\text{PEGwUC}} y' \). Then, the results are merged into a new result \( x' \lor y' \). 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 \( e \) 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 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 \( \neg e \) 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, \( A \sim Aa/b \) 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 \( G = (N, \Sigma, R, e) \) be a PEGwUC. The language \( L(G) \) is the set of strings \( x \in \Sigma^* \) for which the start expression \( e \) matches \( x \).

As with in Ref. [7], “match” means that the start expression \( e \) does not fail on the string \( x \), that is, \( e \) matches \( x \) if \( e \) 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 \( G = (\{1\}, \{a\}, \{1\}, a) \) be a PEGwUC. Then, the language \( L(G) = \{ax \mid x \in \Sigma^*\} \).

A language \( L \) over a terminal \( \Sigma \) is a language of a parsing expression with unordered choices if there exists a PEGwUC \( G \) whose language is \( L \).

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

**Theorem 1.** If \( L \) and \( M \) are the languages of PEGwUC, then \( L \cup M \) is also a language of PEGwUC.

**Proof.** Since \( L \) and \( M \) are the languages of PEGwUC, there exists PEGwUC \( G_L = (N_L, \Sigma, R_L, e_{\Sigma L}) \) and \( G_M = (N_M, \Sigma, R_M, e_{\Sigma M}) \) whose languages are \( L \) and \( M \). Then, \( L \cup M \sim L(e_{\Sigma L} \cup e_{\Sigma M}) \).

**Theorem 2.** If \( L \) over \( \Sigma \) is a language of PEGwUC, then \( L = \Sigma^* \sim L \) is also a language of PEGwUC.

**Proof.** Since \( L \) is a language of PEGwUC, there exists a
3. Nonterminals

A PEGwUC parser and show some examples in Section 3.8. Theorem 3. If \( L \) and \( M \) are languages of PEGwUC, then \( L \cap M \) is also a language of PEGwUC.

Proof By DeMorgan’s laws, \( L \cap M = \overline{\overline{L} \cup \overline{M}} \). \( \Box \)

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. \( \Box \)

Theorem 5. It is undecidable whether a PEGwUC G1 and a PEGwUC G2 are equivalent.

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

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 \( \text{parse}_A(e) \) and \( \text{parse}_e(e) \), respectively. We assume that the names of the functions are distinct. Figure 3 shows the pseudocode.

In Fig. 3, \# denotes a comment line and \( \text{code}(e) \) denotes a placeholder for an expression \( e \). We can replace \( \text{code}(e) \) with a code for parsing the expression \( e \). The details of \( \text{code}(e) \) 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 \( \text{Curr} \), \( \text{Next} \), and \( \text{Temp} \) in order to handle parsing results. \( \text{Curr} \) and \( \text{Next} \) are used for storing current and next input positions, respectively. Furthermore, \( \text{Temp} \) is used for storing input positions temporarily. Basically, elements of a set \( \text{Next} \) 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. \( \text{fail} \) is the same as in Section 2.3; that is, \( \text{fail} \) 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 \( \text{code}(A) \) for parsing a nonterminal \( A \) is shown in Fig. 4. The code corresponds to (var.1) shown in Fig. 2. More specifically, \( \text{parse}_A(i) \) corresponds to \( G[R(A)] x \). In this code, a PEGwUC parser stores the parsing result of \( \text{parse}_A(i) \) in the set \( \text{Next} \). This means \( G[R(A)] x \xrightarrow{\text{PEGwUC}} X \) since the set \( \text{Next} \) is the parsing result of the nonterminal \( A \) and the behavior corresponds to (var.1).

3.2 Terminals

A code \( \text{code}(a) \) 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 \( \text{code}(e_1 e_2) \) 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 \( \text{code}(e_1) \) is \( \text{fail} \), the parsing results of the code \( \text{code}(e_1 e_2) \) is also \( \text{fail} \) as with (seq.1). In the other cases, a PEGwUC parser parses \( \text{code}(e_2) \) as with (seq.2) and (seq.3).

3.4 Ordered Choices

A code \( \text{code}(e_1/ e_2) \) 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 result of \( \text{parse}_e(i) \) do not have \( \text{fail} \), a PEGwUC parser finishes parsing the expression \( e \) at the input position \( i \) as with (order.1), since the results are stored in the set \( \text{Next} \). The set \( \text{Next} \) 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 \( \text{code}(e_1 | e_2) \) for an unordered choice is shown in
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` in the code corresponds to `G[e] x`. When the parsing result of `parse_e` is fail, that is, `G[e] x PEGwUC 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 PEGwUC ~ x` and the behavior corresponds to (not.1). When the parsing result of `parse_e` has both fail and an input position `i`, that is, `G[e] x PEGwUC fail ~ x` and the behavior corresponds to (not.2). When the parsing of `parse_e` does not have fail, that is, `G[e] x PEGwUC ~ x` and the behavior corresponds to (not.3). When the parsing result of `parse_e` is fail, that is, `G[e] x PEGwUC ~ fail` and the behavior corresponds to (not.1).

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 ← a|, A)`. We first write a parse function `parse_A`.

```
parse_A(I)
Cur = (I)
code(a)
return Cur
```

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

Next, we show a more complex example.

**Example 3.** Let a PEGwUC G = ((A), [a], [A ← a | 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 \text{code}(e) is 1. Then, we check the size of the set Curr is 1 for each \text{code}(e).

1. **Case** \text{code}(a)
   - Obviously, the size of the set Curr is 1 because if \(I[i] = a\), then Curr = \{i + 1\}, otherwise Curr = \{fail\}.

2. **Case** \text{code}(e_1 e_2)
   - By the assumption, the size of the set Curr is 1.

3. **Case** \text{code}(e_1/e_2)
   - By the assumption, the number of iterations of foreach \(i \in \text{curr}\) and foreach \(i \in \text{parse}_{e_j}(i)\) is 1. Thus, the size of the set Curr is 1.

4. **Case** \text{code}(e^*)
   - By the assumption, the number of iterations of foreach \(i \in \text{curr}\) 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** \text{code}(!e)
   - This is same with the case \text{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 \text{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 $S \leftarrow S S S S S \alpha$ 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.

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


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.