**Regular Paper**

# Packrat Parsers Can Support Multiple Left-recursive Calls at the Same Position

Masaki Umeda[1,a) ] Atusi Maeda[2]

Received: June 29, 2020, Accepted: September 22, 2020

***Abstract:*** One of the common problems with the recursive descent parsing method is that when parsing with a left-recursive grammar, the parsing does not terminate because the same parsing function is recursively invoked indefinitely without consuming the input string. Packrat parsing, which is a variant of recursive descent parsing method that handles grammars described in parsing expression grammars (PEGs) by backtracking, is also affected by the above problem. Although naive backtracking parsers may exhibit an exponential execution time, packrat parsers achieve a linear time complexity (for grammars that are not left-recursive) by memoizing the result of each call to the parsing functions. Some methods have been proposed to solve the problem of left recursion in packrat parsers. In these methods, memoization tables in packrat parsers are modified to limit the depth of the recursive calls. By calling the same parsing function repeatedly while increasing the limit, the parsed range in the input string is expanded gradually. These methods have problems in that multiple occurences of left-recursive calls at the same input position cannot be handled correctly, and some of the grammars that does not include left recursion cannot be handled. In this research, we propose and implement a new packrat parser to address these problems. This packrat parser can handle multiple occurences of left-recursive calls at the same position in the input by giving priority to the most recently used rule when gradually increasing the parsed range of the recursion. In the evaluation of the proposed method, in addition to the grammars including left recursion manageable by the methods proposed in existing studies, we confirmed that our approach supports the grammars that cannot be handled by those existing methods.

***Keywords:*** PEG, packrat parser, left recursion, memoization

## 1. Introduction

### 1.1 Background

#### 1.1.1 Parsing Expression Grammars

Parsing expression grammars (PEGs) [2] are formal grammars which, along with context free grammars (CFGs) and regular expressions, consists of a set of rules that describe the procedure for parsing strings in a top-down manner. PEGs adopt ordered choices, and thus there is no ambiguity from choices, whereas CFGs have no priority in the analysis of choices, and therefore there can be an ambiguity that a sentence have multiple analysis results for the same sentence. In addition, PEGs can describe a range of grammar classes that cannot be represented by CFGs, because of their unlimited lookahead [2].

A PEG rule is written using parsing expression $e$ and nonterminals `N` as follows.

`<N>` $\leftarrow e$

And the set of parsing expressions is defined recursively by applying the following rules to $\varepsilon$ (the empty string) and the elements of the set of terminals $T$ and nonterminals $NT$.

- The empty string $\varepsilon$ is a parsing expression.
- A terminal $t$ ($t \in T$) is a parsing expression.
- The constant $\cdot$, which stands for any terminal symbol, is a parsing expression.
- A nonterminal `N` (`N` $\in NT$) is a parsing expression.
- A concatenation $e_1 e_2$ of two parsing expressions $e_1, e_2$ is a parsing expression.
- An ordered choice $e_1/e_2$ of two parsing expressions $e_1, e_2$ is a parsing expression.
- A zero-or-more repetitions $e*$ of a parsing expression $e$ is a parsing expression.
- A one-or-more repetitions $e+$ of a parsing expression $e$ is a parsing expression.
- An affirmative lookahead $\&e$ of a parsing expression $e$ is a parsing expression.
- A negative lookahead $\&e$ of a parsing expression $e$ is a parsing expression.
- An optional match $e?$ of a parsing experssion $e$ is a parsing expression.

In this paper, we write a string of terminals as `‘str’`, and a nonterminal in a grammar definition as `<NT>`.

The following is an example of a grammar described in a PEG, which represents a non-context-free language.

```
<S> ← &(<A>!‘b’)‘a’ + <B>! ·
<A> ← ‘a’<A>?‘b’
<B> ← ‘b’<B>?‘c’
```

---

[1] Graduate School of Science and Technology Degree Programs in Systems and Information Engineering Master’s Program in Computer Science, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[2] Department of Information Engineering, Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[a)] umeda@ialab.cs.tsukuba.ac.jp

This grammar matches $\{a^n b^n c^n \mid n \geq 1\}$, a length-$n$ sequence of `'a'` followed by sequences of `'b'` and `'c'` of the same length, in that order. That is, it matches for `'aaabbbccc'` but not for `'abbcc'`.

In the above rules, zero-or-more repetitions, one-or-more repetitions, affirmative lookahead, and optional match parsing expressions are syntactic sugars, each of which can be rewritten with a nonterminal symbol N and is defined as follows.

**$e*$: zero-or-more repetitions of $e$**

$$N \leftarrow eN/\varepsilon$$

**$e+$: one-or-more repetitions of $e$**

$$N \leftarrow eN/e$$

**$\&e$: affirmative lookahead of $e$**

$$N \leftarrow !!e$$

**$e?$: optional match of $e$**

$$N \leftarrow e/\varepsilon$$

### 1.1.2 Packrat Parsing

The parsing directed by a PEG can be performed using a recursive descent parsing with backtracking. However, backtracking may cause the same input and rule combination to be analyzed multiple times, leading to an exponential time-complexity, which is unacceptable in practice. To solve this problem, a technique called memoization is used [1].

In memoization, the arguments and results of function calls are recorded. If the arguments of a subsequent call are in the table, the stored value can be returned. In the case of recursive descent parsing, each rule and parsing position in the input are considered as arguments. When the same rules and parsing positions are used for parsing, the recorded parsing results are reused.

Packrat parsing achieves a linear time complexity (for 'well-formed' grammars [2], which excludes left recursion) through memoization, even though it allows backtracking [1]. A parser implementation using pacrkat parsing is called a packrat parser.

### 1.1.3 Left Recursion in PEG

Consider the following PEG.

```
<Expr> ← <Expr>'+'<Term>/<Expr>'-'<Term>/<Term>
```

Here, if `Term` matches an unsigned integer such as `'123'` or `'0'`, what is parsed by `Expr` is either an integer arithmetic expression consists of addition and subtraction such as `'12+361'` or `'1000-700+73'`, or a single integer such as `'373'`. In this example, the first and second alternatives of the `Expr` are prefixed by the `Expr` itself. The structure in which the nonterminal on the left side of the rule appears to the leftmost of any of the alternatives for each of the ordered choices on the right side is called (direct) left recursion.

Recursive descent parsers parse an input string by calling the function corresponding to each nonterminal symbol recursively. Therefore, when a naive recursive descent parser encounters a rule that involves left recursion, the parsing generally does not terminate because it repeats recursive calls infinitely without consuming the input characters.

In this paper, we call a function call corresponding to a nonterminal symbol (e.g., `Expr` in the example above) causing left recursion a *left-recursive call*.

In general, it is possible to convert a left-recursive grammar to a non-left-recursive grammar that accepts the same input string, although the shape of the resulting syntax tree changes. As an example, the input string `'1-2-3'` will be parsed as `'(1-2)-3'` prior to conversion, whereas after conversion it is parsed as `'1-(2-3)'`. This means that the conversiion changes the calculation result when the parsed result is interpreted as an arithmetic expression.

### 1.1.4 Handling of Left Recursion in Previous Studies

In Warth et al. [6] and Goto et al. [3], the authors modified the memoization table of the packrat parser to limit the number of functions called at a time and prevent an infinite recursion. Then, by repeating the parsing process over same rules, they have succeeded in parsing the grammars including left recursion. However, Warth et al.'s method [6] has a problem in that it is not possible to analyze multiple left-recursive calls that occur at the same position in the input string. For example, if we give an input string `'baab'` to the grammar

```
<S> ← <A>'b'/'b'
<A> ← <A>'a'/<S>'a'
```

(where the start symbol is S), a left-recursive invocation of the invocation path <S> → <A> → <A> and another left- recursive invocation of the path <S> → <A> → <S> will occur at the first input character position 0, and the Warth et al.'s method [6] will terminate abonormally. Goto et al.'s method [3], which addressed the problem of Warth et al.'s method, also has their own problems. Given a grammar

```
<S> ← <A>'-'<A>
<A> ← <B>'b'/'b'
<B> ← <B>'a'/<A>'a'
```

(where the start symbol is S), which causes multiple left-recursive calls at the same location plus multiple left-recursive calls at multiple input character positions, their method does not behave as the authors claimed (e.g., parsing of the string `'baab-baab'` fails). In addition, Goto et al.'s method [3] cannot handle the following grammars, which was handled by Warth et al.'s method. In particular, Goto et al's method:

- fails to parse input `'1+1'` with the grammar

  ```
  <Exp> ← '1''+'<Exp>/'1',
  ```

  which does not include any left-recursive calls,

- fails to parse non-repetitive input `'b'` with the grammar

  ```
  <S> ← <A>'b'/'b'
  <A> ← <A>'a'/<S>'a',
  ```

- and fails to parse input `'aaa'` with the grammar

  ```
  <S> ← <S>'a'/'a',
  ```

  which includes only direct left recursion.

### 1.2 Purpose of this Research

As mentioned in Section 1.1.3, it is possible to eliminate left recursion in a grammar described in a PEG by transforming the grammar into a different version with a different form of a syntax tree. However, in general, a change in the shape of the syntax tree may change the meaning of the parsed results, which is undesirable. In this study, we propose a novel algorithm that can handle PEGs that cannot be handled in previous methods, as well as those that have successfully been handled in previous approaches, increasing the usability and applicability of packrat parsing.

### 1.3 Structure of the Paper

In Section 2, we present the basic implementation of our packrat parser using a pseudo code. In Section 3, we extend the packrat parser to allow it to handle grammar rules with left recursion, according to the algorithm proposed in a previous study, and describe its problems. In the following Section 4, we describe our method to solve these problems. Section 5 presents an evaluation of the proposed method, and finally, Section 6 provides some concluding remarks as well as future issues and prospects.

## 2. The Basic Packrat Parser

In this section, we describe the general structure and behavior of the packrat parser used throuout this paper. The parser described in this paper is assumed to be a recognizer that takes a PEG and a string as input and judges whether the whole input string matches the grammar without generating a syntax tree.

### 2.1 Structure of the Packrat Parser

Our basic version of packrat parser is defined using the mutually recursive functions `eval` (**Fig. 1**) and `applyRule` (**Fig. 2**).

The results of the parse are expressed in `Ast` type, which has two values, MATCH, and MISMATCH.

In addition, the following global variables are defined.

**POS**: This indicates the current parsing position of the input string, initialized to zero.

**MEMO**: This is a table indexed by the positions and nonterminals defined in the grammar, and stores a pair of `ans`, which is a value of type `Ast` and is the parsed result, and `pos_next`, which is of a **Position** type and indicates the start position of the next parse. All parse results are initialized to NULL.

#### 2.1.1 The eval Function

The `eval` function takes a parsing expression as an argument. It decomposes the given parsing expression and tries to parse the input string. If each decomposed element is a string of terminals, it is checked against the input characters, and if it is a nonterminal, the `applyRule` function is applied to the symbol.

The pseudo code for the `eval` function is shown in Fig. 1. The parsing expression given is assumed to be desugared and replaced by an alternative representation.

In our implementation, ordered choices and concatenations are processed from left to right, in depth-first order.

#### 2.1.2 The applyRule Function

The `applyRule` function takes a nonterminal symbol and a position as arguments. It parses the input string using the parsing expression which is associated with the given nonterminal symbol. It also maintains the memo table MEMO and reuses the previously recorded result, if the result for the arguments can be found in the table.

```
Ast eval(ParsingExpression $e$){
  p = POS
  if($e$ is the empty string $\varepsilon$)
    return MATCH

  if($e$ is a terminal str){
    if(the input at POS starts with str){
      POS = POS + (the length of str)
      return MATCH
    }else
      return MISMATCH
  }

  if($e$ is a negative lookahead $!e_{in}$){
    if(eval($e_{in}$) == MATCH){
      POS = p
      return MISMATCH
    }else
      return MATCH
  }

  if($e$ is a nonterminal $N$)
    return applyRule(N, POS)

  if($e$ is an ordered choice $e_1/e_2$){
    if(eval($e_1$) == MATCH)
      return MATCH
    else
      return eval($e_2$)
  }

  if($e$ is a concatenation $e_1e_2$){
    if(eval($e_1$) == MATCH){
      if(eval($e_2$) == MISMATCH){
        POS = p
        return MISMATCH
      }else
        return MATCH
    }else
      return MISMATCH
  }
}
```

**Fig. 1** The `eval` function.

```
Ast applyRule(Nonterminal N, p){
  if(MEMO(N, p) == NULL){
    exp = getRuleDef(N)
    ans = eval(exp)
    MEMO(N, p) = {ans, POS}
    return ans
  }else{
    POS = MEMO(N, p).pos_next
    return MEMO(N, p).ans
  }
}
```

**Fig. 2** `applyRule`.

The pseudo code for the `applyRule` function is shown in Fig. 2. We introduce an auxilary function `getRuleDef`, which takes a nonterminal `N` as an argument and returns the parsing expression associated with `N`.

### 2.2 How the Packrat Parser Works

The parsing of the input string is initiated by calling the `applyRule` function on the start symbol, which is a nonterminal in a given PEG. The parsing terminates when the initial call to the `applyRule` function returns, and the parse is successful if the return value is MATCH and the `POS` at the end of the analysis is the same as the length of the input string.

This packrat parser cannot handle left recursion. When a grammar with left recursion is given, the parser terminates abnormally due to a stack overflow of the function call, caused by infinite recursive calls.

## 3. Goto et al.'s Packrat Parser

In this section, we describe the packrat parser proposed in Goto et al. [3] that can parse constructs containing left recursion.

Let us reconsider the example grammar involving left recursion in Section 1.

`<Expr>` ← `<Expr>'+'<Term>/<Expr>'-'<Term>/<Term>`

In general, we can substitute a nonterminal symbol in any parsing expression for the parsing expression of the right-hand side of the symbol. Repeating this substitution may result in left-recursive rules (similar to the one above) we previously defined. We call such derived case *indirect* left recursion. By contrast, as with `Expr` in the example above, a single rule involving left-recursive calls is called direct left recursion. Left-recursive rules are widely used in various types of grammars, for example, when defining left-associative operators. In this paper, following the terminology of previous studies [3], [6], we call the first rule of a cyclic path in call graph that is created by left-recursive call the "head rule" and the other rules in the cycle "involved rules." In the case of direct left recursion there are no involved rules, but only a head rule for each cycle.

### 3.1 Resolving Left Recursion with Seeds

Consider the following grammar.

`<A>` ← `<A>'+n'/'n'`

The strings that can be parsed by this grammar will be `'n'`, `'n+n'`, and so on; however, given the grammar, the packrat parser in Section 2 recurs infinitely and the parsing never terminates. Thus, we rewrite this grammar to the following with a limit on the number of left-recursive calls.

$<\mathtt{A}_0>$ ← MisMatch

$<\mathtt{A}_n>$ ← $<\mathtt{A}_{n-1}>$`'+n'/'n'` $(n \geq 1)$

Suppose we parse `'n+n+n'` using this grammar. The range to be parsed and the results are shown in **Table 1**. First, we use $\mathtt{A}_0$ to match with $\mathtt{A}_1$. Since $\mathtt{A}_0$ always fails, the first alternative of $\mathtt{A}_1$ fails and the next alternative `'n'` matches the first `'n'` in the input string. Therefore, $\mathtt{A}_1$ as a whole matches `'n'`. We then use $\mathtt{A}_1$ to match with $\mathtt{A}_2$. Since $\mathtt{A}_1$ matches `'n'` in the first alternative, we consider whether the next `'+n'` matches. In addition, since the rest of the input string is `'+n+n'`, the entire $\mathtt{A}_2$ matches `'n+n'`, leaving `'+n'`. Similarly, let us try to match $\mathtt{A}_3$ using $\mathtt{A}_2$. Since $\mathtt{A}_2$ matches `'n+n'`, the next `'+n'` matches the remaining `'+n'` of the input string, and the entire input string `'n+n+n'` matches $\mathtt{A}_3$ and the parse terminates successfully.

Table 1 Matching A with bounded left recursion.

| input | n | + | n | + | n | | result |
|---|---|---|---|---|---|---|---|
| $\mathtt{A}_0$ | × | | | | | | MisMatch |
| $\mathtt{A}_1$ | 'n' | | | | | | Match |
| $\mathtt{A}_2$ | $\mathtt{A}_1$ | '+n' | | | | | Match |
| $\mathtt{A}_3$ | $\mathtt{A}_2$ | | | '+n' | | | Match |
| $\mathtt{A}_4$ | 'n' | | | | | | Match |

Now, we consider how to proceed further. That is, matching $\mathtt{A}_4$ using $\mathtt{A}_3$. Since $\mathtt{A}_3$ matches `'n+n+n'`, which is the entire input string, the next `'+n'` match fails because no further input to be matched remains. Since the first alternative fails, the parser backtracks to the beginning of the input string, and tries the next alternative. Since the second alternative is `'n'`, it matches the first `'n'` of the input string, leaving `'+n+n'` as unprocessed. Therefore, $\mathtt{A}_4$ as a whole matches `'n'`.

Thus, if we consider the match of $\mathtt{A}_n$ using $\mathtt{A}_{n-1}$ as a single parsing step, the shortest string is matched in the first step and the matched range increases with each subsequent step. When the match is already reached to the longest possible range, the result of the next parsing step returns to the first shortest value. We call the result of the first match, $\mathtt{A}_1$, a "seed." Based on this observation, the authors of Refs. [6] and [3] each proposed a version of packrat parsers that can handle left recursion, although with some limitations. Warth et al.'s method [6] does not take into account the case in which multiple call cycles (and thus multiple head rules) occur at the same location, and it terminates abnormally when analyzing such left-recursive grammars. In the following, we describe Goto et al.'s method [3], which takes into account the case in which there are multiple head rules.

### 3.2 Goto et al.'s Method

The packrat parser proposed by Goto et al. [3] uses the same `eval` function applied in the basic packrat parser but modifies the `applyRule` function. In addition, the updateMemo and `glowLR` functions are used.

In addithion, the following variables are newly defined.

**MAXPOS:** This is a variable of **Position** type, which records the maximum value of the parse position. As the parsing progresses, if the `POS` is updated to a value larger than the current MAXPOS, MAXPOS is updated.

**CALL:** This is a call stack recording rules already applied by the `applyRule` function.

**GROWSTATE:** This is a boolean variable that indicates whether the analysis is in the iteration mode (i.e. during a call to the `glowLR` function).

In addition to Match and MisMatch, Goto et al. [3] include Fail as a value of `Ast` type, which is treated like MisMatch by the `eval` and other functions. This corresponds to $\mathtt{A}_0$ in Section 3.1.

#### 3.2.1 Detection of Left-Recursive Calls

When the basic packrat parser tries to math nonterminals using the `applyRule` function, if the previous result is not found in the MEMO table, then the `eval` function is called, and the result is recorded in the MEMO table.

```
if(MEMO(N, p) == NULL){
  exp = getRuleDef(N)
  ans = eval(exp)
  MEMO(N, p) = {ans, POS}
}
```

We change the code to pre-write a special value Fail in the MEMO table before calling the `eval` function.

```
if(MEMO(N, p) == Null)
  MEMO(N, p) = {Fail, POS}
  exp = getRuleDef(N)
  ans = eval(exp)
  MEMO(N, p).ans = ans
  MEMO(N, p).pos_next = POS
```

This allows us to detect the occurrence of a left-recursive call at the second call of the applyRule function.

#### 3.2.2 The Updated applyRule Function

As the main change, the rule defined by the nonterminal N is applied by calling the updateMemo function. The new applyRule function also takes a nonterminal N and a **Position** as arguments and returns Ast. The modified applyRule function is shown in **Fig. 3** [*1].

#### 3.2.3 The updateMemo Function

The updateMemo function applies a bookkeeping task for MEMO tables, which was achieved using the applyRule function in the basic version. When matching nonterminals without left recursion, the content of MEMO(N, p) is returned, whereas when left recursion occurs, the content of MEMO(N, p) is used to parse the input again. In addition, the matched result is returned after updating MEMO. The pseude code of the updateMemo function is shown in **Fig. 4**.

By keeping the nonterminal symbol that started the parse pushed onto the CALL stack, we can detect the parse with the same rule and prevent a single parse from becoming too deep. When parsing a construct with left recursion, the result returned by the eval function called at line 7 is used as a seed. As shown in Section 3.1, it is possible to parse an input with a left-recursive grammar by repeatedly applying head rule, starting from a seed. In Goto et al.'s method [3], however, any left-recursive call in a grammar is processed by repeatedly calling the start symbol instead of the head rule. This iterative parsing process is performed regardless of whether the start symbol is (directly or indirectly) left-recursive. The glowLR function shown in Section 3.2.4 is responsible for one step of the iteration, and the process proceeds by repeatedly calling this function.

```
Ast applyRule(Nonterminal N, p){
  m = MEMO(N, p)
  if(m != Null && N∈CALL){
    POS = m.pos_next
    return m.ans
  }else
    return updateMemo(N, p)
}
```

**Fig. 3** applyRule used in Goto et al.'s method.

```
1   Ast updateMemo(Nonterminal N, p){
2     CALL.push(N)
3     if(MEMO(N, p) == Null)
4       MEMO(N, p) = {Fail, p}
5     m = MEMO(N, p)
6     exp = getRuleDef(N)
7     ans = eval(exp)
8     if(ans == Match)
9       m.pos_next = POS
10    if(MAXPOS < m.pos_next)
11      MAXPOS = m.pos_next
12    m.ans = ans
13    CALL.pop()
14    if(GROWSTATE == False && N == start symbol){
15      while(True){
16        ans = growLR(N, p)
17        if(ans == Fail || POS ≥ length of the input )
18          break
19      }
20    }
21    return ans
22  }
```

**Fig. 4** updateMemo in Goto et al.'s method.

[*1] The pseudo-code in Ref. [3] is inconsistent with the explanation, and we rewrite the code according to the explanation in the paper because otherwise the parse will terminate abnormally.

#### 3.2.4 The growLR Function

The glowLR function takes a nonterminal symbol and a position and returns Ast. The function repeats the iteration based on a seed obtained by the call to the eval in the updateMemo function. The pseudo code for glowLR is shown in **Fig. 5**.

### 3.3 Example of Successful Analysis

As an example of a PEG with left recursion that can be parsed using the Goto et al.'s method [3], we show a grammar for arithmetic expressions that contain multiple additions.

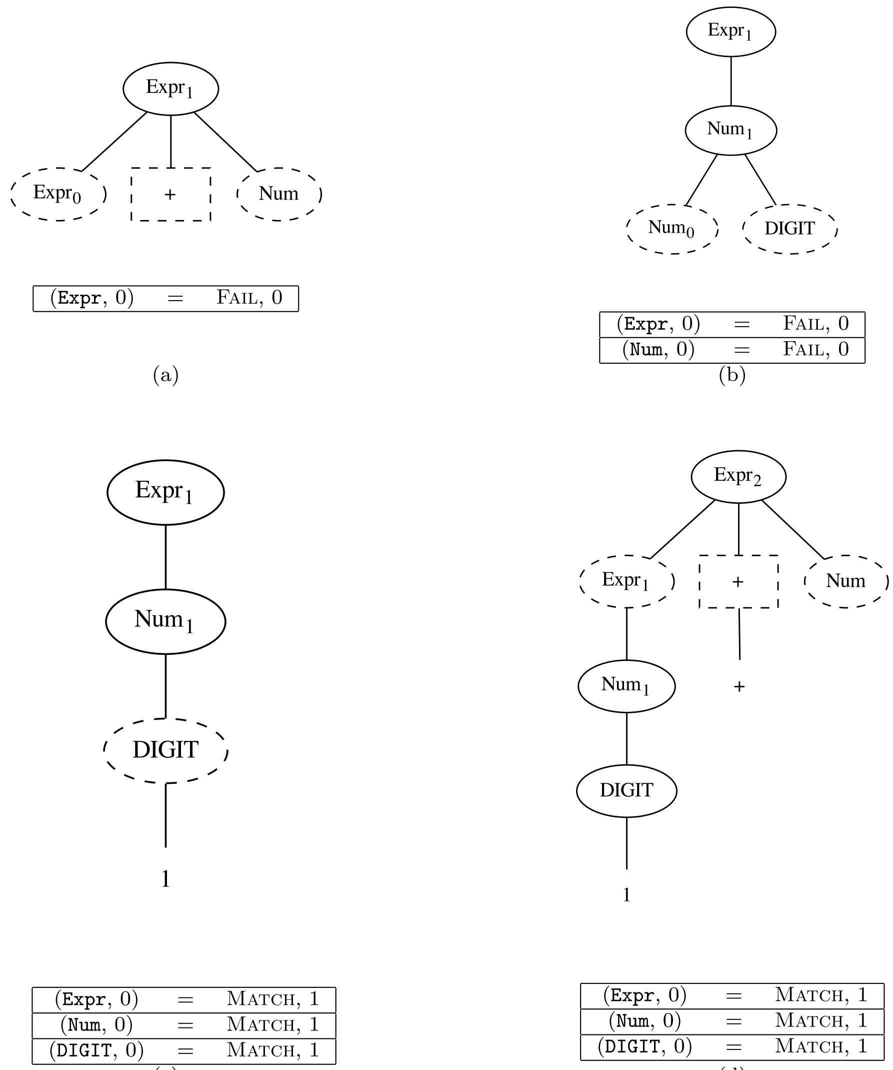
$$
\begin{aligned}
\texttt{<Expr>} &\leftarrow \texttt{<Expr>'+'<Num>/<Num>}\\
\texttt{<Num>} &\leftarrow \texttt{<Num><DIGIT>/<DIGIT>}\\
\texttt{<DIGIT>} &\leftarrow \texttt{'0'/'1'/'2'/'3'/'4'/'5'/'6'/'7'/'8'/'9'}
\end{aligned}
$$

The input string to be analyzed is '12+3'. Because the start symbol is Expr, parsing starts with a call to applyRule(Expr, 0). Immediately after the parsing process starts, nothing is recorded in MEMO, and thus the updateMemo function is called, and the entire parsing process proceeds mainly within this function. An example of imaginary syntax trees and MEMO table states during the execution of the method are shown in **Figs. 6** and **7**. Here, the nodes representing the portion of the tree currently being matched are shown in dashed lines.

First, the parser attempts to parse the first alternative of Expr, <Expr>'+'<Num>, as shown in Fig. 6 (a). Then, because the MEMO table contains Fail, the parsing fails and the parser tries the next alternative. The second alternative is <Num>; because nothing is recorded in the MEMO table, we start matching with Num and try to match with its first alternative, <Num><DIGIT>. At this time, as shown in Fig. 6 (b), this alternative fails because Fail is recorded in the MEMO table, and we move on to the next alternative. Here, the analysis with the nonterminal DIGIT succeeds

```
Ast growLR(NonTerminal N, p){
  CALL.push(N)
  GROWSTATE = True
  old_pos = MAXPOS
  m = MEMO(N, p)
  POS = p
  exp = getRuleDef(N)
  ans = eval(exp)
  if(ans == Match && POS > m.pos_next){
    m.ans = ans
    m.pos_next = POS
  }
  GROWSTATE = False
  CALL.pop()
  if(old_pos != MAXPOS)
    return ans
  else
    return Fail
}
```

**Fig. 5** glowLR in Goto et al.'s method.



| | | |
|---|---|---|
| (Expr, 0) | = | Fail, 0 |

(a)

| | | |
|---|---|---|
| (Expr, 0) | = | Fail, 0 |
| (Num, 0) | = | Fail, 0 |

(b)

| | | |
|---|---|---|
| (Expr, 0) | = | Match, 1 |
| (Num, 0) | = | Match, 1 |
| (DIGIT, 0) | = | Match, 1 |

(c)

| | | |
|---|---|---|
| (Expr, 0) | = | Match, 1 |
| (Num, 0) | = | Match, 1 |
| (DIGIT, 0) | = | Match, 1 |

(d)

**Fig. 6** Successful parse example using Goto et al.'s method #1.

and matches to `'1'`, and the first result of the nonterminal `Expr` becomes Match, the result of which is recorded in the MEMO table, as shown in Fig. 6 (c). This result becomes the seed, which we write as Expr$_1$. Next, the match process moves on to the repetition in the `glowLR` function.

In the parsing using the `glowLR` function, <Expr>'+' <Num>/<Num> is parsed again with Expr$_1$ being recorded in the MEMO table. Rewriting the PEG with the explicit recursion count, we obtain <Expr$_2$> $\leftarrow$ <Expr$_1$>'+'<Num>/<Num$_2$>. As shown in Fig. 6 (d), we start matching the first alternative; however, it fails because the second character in the input string is `'2'` and does not match `'+'`. The parser tries the next alternative and starts matching with `Num`. The result is as shown in Fig. 7 (e), and the first `'12'` of the input string is recognized by the parser as the result of the parsing of the entire expression Expr$_2$.

The second parsing with the `glowLR` function attempts to match Expr$_3$ using Expr$_2$ recorded in the MEMO table. As shown in the figure, the parsing of the first alternative proceeds to match between the nonterminal `Num` and `'3'` in the input string. Because the MEMO table contains Fail, the first alternative fails and the parser tries the next alternative `DIGIT`. As shown in Fig. 7 (g), `DIGIT` and `'3'` match, and the entire `Expr` and `'12+3'` match.

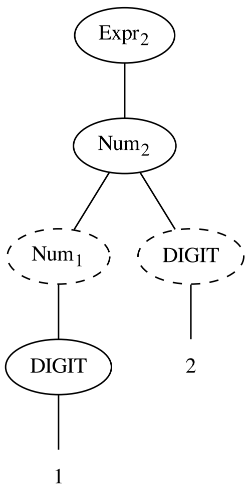
Now, before the third parsing by the `glowLR` function begins, the `POS` matches the length of the input string. Therefore, the complete result is returned as `Match` and the parsing is finished.

## 3.4 Example of Analysis Failure

Goto et al.'s method [3] can continue to parse multiple left-recursive calls at the same position without errors. However, because the repetition is limited to the start symbol, their method cannot parse correctly when left recursion on another nonterminal symbol should be repeated.

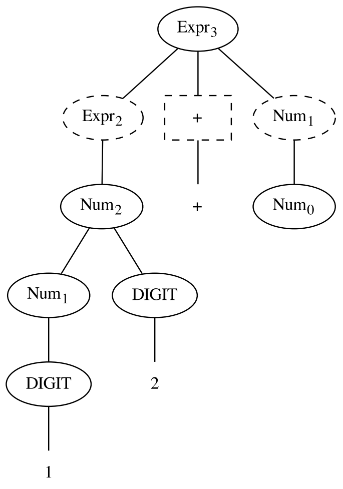
We show an example of parsing the input string `'12+34'` using the same grammar as in the successful example. **Figure 8** shows the tree representing the parsing process and the MEMO table. We skip up to the parsing of `'12+3'` because it is the same as the case of a success, and start with the parsing by Expr$_4$.

Figure 8 (a) shows the third parsing of `Expr` using the `glowLR` function. Because the first alternative `Expr3` matches the input string `'12+3'` and the following `'+'` does not match the input string `'4'`, we move to parsing by the second alternative, as shown in Fig. 8 (b). Because the parsing expression of the second alternative is `Num`, we parse the input by Num$_3$ using Num$_2$ in



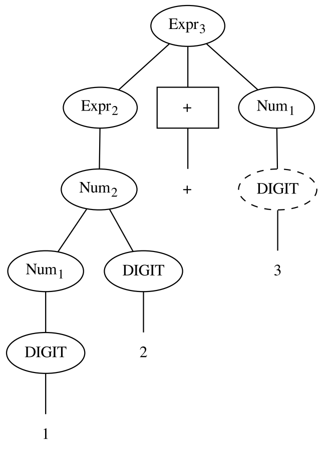
| | | |
|---|---|---|
| (`Expr`, 0) | = | Match, 2 |
| (`Num`, 0) | = | Match, 2 |
| (`DIGIT`, 0) | = | Match, 1 |
| (`DIGIT`, 1) | = | Match, 2 |

(e)



| | | |
|---|---|---|
| (`Expr`, 0) | = | Match, 2 |
| (`Num`, 0) | = | Match, 2 |
| (`DIGIT`, 0) | = | Match, 1 |
| (`DIGIT`, 1) | = | Match, 2 |
| (`Num`, 3) | = | Fail, 3 |

(f)



| | | | | | |
|---|---|---|---|---|---|
| (`Expr`, 0) | = | Match, 4 | (`DIGIT`, 1) | = | Match, 2 |
| (`Num`, 0) | = | Match, 2 | (`Num`, 3) | = | Match, 4 |
| (`DIGIT`, 0) | = | Match, 1 | (`DIGIT`, 3) | = | Match, 4 |

(g)

**Fig. 7** Successful parse example using Goto et al.'s method #2.

the MEMO table. The first alteranative `DIGIT` of `Num`$_3$ does not match `'+'` of the input string, so the parsing by `Num`$_3$ fails. Finally, we try to parse the second alternative, `DIGIT`, which is the state in Fig. 8 (c). Because only a `'1'` of the input string is parsed by `DIGIT`, this is the result of `Expr`; because MAXPOS is not updated any further, the execution is terminated and only `'1'` is the final parsed result of `Expr`, leaving the rest of the input string, `'2+34'`, unparsed.

In addition, Goto et al.'s method [3] has the following problems, as described in Section 1.1.4:

- It fails to analyze the input that does not require repetition.
- It fails to handle direct left recursion.

## 4. Proposed Method

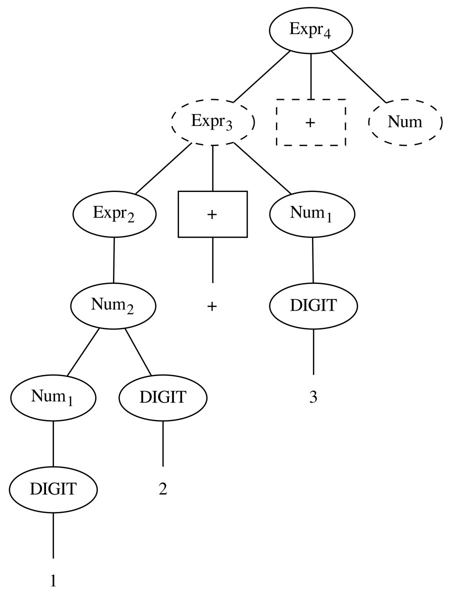
In this section, we propose a parser that can handle multiple left-recursive calls occurring at the same position in the input string and can handle multiple left-recursive calls occurring at a different position [*2].

Whereas Goto et al.'s method repeats on the start symbol, the proposed method repeats on the most recently detected left-recursive head rule.

### 4.1 The sturcture of Proposed Packrat Parser

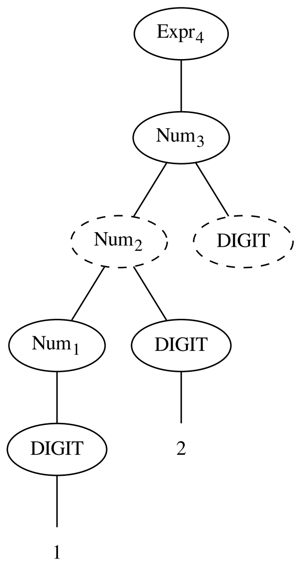
In addition to the functions `eval`, `applyRule`, and `glowLR`,

[*2] Demonstration Java code is available at https://github.com/ialab/LeftRecursion.



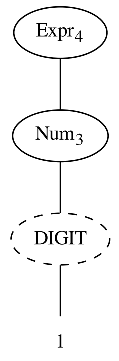
| | | |
|---|---|---|
| (Expr, 0) | = | MATCH, 4 |
| (Num, 0) | = | MATCH, 2 |
| (DIGIT, 0) | = | MATCH, 1 |
| (DIGIT, 1) | = | MATCH, 2 |
| (Num, 3) | = | MATCH, 4 |
| (DIGIT, 3) | = | MATCH, 4 |

(a)



| | | |
|---|---|---|
| (Expr, 0) | = | MATCH, 4 |
| (Num, 0) | = | MATCH, 2 |
| (DIGIT, 0) | = | MATCH, 1 |
| (DIGIT, 1) | = | MATCH, 2 |
| (Num, 3) | = | MATCH, 4 |
| (DIGIT, 3) | = | MATCH, 4 |
| (DIGIT, 2) | = | MISMATCH, 2 |

(b)



| | | |
|---|---|---|
| (Expr, 0) | = | MATCH, 4 |
| (Num, 0) | = | MATCH, 1 |
| (DIGIT, 0) | = | MATCH, 1 |
| (DIGIT, 1) | = | MATCH, 2 |
| (Num, 3) | = | MATCH, 4 |
| (DIGIT, 3) | = | MATCH, 4 |
| (DIGIT, 2) | = | MISMATCH, 2 |

(c)

**Fig. 8** Failure parse example using Goto et al.'s method.

two new functions, `applyRuleGrow` and `evalGrow`, are added to our parser. The `Ast` type is extended as in Goto et al. [3] and takes three values of MATCH, MISMATCH and FAIL. In addition to `ans` and `pos_next`, MEMO also records `grow`, which is a bool-type field that tells the caller that a left-recursive call has occurred.

### 4.1.1 The applyRule Function

The `applyRule` function in the proposed method is shown in **Fig. 9**. As the main difference from the basic `applyRule` function, this version records FAIL in the MEMO table before calling the `eval` function and writes a value in the MEMO table to indicate that a left-recursive call has occurred if FAIL has already been recorded in the MEMO table.

### 4.1.2 The glowLR Function

The `glowLR` function in the proposed method is shown in **Fig. 10**. Unlike Goto et al.'s version, the nonterminal symbol to which the `glowLR` function is applied is not limited to the start symbol, and thus repetition decision is based on the results of each parsing session, not on the maximum value of `POS` during the analysis.

### 4.1.3 The applyRuleGrow Function

The `applyRuleGrow` function is shown in **Fig. 11**. If, when matching an input string with a non-terminated symbol, the MEMO table has already recorded the parsing result of the non-terminated symbol, the behavior of the function changes depending on whether a left-recursive call occurs. If left-recursive calls do not occur up to this point, the record of the analysis results of the MEMO table can be returned as is, and the `applyRule` function takes care of this behavior. Otherwise, the

```
Ast applyRule(NonTerminal N, p){
  if(MEMO(N, p) == NULL){
    MEMO(N, p) = {FAIL, p, FALSE}
    m = MEMO(N, p)
    exp = getRuleDef(N)
    ans = eval(exp)
    m.ans = ans
    m.pos_next= POS
    if(m.grow == TRUE){
      growLR(N, p)
      m.grow = FALSE
      ans = m.ans
      POS = m.pos_next
    }
    return ans
  }else if(MEMO(N, p).ans == FAIL){
    MEMO(N, p).ans = MISMATCH
    MEMO(N, p).grow = TRUE
    return MEMO(N, p).ans
  }else{
    m = MEMO(N, p)
    POS = m.pos_next
    return m.ans
  }
}
```

**Fig. 9** applyRule in the proposed method.

```
void growLR(N, p){
  while(TRUE){
    old_pos = POS
    POS = p
    exp = getRuleDef(N)
    ans = evalGrow(exp, p, {N})
    if(ans != MATCH || POS <= old_pos)
      break
    MEMO(N, p).ans = ans
    MEMO(N, p).pos_next = POS
  }
}
```

**Fig. 10** glowLR in the proposed method.

```
Ast applyRuleGrow(NonTerminal N, p, limits){
  limits.add(N)
  exp = getRuleDef(N)
  ans = evalGrow(exp, p, limits)
  if(ans != MATCH || POS <= MEMO(N, p).pos_next){
    ans = MEMO(N, p).ans
    POS = MEMO(N, p).pos_next
  }else{
    MEMO(N, p).ans = ans
    MEMO(N, p).pos_next = POS
  }
  return ans
}
```

**Fig. 11** applyRuleGrow in the proposed method.

results stored in the MEMO table are used to perform the parsing again. This re-parsing should only be performed when the `evalGrow` function is repeatedly called in the `glowLR` function, and the `applyRuleGrow` function takes care of this process.

The `applyRuleGrow` function takes a set of nonterminal symbols, `limits`, as an additional argument. This set contains nonterminals that have already appeared in the parsing, which prevents the recursive calls from becoming too deep during the parsing.

#### 4.1.4 The evalGrow Function

The `evalGrow` function is shown in **Fig. 12**. The basic behavior of this function is the same as that of the `eval` function. It takes as an argument a set of nonterminal symbols, `limits`, and calls the `applyRuleGrow` function if a match is made with a nonterminal symbol at the parsing position where it is called.

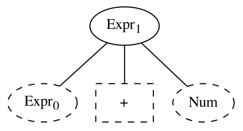
```
Ast evalGrow(ParsingExpression e, p, limits){
  if(e is the empty string ε)
    return MATCH

  if(e is a terminal str){
    if(the input at POS starts with str){
      POS = POS + (the length of str)
      return MATCH
    }else
      return MISMATCH
  }

  if(e is a negative lookahead !e_in){
    if(evalGrow(e_in, p, limits) == MATCH){
      POS = p
      return MISMATCH
    }else
      return MATCH
  }

  if(e is a nonterminal N)
    if(POS == p && N ∉ limits)
      return applyRuleGrow(N, POS, limits)
    else
      return applyRule(N, POS)
  }

  if(e is an ordered choice e1/e2){
    if(evalGrow(e1, p, limits) == MATCH)
      return MATCH
    else
      return evalGrow(e2, p, limits)
  }

  if(e is a concatenation e1e2){
    if(evalGrow(e1, p, limits) == MATCH){
      if(evalGrow(e2, p, limits) == MISMATCH){
        POS = p
        return MISMATCH
      }else
        return MATCH
    }else
      return MISMATCH
  }
}
```

**Fig. 12** evalGrow in the proposed method.

### 4.2 Examples of Analysis Using the Proposed Method

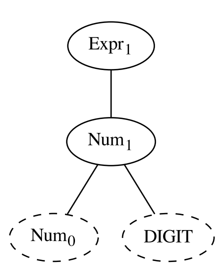
We will reuse the grammar used in the example in Section 3.3 as an example. Let the input string be parsed as `'12+34'`. The syntax tree and MEMO table composed during the parsing are shown in **Figs. 13** and **14**. The value of grow in the MEMO table is represented as T for TRUE and F for FALSE.

Parsing begins with the nonterminal symbol `Expr`. Before starting the matching process of the expression <Expr>'+'<Num>/<Num>, which is the right-hand side of `Expr`, FAIL is recorded in the MEMO table. When processing the first alternative `Expr '+' Num` shown in Fig. 13 (a), because FAIL is recorded in the MEMO table, the fact that a left-recursive call occurred is detected and recorded in the MEMO table. Also, the result of (conceptually numbered) nonterminal $\texttt{Expr}_0$, MISMATCH, is stored in the MEMO table, and returned. Because the result makes parsing with the expression <Expr>'+'<Num> fail, parser tries the next alternative, `Num`. Figure 13 (b) shows an attempt to match the input with the first alternative <Num><DIGIT> of the nonterminal symbol `Num`. The MEMO table contains FAIL for `Num`. The parser therefore records that a left-recursive call has occurred and the match fails. Because the first alternative failed, the parser tries to match the next alternative, `DIGIT`. Because the nonterminal `DIGIT` matches `'1'` in the input string, the result of



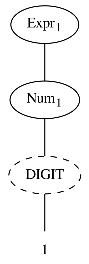
| | | |
|---|---|---|
| (Expr, 0) | = | Fail, 0, F |

(a)



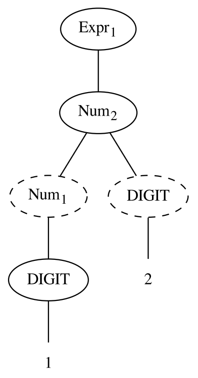
| | | |
|---|---|---|
| (Expr, 0) | = | MisMatch, 0, T |
| (Num, 0) | = | Fail, 0, F |

(b)



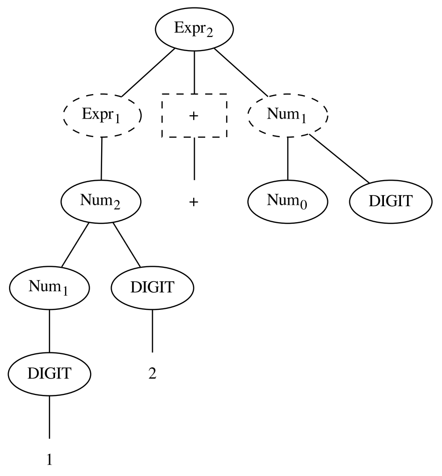
| | | |
|---|---|---|
| (Expr, 0) | = | MisMatch, 0, T |
| (Num, 0) | = | Match, 1, T |
| (DIGIT, 0) | = | Match, 1, F |

(c)



| | | |
|---|---|---|
| (Expr, 0) | = | MisMatch, 0, T |
| (Num, 0) | = | Match, 2, T |
| (DIGIT, 0) | = | Match, 1, F |
| (DIGIT, 1) | = | Match, 2, F |

(d)



| | | |
|---|---|---|
| (Expr, 0) | = | Match, 2, T |
| (Num, 0) | = | Match, 2, F |
| (DIGIT, 0) | = | Match, 1, F |
| (DIGIT, 1) | = | Match, 2, F |
| (DIGIT, 2) | = | MisMatch, 2, F |
| (Num, 3) | = | Fail, 3, F |

(e)

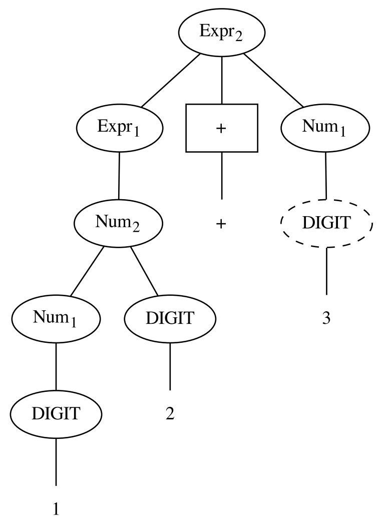
**Fig. 13** Example parse using proposed method #1.

Num$_1$ is determined. The state of the parser at this time is shown in Fig. 13 (c).

Since the result of Num$_1$ has now been determined and the value of (Num, 0).grow in the MEMO table is set to True, the parser starts repeating on the head rule named by Num. Figure 13 (d) shows the state of the parser at the point of attempting to match the first alternative of the nonterminal Num, <Num><DIGIT>. Now Num$_2$ turns out to match '12', because the result of Num$_1$ is recorded as Match. The parser then attempts to match Num$_3$, and the first alternative <Num$_2$><DIGIT> does not match the first part of the input string, '12+', whereas the second alternative <DIGIT> matches '1', using the contents of the MEMO table at (DIGIT, 0). However, the value of POS is reduced compared to the result of Num$_2$, and thus the parser ends the iteration with the head rule Num, and leaves the result of matching Num$_2$ in the MEMO table instead of the result of Num$_3$, and sets (Num, 0).grow to False.

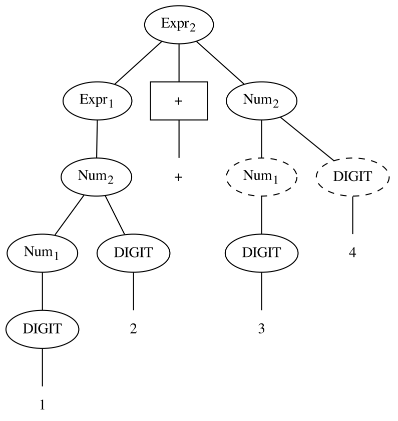
Because it is determined that Num matches the '12' in the input string, the result of parsing Expr$_1$ is determined. Because (Expr, 0).grow in the MEMO table is True, the parser starts a left recursion iteration with the head rule Expr. Figure 13 (e) shows the state of parsing at the time of the match with the first alternative of Expr, <Expr>'+'<Num>. Because Expr$_1$ matches '12' as does the next '+' also matches, the parser starts matching Num with the rest of the input string '34'. It records Fail in the MEMO table and tries to match '34' with the first alternative of Num, <Num><DIGIT>. At this time, because Fail is recorded in the MEMO table, the MEMO table is updated to indicate that a left-recursive call has occurred. The second alternative <DIGIT> of Num matches '3' in the input string and updates the MEMO table. The state at this time is shown in Fig. 14 (f).

Because the result of Num$_1$ is determined and (Num, 3).grow in the MEMO table is True, the parsing of the left recursion with Expr as the head rule is suspended, and the iteration with the head rule Num is started. Figure 14 (g) shows the state of the match with the first alternative <Num><DIGIT> of Num$_2$. Since the MEMO table contains a record showing that Num$_1$ matched the input substring '3', Num$_2$ matches '34'. The parser stores this result in the



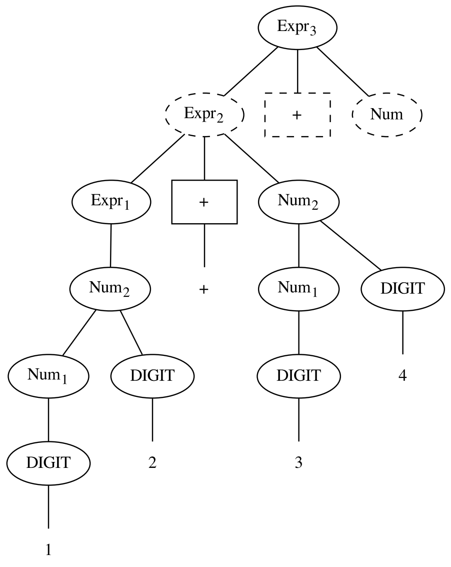
| | | |
|---|---|---|
| (`Expr`, 0) | = | Match, 2, T |
| (`Num`, 0) | = | Match, 2, F |
| (`DIGIT`, 0) | = | Match, 1, F |
| (`DIGIT`, 1) | = | Match, 2, F |
| (`DIGIT`, 2) | = | MisMatch, 2, F |
| (`Num`, 3) | = | Match, 4, T |
| (`DIGIT`, 3) | = | Match, 4, F |

(f)



| | | |
|---|---|---|
| (`Expr`, 0) | = | Match, 2, T |
| (`Num`, 0) | = | Match, 2, F |
| (`DIGIT`, 0) | = | Match, 1, F |
| (`DIGIT`, 1) | = | Match, 2, F |
| (`DIGIT`, 2) | = | MisMatch, 2, F |
| (`Num`, 3) | = | Match, 5, T |
| (`DIGIT`, 3) | = | Match, 4, F |
| (`DIGIT`, 4) | = | Match, 5, F |

(g)



| | | |
|---|---|---|
| (`Expr`, 0) | = | Match, 5, F |
| (`Num`, 0) | = | Match, 2, F |
| (`DIGIT`, 0) | = | Match, 1, F |
| (`DIGIT`, 1) | = | Match, 2, F |
| (`DIGIT`, 2) | = | MisMatch, 2, F |
| (`Num`, 3) | = | Match, 5, F |
| (`DIGIT`, 3) | = | Match, 4, F |
| (`DIGIT`, 4) | = | Match, 5, F |
| (`DIGIT`, 5) | = | MisMatch, 5, F |

(h)

**Fig. 14** Example parse using proposed method #2.

MEMO table and then tries to match the rest of the input string with $\texttt{Num}_3$. Matching the first alternative <`Num`><`DIGIT`> will fail because there are no more characters left in the input string to read. The match with the next alternative <`DIGIT`> matches `‘3’` based on the contents of the MEMO table; however, because the value of `POS` is less than the result of $\texttt{Num}_2$, the iteration with `Num` as the head rule is terminated, and the pending iteration of left recursion with `Expr` as the head rule is resumed.

The state of resumed parsing of the left recursion with `Expr` as the head rule is shown in Fig. 14 (h). Matching by the first alternative <`Expr`>`‘+’`<`Num`> will fail because $\texttt{Expr}_2$ will match the entire input string `‘12+34’` and there will be no match for the next `‘+’`. The next alternative <`Num`> matches `‘12’` from the contents of the MEMO table. At this time, because `POS` is less than the value in the case of $\texttt{Expr}_2$, the parser ends the iteration with `Expr` as the head rule, leaving the parsing results of $\texttt{Expr}_2$ in the MEMO table instead of adopting the parsing results of $\texttt{Expr}_3$.

This causes the entire input string `‘12+34’` to be parsed by the nonterminal `Expr`, and the parsing succeeds.

```
                          <Primary> ← <PrimaryNoNewArray>
                <PrimaryNoNewArray> ← <ClassInstanceCreationExpression>
                                      /<MethodInvocation>
                                      /<FieldAccess>
                                      /<ArrayAccess>
                                      /‘this’
  <ClassInstanceCreationExpression> ← ‘new ’<ClassOrInterfaceType>‘()’
                                      /<Primary>‘.new ’<Identifier>‘()’
                 <MethodInvocation> ← <Primary>‘.’<Identifier>‘()’
                                      /<MethodName>‘()’
                      <FieldAccess> ← <Primary>‘.’<Identifier>
                                      /‘super.’<Identifier>
                      <ArrayAccess> ← <Primary>‘[’<Expression>‘]’
                                      /<ExpressionName>‘[’<Expression>‘]’
             <ClassOrInterfaceType> ← <ClassName>/<InterfaceTypeName>
                        <ClassName> ← ‘C’/‘D’
                <InterfaceTypeName> ← ‘I’/‘J’
                       <Identifier> ← ‘x’/‘y’/<ClassOrInterfaceType>
                       <MethodName> ← ‘m’/‘n’
                   <ExpressionName> ← <Identifier>
                       <Expression> ← ‘i’/‘j’
```

**Fig. 15** Java primary expression grammar.

## 5. Evaluation

In this section, we evaluate the proposed method by comparing the method with Warth et al.'s method [6] and Goto et al.'s method [3]. Specifically, we evaluate our method in the following aspects.

- Is it possible to analyze a grammar that can be analyzed using Warth et al.'s method?
- Is it possible to analyze a grammar that can be analyzed using Goto et al.'s method?
- Is it possible to analyze a grammar that has not been analyzed using Goto et al.'s method?
- How does the number of function calls differ with these methods?

### 5.1 Java Primary Expressions

Using the simplified Java primary expression grammar used in the evaluation of Warth et al. [6], we show that the proposed method supports grammar classes that can be handled by existing methods.

The grammar to be used is shown in **Fig. 15**. Note that the start symbol is the first rule of the grammar, `Primary`.

Using this grammar, we parse the following input.

- `‘this’`
- `‘this.x’`
- `‘this.x.y’`
- `‘this.x.m()’`
- `‘x[i][j].y’`

The results are shown in **Table 2**.

**Table 2** Parsing result of Java primary expressions.

| | Warth | Goto | Proposed |
|---|---|---|---|
| `‘this’` | Match | MisMatch [*3] | Match |
| `‘this.x’` | Match | Match | Match |
| `‘this.x.y’` | Match | Match | Match |
| `‘this.x.m()’` | MisMatch | MisMatch | MisMatch |
| `‘x[i][j].y’` | Match | Match | Match |

Note that the result for `‘this.x.m()’` is MisMatch for all methods, because the input string does not match the Java primary expression syntax. In addition, the result of the input `‘this’` in Goto et al.'s method is MisMatch. This is because their method fails to parse some class of inputs that do not require repetition.

### 5.2 Multiple Left-recursive Calls Occurring at the Same Position

The following was presented by Goto et al. [3] as an example of a grammar that cannot be handled correctly using Warth et al's method because multiple left-recursive calls occur at the same position.

```
<S> ← <A>‘b’/‘b’
<A> ← <A>‘a’/<S>‘a’
```

[*3] Although Goto et al. [3] claimed to have successfully analyzed these input strings, we were unable to reproduce their results.

The result of the analysis using this grammar is shown in **Table 3**. When the parse terminates abnormally, it is represented as Error.

## 5.3 Left-recursive Calls that Occur at Multiple Positions

The following grammar causes left-recursive calls at multiple positions in addition to left-recursive calls at the same position.

<S> ← <A>'-'<A>

<A> ← <B>'b'/'b'

<B> ← <B>'a'/<A>'a'

The results are shown in **Table 4**. In this section, we empirically showed that the proposed method can handle some grammar classes that were not handled correctly by the existing methods. In addition, based on the testing thus far, the proposed method can successfully handle all grammar classes that can be handled by previous methods.

## 5.4 Number of Function Calls

Consider the following grammar.

<S> ← $<A>^k$<L>

<A> ← $\varepsilon$

<L> ← <L>'1'/$\varepsilon$

where $<A>^k$ denotes a sequence of $k$ copies of a nonterminal A.

**Table 3** Parse with a grammar that includes multiple occurences of left recursion at the same position.

| | Warth | Goto | Proposed |
|---|---|---|---|
| 'b' | Error | MisMatch [*3] | Match |
| 'bab' | Error | Match | Match |
| 'baab' | Error | Match | Match |
| 'baabab' | Error | Match | Match |
| 'baabaab' | Error | Match | Match |

**Table 4** Parse with a grammar that includes multiple occurences of left recursion at multiple positions.

| | Warth | Goto | Proposed |
|---|---|---|---|
| 'b-b' | Error | MisMatch | Match |
| 'bab-b' | Error | Match | Match |
| 'b-bab' | Error | Match | Match |
| 'bab-bab' | Error | MisMatch | Match |
| 'babab-babab' | Error | MisMatch | Match |

The input string that can be parsed by this grammar is a string with an arbitrary number of 1s. The total number of calls to `eval` and `evalGrow` functions when k and n are increased during the parse using this grammar and input, respectively, is measured. Calls to `eval` and/or `evalGrow` from `applyRule` and `glowLR` functions are counted. Recursive calls from within `eval` and `evalGrow` themselves are not counted.

We measured the results when $n$ was increased from zero to 100 by 10 for $k$ = 1, 10, and 100, respectively. The results are shown in **Tables 5**, **6**, and **7**, and **Figs. 16**, **17**, and **18**.

The horizontal axis of the graph is the length of the input string and the vertical axis is the number of calls.

These results confirm that the proposed method is efficient and competitive with existing methods.

## 6. Conclusion

### 6.1 Summary

Because the basic packrat parser parse the input strings by recursively calling functions corresponding to the grammar rules, parsing using the parser with a left-recursive grammar does not terminate. Warth et al. [6] and Goto et al. [3] have proposed solutions to this problem. However, Warth et al.'s method [6] does not take into account the case in which multiple left-recursive calls occur at the same point of input, and thus the parsing terminates abnormally. Goto et al.'s method [3] fails when multiple

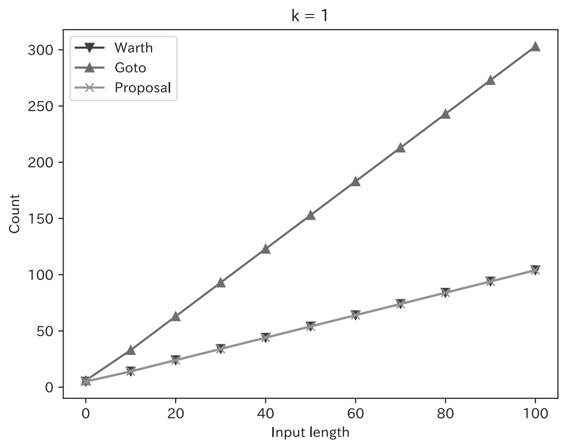


**Fig. 16** Count of `eval` and evalGrow with different length of input ($k$ = 1).

**Table 5** Function call count with $k$ = 1.

| | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Warth | 5 | 14 | 24 | 34 | 44 | 54 | 64 | 74 | 84 | 94 | 104 |
| Goto | 6 | 33 | 63 | 93 | 123 | 153 | 183 | 213 | 243 | 273 | 303 |
| Proposal | 5 | 14 | 24 | 34 | 44 | 54 | 64 | 74 | 84 | 94 | 104 |

**Table 6** Function call count with $k$ = 10.

| | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Warth | 14 | 23 | 33 | 43 | 53 | 63 | 73 | 83 | 93 | 103 | 113 |
| Goto | 24 | 132 | 252 | 372 | 492 | 612 | 732 | 852 | 972 | 1,092 | 1,212 |
| Proposal | 14 | 23 | 33 | 43 | 53 | 63 | 73 | 83 | 93 | 103 | 113 |

**Table 7** Function call count with $k$ = 100.

| | 0 | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Warth | 104 | 113 | 123 | 133 | 143 | 153 | 163 | 173 | 183 | 193 | 203 |
| Goto | 204 | 1,122 | 2,142 | 3,162 | 4,182 | 5,202 | 6,222 | 7,242 | 8,262 | 9,282 | 10,302 |
| Proposal | 104 | 113 | 123 | 133 | 143 | 153 | 163 | 173 | 183 | 193 | 203 |

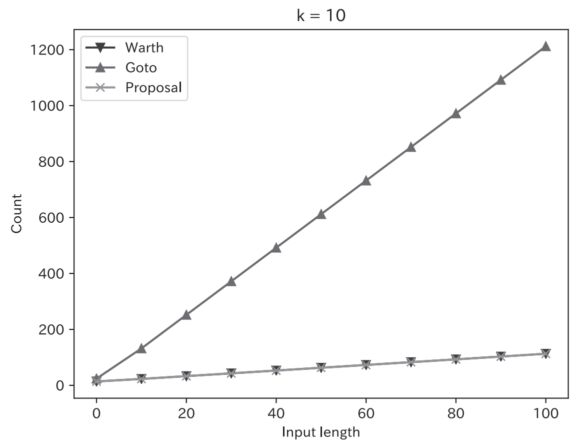


**Fig. 17** Count of `eval` and evalGrow with different length of input ($k = 10$).

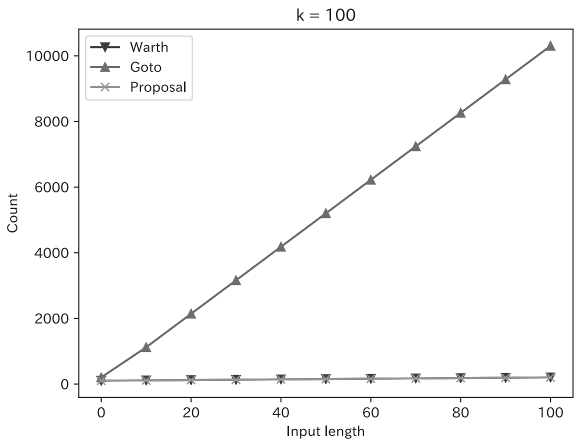


**Fig. 18** Count of `eval` and evalGrow with different length of input ($k = 100$).

left-recursive calls occur at multiple points, although it can handle multiple left-recursive calls at the same position.

In this study, we proposed a method to parse the most recently detected left-recursion first and showed that the method extends the range of grammars. In the evaluation experiments, we confirmed that our method can handle all grammars that previous methods have been able to handle, and can handle some grammars that existing methods have failed to parse. Moreover, the measurement results of the number of calls to the `eval` function during an analysis indicate that the analysis time is likely to be shorter than that of Goto et al.'s method.

### 6.2 Future Tasks and Prospects

Although we have demonstrated that the proposed method can handle more general forms of left recursion, it is yet unknown whether this method can handle all forms of left-recursive PEGs. Our future aim will be to formalize the method and provide proofs of the method to determine the range of grammars that can be handled.

In Warth et al.'s method [6], Tratt [5] pointed out that the right recursion takes precedence when parsing with grammars that contain both a left and right recursion, as in the case of the following.

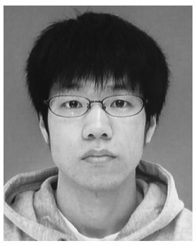
`<Expr> ← <Expr>'+'<Expr>/<Num>.`

We regard this as a problem related to the ambiguity of a PEG that has arisen since left recursion first became available for parsing. The ability of a PEG to handle left recursion may lead to the loss of the PEG's determinism. With regard to this, we need to define a new semantics of PEGs that can handle left recursion while maintaining the determinism.

Medeiros et al. [4] proposed a parsing algorithm for PEGs that allows for left recursion. Their algorithm uses the memoization table only to maintain a growing parse tree in left recursion, which result in an exponential worst-case computation time regardless of whether a grammar is left-recursive. By contrast, our algorithm applies a memoized analysis for non-left-recusive rules as in a usual packrat parser. For left-recursive rules, we expect an almost-linear compexity to be exhibited, although this estimate needs to be proven.

## References

[1] Ford, B.: Packrat parsing: Simple, powerful, lazy, linear time, functional pearl, *ACM SIGPLAN Notices*, Vol.37, No.9, pp.36–47 (2002).
[2] Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation, *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.111–122 (2004).
[3] Goto, Y., Shirata, Y., Kiyama, M. and Ashihara, H.: Implementation of packrat parser with update detection to parse left recursive grammars (in Japanese), *IPSJ Trans. Programming* (*PRO*), Vol.4, No.3, pp.84–93 (2011).
[4] Medeiros, S., Mascarenhas, F. and Ierusalimschy, R.: Left recursion in parsing expression grammars, *Science of Computer Programming*, Vol.96, pp.177–190 (2014).
[5] Tratt, L.: Direct left-recursive parsing expression grammars, *School of Engineering and Information Sciences, Middlesex University* (2010).
[6] Warth, A., Douglass, J.R. and Millstein, T.: Packrat parsers can support left recursion, *Proc. 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pp.103–110 (2008).



**Masaki Umeda** graduated from the College of Information Science, University of Tsukuba in 2020. He is currently a student in the Master's program in computer science, Graduate School of Systems and Information Engineering, University of Tsukuba. His research interest is parsing of programming languages.

**Atusi Maeda** received his Ph.D. in engineering from Keio University. He became a research associate at the University of Electro-Communications in 1997. He is currently an associate professor in the Faculty of Engineering, Information and Systems, University of Tsukuba. His research interests include the implementation of programming languages, runtime systems, and dynamic resource management. He is a member of ACM and JSSST.