Is Stateful Packrat Parsing Really Linear in Practice? A Counter-Example, an Improved Grammar, and Its Parsing Algorithms

Nariyoshi Chida
NTT Secure Platform Laboratories,
NTT Corporation
Japan
nariyoshi.chida.nx@hco.ntt.co.jp

Yuhei Kawakoya
NTT Secure Platform Laboratories,
NTT Corporation
Japan
yuuhei.kawakoya.sy@hco.ntt.co.jp

Dai Ikarashi
NTT Secure Platform Laboratories,
NTT Corporation
Japan
dai.ikarashi.rd@hco.ntt.co.jp

Kenji Takahashi
NTT Security
USA
Kenji.Takahashi@nttsecurity.com

Koushik Sen
University of California, Berkeley
USA
ksen@cs.berkeley.edu

Abstract

Stateful packrat parsing is an algorithm for parsing syntaxes that have context-sensitive features. It is a well-known knowledge among researchers that the running time of stateful packrat parsing is linear for real-world grammars, as demonstrated in existing studies. However, we have found the cases in real-world grammars and tools that lead its running time to become exponential.

This paper proposes a new grammar, *parsing expression grammar with variable bindings*, and two parsing algorithms for the grammar, *stateful packrat parsing with selected global states* and *stateful packrat parsing with conditional memoization*. Our proposal overcomes the exponential behavior that appears in parsers and guarantees polynomial running time. The key idea behind our algorithms is to memoize the information relevant to the use of the global states in order to avoid memoizing the full global states. We implemented our algorithms as a parser generator and evaluated them on real-world grammars. Our evaluation shows that our algorithms significantly outperform an existing stateful packrat parsing algorithm in terms of both running time and space consumption. In particular, stateful packrat parsing with conditional memoization improves the running time and space consumption for malicious inputs that lead to exponential behavior with the existing algorithm by 260x and 217x, respectively, compared to the existing algorithm.

CC '20, February 22–23, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7120-9/20/02.
https://doi.org/10.1145/3377555.3377898

CCS Concepts • **Theory of computation → Parsing**; • **Software and its engineering → Parsers**;

Keywords Packrat parsing, stateful parsing, parsing expression grammars, and memoization

ACM Reference Format:
Nariyoshi Chida, Yuhei Kawakoya, Dai Ikarashi, Kenji Takahashi, and Koushik Sen. 2020. Is Stateful Packrat Parsing Really Linear in Practice? A Counter-Example, an Improved Grammar, and Its Parsing Algorithms. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377555.3377898

1 Introduction

Parsing is a widely-used technique for recognizing an input string based on pre-defined specifications, e.g., it is used in the front-end phase of a compiler. One of the significant challenges in parsing is to reduce its running time because the huge running time for parsing easily leads the parsing into a stalling or denial-of-service (DoS) state (or crash in the worst case) [5][7][20][21]. Therefore, it is crucial to reduce the running time for arbitrary inputs including malicious inputs. One prominent approach for reducing the running time is packrat parsing, a recursive descent parsing algorithm with backtracking and memoization [8][9]. Especially, stateful packrat parsing (SPP) [8][9], which is an extended version of packrat parsing with incorporating global states and operators like the global states and semantic actions of Yacc[17], is practically used for parsing syntaxes that have context-sensitive features such as: (1) Typedef of C/C++ [12]; (2) Opening and closing tags of XML [15]; and (3) Indentations of Haskell, Python, and F# [3].

Several studies for SPP have been conducted until now [15][16]; those studies demonstrated that the running time of their algorithm is linear even for real-world grammars through their experiments. However, we have found several cases that did not match to their experimental results. More precisely, we have discovered the cases in real-world grammars whose running time for parsing becomes exponential. The detail of it is shown in the following paragraphs.

Problem. Consider the following simplified HTML grammar defined by the Extended Backus-Naur Form (EBNF) with three operators: bind, match, and scope. This grammar accepts simplified HTML documents that consist of opening tags, or the pairs of the opening and closing tags that have the same tag names. The three operators are used to check the correspondence of the tag names. Specifically, the bind operator captures an opening tag, and the match operator refers the opening tag to check whether the tag name is the same as the one of a closing tag. Here, captured opening tags are stored in a global state, and the global state is represented as a list of key-value pairs. The scope operator releases the captured opening tags. This grammar is a simplified version of ANother Tool for Language Recognition 4 (ANTLR4) grammar for HTML[1] with removing the rules not related to the causes of exponential running time for explanation.

$$\begin{array}{lcl} \texttt{HTML} & = & \texttt{scope('<'bind(}v\texttt{, Name)'>'HTML*'< /'match(}v\texttt{, Name)'>')} \\ & | & \texttt{'<'Name'>'} \\ \texttt{Name} & = & [a-zA-Z]\texttt{+} \end{array}$$

Also, given the following simple HTML document: *<html><body><br><img><hr></body></html>*. Even though this document seems like a normal HTML document, it becomes a problem in the context of SPP, i.e., it incurs the exponential increase of running time. This is because the number of global states grows exponentially. We can see the exponential growth from the parsing of the simple HTML document. The following is a part of the global states that appeared in the parsing, and the number of the global states is 2^3 against the 3 opening tags: *br*, *img*, and *hr*.

① $[\cdots]$
② $[\cdots (v, br)]$
③ $[\cdots (v, br), (v, img)]$
④ $[\cdots (v, br), (v, img), (v, hr)]$
⑤ $[\cdots (v, br), (v, hr)]$
⑥ $[\cdots (v, img)]$
⑦ $[\cdots (v, img), (v, hr)]$
⑧ $[\cdots (v, hr)]$
$\cdots$

In fact, we confirmed in our preliminary experiment the exponential increase of running time happened when we parsed the document that consists of many contiguous opening tags with widespread tools, Nez [15] and Pegasus (43,146 total downloads in 2019) [11]. More details of this experiment are described in §3.

We consider that the causes of this increase of running time are two-fold: (i) *Cost of parse function:* SPP is so flexible that it accepts various types of global states and operators, even including the ones that lead to exponential running time (**Problem 1**). (ii) *Size of memoization table:* the size of memoization table grows exponential due to the inputs that frequently change the global states as we saw above (**Problem 2**).

Proposal. In this paper, we present SPP that guarantees polynomial running time. To this end, we introduce a new formalism and two algorithms for parsing. In detail, to solve Problem 1, i.e., to reduce the cost of parse functions to polynomial, we need to restrict the use of the global states. To enforce the restrictions, we develop *parsing expression grammars with variable bindings* (V-PEGs), an extension of *parsing expression grammars* (PEGs) [10]. V-PEGs have operators that handle the global states and the operators enable us to express context-sensitive syntaxes by adhering to discipline that achieves polynomial running time. Moreover, despite the restriction, V-PEGs can express all real-world grammars described in [15].

For Problem 2, i.e., to reduce the size of the memoization table to polynomial, we introduce two types of memoization algorithm: *stateful packrat parsing with selected global states* (SG-SPP for brevity) and *stateful packrat parsing with conditional memoization* (CM-SPP for brevity). The SG-SPP is a simple improvement of existing SPP. As with the existing SPP, the SG-SPP extends the memoization table with global states. A difference originates from the use of the global states in the keys of the memoization table between the SG-SPP and the existing algorithms. That is, SG-SPP only uses a part of global states that affects the later parsing as keys of the memoization table, while existing SPP uses the entire global states. SG-SPP guarantees polynomial running time, but unfortunately, its running time and space consumption is still a problem in practice because the parser based on the algorithm always runs in the worst-case time and space complexities against *malicious inputs*, the inputs that change the global states but the changes do not affect the later parsing. For example, in the case of the HTML grammar, the size of the memoization table of the parser implemented by the algorithm grows approximately $n^2/2$ where n is the number of opening tags, and the running time of the parser is $O(n^2)$. Due to the running time and space consumption problem, the major practical challenge in implementing a stateful packrat parser is to memoize the parsing results without using the global states as keys of the memoization table.

To address this problem, we present another new stateful parsing algorithm called CM-SPP. The key insight to this algorithm is that the existing SPP uses global states as a key of the memoization table in order to uniquely determine the results of operators affected by the global states. On the contrary, we use the results of the operators affected by global states as a key of the memoization table. This guarantees that the parser based on CM-SPP runs in linear time against the malicious inputs.

We have implemented SG-SPP and CM-SPP as a proof of concept[2]. Then, we performed experiments to compare the

[1] https://github.com/antlr/grammars-v4/blob/master/html/HTMLParser.g4

[2] Our code and full experimental results are available online at [2].

performance of these algorithms with other parsing algorithms. Our experimental results show that our algorithms significantly improve on the running time and space consumption, compared to the existing parsing algorithm. In particular, CM-SPP improves the running time and space consumption even for malicious inputs that lead to exponential behavior of the existing work by 260x and 217x, respectively, compared to the ones of other parsing algorithms.

Consequently, we believe that our proposal is more suitable for applying to real-world grammars than existing parsing algorithms because it allows us to avoid the huge running time that leads the parsing into a stalling or DoS state and also allows us to estimate the maximum time and space complexities in the worst cases.

Contributions. The contributions of this paper are summarized below.

- We show that existing SPP yields exponential running time even with real-world grammars (§3). To the best of our knowledge, we are the first to report that SPP incurs exponential running time even with real-world grammars and to provide inputs that cause exponential behavior.
- We present V-PEGs, an extension of PEGs that enforces discipline to achieve polynomial running time (§5).
- We introduce new parsing algorithms called SG-SPP (§6) and CM-SPP (§8) that guarantee polynomial running time and space consumption and outperform the other strategies. In particular, CM-SPP improves the running time and space consumption for malicious inputs that lead to exponential behavior of the existing work by 260x and 217x, respectively, compared to those for other strategy (§9).

2 Preliminaries

2.1 Packrat Parsing

Broadly speaking, packrat parsing is recursive descent parsing with backtracking and memoization. Packrat parsing exploits the insight that the packrat parser always returns the same parsing results when the parser tries to parse a nonterminal in the same *parsing state*. Here, the parsing state for packrat parsing is a tuple of a nonterminal and an input position. To cache the parsing results, the packrat parser has a memoization table $M : N \times I \longrightarrow (I \cup \{\text{fail}\})$ where N is a set of nonterminals, I is a set of positions on an input string x, i.e., $I = \{i \mid 0 \leq i \leq |x|\}$, and $I \cup \{\text{fail}\}$ is a set of parsing results. If $M(A, i) = j$, then it means that A matches the substring of x starting at index i and ending before the index j. $M(A, i) = \text{fail}$ denotes that A cannot match any substring starting at i.

An example of how this works is omitted due to the space constraints. We refer the reader to [2] for more details.

2.2 Stateful Packrat Parsing

SPP is an extension of packrat parsing [8][9] that recognizes context-sensitive features in real-world grammars. Basically, the parsing strategy of SPP is the same as that for packrat parsing. The only difference comes from the fact that SPP has global states while packrat parsing does not. More precisely, SPP allows us to incorporate desired global states and operations in the parser.

To understand how this works, we consider the following EBNF grammar for XML:

$$
\begin{array}{lcl}
\texttt{XML} & = & \text{scope}(\text{'<' bind}(v, \texttt{Name})\ \text{'>'}\ \texttt{XML}\ \text{'</' match}(v, \texttt{Name})\ \text{'>'}) \\
 & \mid & \text{'<'}\ \texttt{Name}\ \text{'/>'} \\
 & \mid & \text{''} \\
\texttt{Name} & = & [a-zA-Z]+
\end{array}
$$

Here, we incorporated a list of key-value pairs as a global state and three operators: bind, match, and scope, in the parser. The list is used to store the opening tags. The bind(v, e) operator binds a substring matched to the expression e, i.e., an opening tag, into the variable v, and stores v into the list. The match(v, e) operator checks whether v is in the list, and if so, checks whether the string bound to v is the same as the substring matched to e. The scope(e) operator eliminates all variables bound in e from the list.

Due to the extension, the parsing state for packrat parsing is insufficient to determine the parsing results for SPP because the global state affects the parsing results. Therefore, the parsing state for SPP also contains the global state. As such, the memoization table for SPP is defined as $M_E : N \times I \times E \longrightarrow (I \cup \{\text{fail}\}) \times E$ where E is a set of global states.

SPP can be modeled by parse function `parse`. This function takes expression e and input position i, and then returns the next input position or fail. For example, we consider the same XML grammar described in this section. We assume that the input string is *<a></a>*. In this case, `parse(XML, 0)` returns 7 since the nonterminal `XML` matches the entire string. Note that the position is 0-indexed.

Fig. 1 shows the behavior of SPP for the XML grammar described in this section. We assume that the input string is *<a><b></b></a>*.

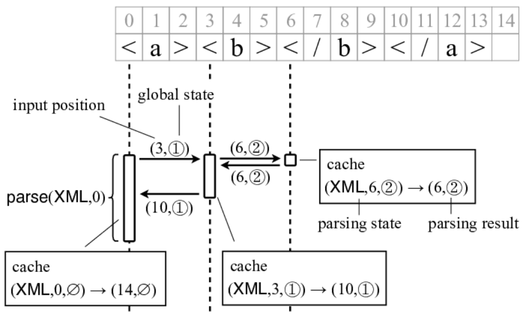

Figure 1. Behavior of SPP for XML grammar

Here, the following are the global states in Fig. 1.

① $[(v,a)]$ ② $[(v,a),(v,b)]$

In this figure, we only focus on the nonterminal XML. That is, we do not consider the memoization pertaining to nonterminal Name. The rounded rectangles on the dashed lines correspond to executions of parsing functions. The arrows pointing to the right denote function calls and those pointing to the left denote exits of the function. The rectangles denote entries of the memoization table. In this example, SPP caches three parsing results. In the rest of this paper, we express the behavior of SPP in the same manner.

3 Motivating Counter-Example

We demonstrate that existing algorithms yield exponential running time even on real-world grammars. Let us consider our example discussed in §1.

```
HTML  =  scope('<'bind(v, Name)'>'HTML*'< /'match(v, Name)'>')
      |  '<'Name'>'
Name  =  [a − zA − Z]+
```

The parser for the HTML grammar implemented using SPP runs in exponential time of the number of the opening tags. To understand why, we consider the following short input string: *<a><b><c>*.

Fig. 2 shows the behavior of the SPP for the HTML grammar.

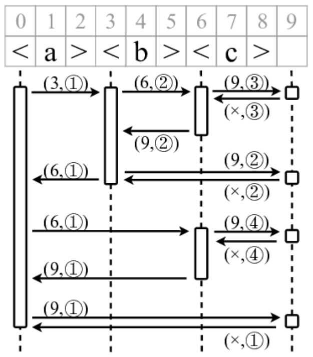

Figure 2. Behavior of stateful packrat parser for HTML grammar

Here, × denotes that the matching failed. The global states are as follows.

① $[(v,a)]$
② $[(v,a),(v,b)]$
③ $[(v,a),(v,b),(v,c)]$
④ $[(v,a),(v,c)]$

In this example, SPP caches eight parsing results. Fig. 3 shows the memoization table.

At this point, the number of entries in the memoization table is 2^3. In the same way, the number of the entries grows

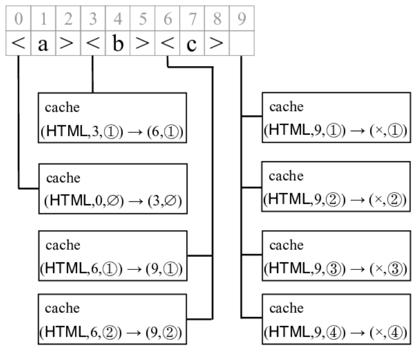

Figure 3. Memoization table

exponentially as the number of opening tags increases. More precisely, when the number of opening tags is i, the number of entries is 2^i. In fact, we demonstrate the inefficiency of Nez [14][3], a parser generator based on SPP, for the HTML grammar takes exponential time. Nez takes 67 min. to parse a string that comprises 37 opening tags, i.e., *<a>...<z><A>...<K>*, whereas our parser takes 0.001 s.

Unfortunately, from an engineering perspective, avoiding writing grammars that yields exponential running time in advance is very hard because when the parser yields exponential running time is unpredictable. In fact, the above counter-example has not been recognized. Therefore, our goal is to guarantee polynomial running time for arbitrary inputs including malicious inputs by providing an efficient parsing algorithm.

4 Overview

This section gives an informal overview of our proposal to achieve polynomial time. The time complexity of SPP is modeled by (cost of a parse function) × (size of a memoization table). For the cost of a parse function, we enforce discipline to SPP (§ 4.1). For the size of a memoization table, we introduce two types of memoization algorithm: SG-SPP (§ 4.2) and CM-SPP (§ 4.3).

4.1 Grammars

We begin by enforcing discipline to SPP since SPP allows us to incorporate desired global states and operations that may yield exponential behavior. We enforce that our algorithm has only two global states E_m and E_e, called *environments*, and five operators bind$_m$(v,e), bind$_e$(v,e), match(v,e), exists(v,e), and scope(e), where v is a variable and e is an expression. This discipline allows us to use global states and operators only to capture substrings and refer to the captured substrings. That is, bind$_m$ and bind$_e$ are used to capture substrings for match and exists, respectively. match and exists are used to refer a captured substring. scope releases all the

[3]We used the latest version of Nez-1.0-1069 (beta) on Java JVM-9.0.4 (October 2019)

captured substrings in the parsing. E_m and E_e are lists of key-value pairs to store the captured substring for match and exists, respectively. This discipline is based on the idea that real-world grammars that have context-sensitive features can be expressed by match and exists [15]. We introduce the discipline as V-PEGs. The formal definition is described in §5.

4.2 Simple Algorithm

Next, we describe SG-SPP that runs in polynomial time on V-PEGs. Basically, we simply apply SPP to the parsing of V-PEGs. There is, however, a main difference between SPP and the SG-SPP: SG-SPP only uses the latest values bound to variables as the parsing state, whereas SPP uses the entire global states. This is based on the insight that the results of the operators only depends on the latest values or the values appended in the later parsing. The details and the complexity are described in §6.

SG-SPP guarantees polynomial time and space. However, a significant practical problem remains. The algorithm always runs in the worst-case space consumption against the *malicious* inputs such as the input shown in §3. More precisely, we define the malicious input as an input that leads the behavior of the parser that calls $\mathtt{bind}_m$, but does not call match or does not match to e of $\mathtt{match}(v, e)$. To address the problem, we next introduce a new stateful parsing that overcomes the huge space consumption due to the malicious inputs.

4.3 More Efficient Algorithm

The huge running time and space consumption are due to the fact that SG-SPP incorporates global states into a parsing state of the memoization table. Because, as described in §3, SG-SPP does not reuse the parsing result in the memoization table when the global state is changed even if the change does not affect the parsing result. To solve this problem, we focus on a key of the memoization table. In existing SPP, the key is a global state. However, a global state can be modified meaninglessly as we pointed out and it causes enormous tries of memoization. Thus, we take a significantly different approach: Instead of a global state, we adopt the results of operators, which are basically affected by global states but not affected by meaningless modification of them. We describe CM-SPP in more detail in §7 and §8.

5 V-PEG

We now define V-PEGs, an extension of PEGs to recognize context-sensitive features.

Notations. Throughout this paper, we use the following notations. We write $a, b, c, d \in \Sigma$ for a character; $x, y \in \Sigma^*$ for a sequence of characters; $v \in V$ for a variable; $i, j \in I$ for a position on an input string; and $b \in \{true, false\}$ for a boolean value. For string $x = x[0] \cdots x[n-1]$, its length is $|x| = n$. For $0 \leq i \leq j < n$, string $x[i] \cdots x[j]$ is called a substring of x. We write $x[i..j]$ for the substring of x. Let t be tuple $(t_1, t_2, ..., t_m)$, where t_i is the i-th element of t. We write $t.t_i$ for obtaining the i-th element of t. Let f be a (partial) function. Then, $f[\alpha \mapsto \beta]$ denotes the (partial) function that maps α to β and behaves as f for all other arguments. In addition, $\mathtt{dom}(f)$ denotes the domain of f.

Notations for environments. For simplicity, we write E to denote E_m and E_e. We write $E :: (v, x)$ for appending key-value pair (v, x) to E. In addition, we write $E :: E'$ for appending all key-value pairs in environment E' to E in order. We use $E[\![v]\!]$ to denote the value that corresponds to v. If E has more than one pair such that the variable is v, the rightmost one takes precedence. For example, let E be $[(v_1, a), (v_2, b), (v_1, c)]$. Then, $E[\![v_1]\!]$ returns c. If such a pair is not in E, $E[\![v]\!]$ returns a special character $\bullet$ that is not contained in any string. We define the function $in(E, (v, x))$ that checks whether or not (v, x) is in E, and returns a result as a boolean value. For example, let E be $[(v_1, a), (v_2, b)]$. Then, $in(E, (v_1,a))$ is $true$ and $in(E, (v_2,c))$ is $false$. We use $(v, \_)$ to denote a key-value pair where the key is v and the value is arbitrary. We write $E[i..j]$ for a list that consists of the i-th element of E to the j-th element of E. $\mathcal{E}$ denotes a finite set of distinct states of E. We write $|E|$ for the size of E, that is, the number of key-value pairs in E.

5.1 Grammars

Definition 1. *A V-PEG is a quintuple $G = (N, \Sigma, R, V, e_S)$ where N is a finite set of nonterminals, Σ is a finite set of terminals, R is a finite set of rules, V is a finite set of variables, and e_S is a parsing expression with variable bindings termed as a start expression.*

We use $A = e$ for a rule in R, which is a mapping from nonterminal $A \in N$ to a parsing expression with variable bindings e. Hereafter, we refer to a parsing expression with variable bindings as an expression for simplicity. We write $R(A)$ to represent an expression e, which is associated by $A = e$. An expression e is the main specification for describing syntactic constructs. Fig. 4 shows the syntax of an expression e. In Fig. 4, we use e_p to denote a parsing expression. Thus, note that nonterminal A_p does not involve operators for handling environments, i.e., $\mathtt{bind}_m$, $\mathtt{bind}_e$, match, exists, and scope.

Most of the operators of V-PEGs exactly mirror the operators of PEGs. That is, empty ϵ matches an empty string. Character a exactly matches the same input character a. Nonterminal A attempts an expression that corresponds to A. Sequence $e_1 e_2$ attempts two expressions e_1 and e_2 sequentially, backtracking the starting position if either expression fails. Ordered choice e_1 / e_2 first attempts e_1 and then attempts e_2 if e_1 fails. The and-predicate, $\&e$, attempts e without any character consumption. The not-predicate, $!e$, attempts e

e	::=	ϵ	Empty
	|	a	Character
	|	A	Nonterminal
	|	$e\ e$	Sequence
	|	e / e	Ordered choice
	|	$\&e$	And-predicate
	|	$!\,e$	Not-predicate
	|	$\mathsf{bind}_m(v,\ e_p)$	Bind for match
	|	$\mathsf{bind}_e(v,\ e_p)$	Bind for exists
	|	$\mathsf{match}(v,\ e_p)$	Backreference (match)
	|	$\mathsf{exists}(v,\ e_p)$	Backreference (exists)
	|	$\mathsf{scope}(e)$	Scope
e_p	::=	ϵ	Empty
	|	a	Character
	|	A_p	Nonterminal
	|	$e_p\ e_p$	Sequence
	|	e_p / e_p	Ordered choice
	|	$!\,e_p$	Not-predicate

Figure 4. Syntax of V-PEG

without any character consumption and fails if e succeeds but succeeds if e fails.

An import extension is an operator that handles environments. Here, let x be a substring matched by e_p. $\mathsf{bind}_m(v, e_p)$ binds x to v, and appends key-value pair (v, x) to E_m. In the same way, $\mathsf{bind}_e(v, e_p)$ appends (v, x) to E_e. $\mathsf{match}(v, e_p)$ checks whether x is equal to $E_m[\![v]\!]$. $\mathsf{exists}(v, e_p)$ checks whether or not there exists (v, x) in E_e. $\mathsf{scope}(e)$ eliminates all key-value pairs bound in e from environments.

Besides, for exists operators, we also enforce the following behavior: Once an exists operator is called, the subsequent call of the exists operator returns the same result as the first call. That is, the result of the exists operator is memoized and returned in the subsequent call of the exists operator. This is because exists operators are mainly used to store the names of declarations such as classes and typedefs. In this case, we can use the idea that "the state modifications always flow forward through the input, but never backward, and previously parsed and memoized expressions need not be invalidated" [12]. Consequently, we can remove E_e from the parsing state of the memoization table without violating the functional nature of packrat parsers.

We consider that any character $.$ represents ordered choices of all single terminals $(a/b/.../c)$ in Σ. Similarly, many convenient notations used in V-PEG such as option and repetitions are treated as syntactic sugar: $e? = e/\varepsilon$, $e\star = A$ *where* $A = e\ A\ /\varepsilon$, and $e+ = ee\star$. Note that the and-predicate $\&e$ is no longer the syntactic sugar of $!\,!\,e$, whereas the and-predicate of PEGs, $\&e_p$, is the syntactic sugar of $!\,!\,e_p$. This is because $\&e$ changes the environments, whereas $!\,e$ does not.

5.2 Semantics

Table 1 shows the semantics of V-PEG. In the semantics, exists are uniquely determined by using a function $\mathbb{I}$ that maps $\mathsf{exists}(v, e_p)$ to a unique natural number. We use $\rightsquigarrow$ to denote a matching relation in V-PEG. Let $G[e, E_m, E_e, U]$ be a V-PEG whose start expression and environments are replaced with e, E_m, and E_e, respectively, in G. Here, $U : (\mathbb{N}, \Sigma^*, \mathcal{E}_m) \rightarrow \Sigma^*$, where $\mathbb{N}$ is a set of natural numbers and $\mathcal{E}_m$ is a set of distinct states of E_m, is a (partial) function that caches the results of exists. Matching relation $G[e, E_m, E_e, U]\ x \rightsquigarrow \langle y, E'_m, E'_e, U' \rangle$ might be read thusly: when the $G[e, E_m, E_e, U]$ parser attempts to match x with E_m, E_e and U, y remains as an unconsumed string and the parser changes E_m, E_e, and U to E'_m, E'_e, and U', respectively. The parsing of G starts with $G[e, [], [], \emptyset]\ x$ where $[]$ denotes an empty environment. We use fail to denote a failure of a matching. That is, $G[e, E_m, E_e, U]\ x \rightsquigarrow \langle \mathsf{fail}, E'_m, E'_e, U' \rangle$ means that the $G[e, E_m, E_e, U]$ parser cannot parse x. The language of V-PEG G is defined as $L(G) = \{xy \mid\ G[e, [], [], \emptyset]\ xy \rightsquigarrow \langle y, E_m, E_e, U \rangle\ \}$.

For simplicity, we use X as an arbitrary result. That is, X is the same as one of the results: $\langle x, E_m, E_e, U \rangle$ or $\langle \mathsf{fail}, E_m, E_e, U \rangle$. In addition, we omit semantics such that e fails, i.e., $G[e, E_m, E_e, U]\ x \rightsquigarrow \langle \mathsf{fail}, E'_m, E'_e, U' \rangle$, for space considerations.

Algorithm 1: $\texttt{parse}(A, i, E_m, E_e)$

```
1: $key = (A, i, \texttt{filter}(E_m))$
2: if $key \in \mathsf{dom}(M_s)$ :
3:   return $M_s[key]$
4: $(j, E'_m, E'_e) = \texttt{parse}(R(A), i, E_m, E_e)$
5: $M_s = M_s[key \mapsto (j, E'_m, E'_e)]$
6: return $(j, E'_m, E'_e)$
```

6 SG-SPP

SG-SPP can be modeled by parse function $\texttt{parse}(e, i, E_m, E_e)$. Function $\texttt{parse}(e, i, E_m, E_e)$ corresponds to $G[e, E_m, E_e, U]$ $x[i..(|x| - 1)]$ given in Table 1.

To show how SG-SPP works, we give the pseudocode of the parse function for parsing nonterminal A. As shown in Alg. 1, the parse function takes expression A, current input position i, and environments E_m and E_e. Its output is the next input position or fail and environments E'_m and E'_e. In Alg. 1, we use M_s to denote the memoization table that is formalized as $M_s : N \times I \times \mathcal{E}_m \rightarrow (I \cup \{\mathsf{fail}\}) \times \mathcal{E}_m \times \mathcal{E}_e$ where $\mathcal{E}_m$ and $\mathcal{E}_e$ are the sets of distinct states of E_m and E_e, respectively. $\texttt{filter}$ is a function that takes a list of key-value pairs and returns a list of pairs of the unique keys and the latest values bound to the keys. Pseudocodes of the other operators are omitted because they are the same as those of SPP.

The parsing strategy is almost the same as that of SPP. The difference is in the way of using E_m and E_e in the parsing state of the memoization table. More precisely, for E_m, the parsers only use the latest values bound to variables as the parsing state (line 1), whereas SPP uses the entire environment. For E_e, the parsers exclude E_e from the parsing state (line 1).

Table 1. Matching relation $\rightsquigarrow$ via natural semantics

$$G[\varepsilon, E_m, E_e, U]\, x \rightsquigarrow \langle x, E_m, E_e, U\rangle \qquad (\text{EMPTY.1})$$

$$G[a, E_m, E_e, U]\, ax \rightsquigarrow \langle x, E_m, E_e, U\rangle \qquad (\text{CHAR.1})$$

$$\frac{R(A) = e \qquad G[e, E_m, E_e, U]\, x \rightsquigarrow X}{G[A, E_m, E_e, U]\, x \rightsquigarrow X} \qquad (\text{NT.1})$$

$$\frac{G[e_1, E_m, E_e, U]\, xy \rightsquigarrow \langle y, E'_m, E'_e, U'\rangle \qquad G[e_2, E'_m, E'_e, U']\, y \rightsquigarrow X}{G[e_1 e_2, E_m, E_e, U]\, xy \rightsquigarrow X} \qquad (\text{SEQ.1})$$

$$\frac{G[e_1, E_m, E_e, U]\, xy \rightsquigarrow \langle y, E'_m, E'_e, U'\rangle}{G[e_1 / e_2, E_m, E_e, U]\, xy \rightsquigarrow \langle y, E'_m, E'_e, U'\rangle} \qquad (\text{ORDER.1})$$

$$\frac{G[e_1, E_m, E_e, U]\, x \rightsquigarrow \langle \text{fail}, E'_m, E'_e, U'\rangle \qquad G[e_2, E_m, E_e, U']\, x \rightsquigarrow X}{G[e_1 / e_2, E_m, E_e, U]\, x \rightsquigarrow X} \qquad (\text{ORDER.2})$$

$$\frac{G[e, E_m, E_e, U]\, xy \rightsquigarrow \langle y, E'_m, E'_e, U'\rangle}{G[\&e, E_m, E_e, U]\, xy \rightsquigarrow \langle xy, E'_m, E'_e, U'\rangle} \qquad (\text{AND.1})$$

$$\frac{G[e, E_m, E_e, U]\, x \rightsquigarrow \langle \text{fail}, E'_m, E'_e, U'\rangle}{G[!\,e, E_m, E_e, U]\, x \rightsquigarrow \langle x, E_m, E_e, U'\rangle} \qquad (\text{NOT.1})$$

$$\frac{G[e_p, E_m, E_e, U]\, xy \rightsquigarrow \langle y, E_m, E_e, U\rangle}{G[\text{bind}_m(v, e_p), E_m, E_e, U]\, xy \rightsquigarrow \langle y, E_m :: (v, x), E_e, U\rangle} \qquad (\text{BINDM.1})$$

$$\frac{G[e_p, E_m, E_e, U]\, xy \rightsquigarrow \langle y, E_m, E_e, U\rangle}{G[\text{bind}_e(v, e_p), E_m, E_e, U]\, xy \rightsquigarrow \langle y, E_m, E_e :: (v, x), U\rangle} \qquad (\text{BINDE.1})$$

$$\frac{G[e_p, E_m, E_e, U]\, xy \rightsquigarrow \langle y, E_m, E_e, U\rangle \qquad x = E_m[\![v]\!]}{G[\text{match}(v, e_p), E_m, E_e, U]\, xy \rightsquigarrow \langle y, E_m, E_e, U\rangle} \qquad (\text{MATCH.1})$$

$$\frac{\begin{array}{c} t = (\mathbb{I}(\text{exists}(v, e_p)), x, E_m) \quad t \in \text{dom}(U) \\ G[e_p, E_m, E_e, U]\, x \rightsquigarrow \langle U(t), E_m, E_e, U\rangle \end{array}}{G[\text{exists}(v, e_p), E_m, E_e, U]\, x \rightsquigarrow \langle U(t), E_m, E_e, U\rangle} \qquad (\text{EXISTS.1})$$

$$\frac{\begin{array}{c} t = (\mathbb{I}(\text{exists}(v, e_p)), xy, E_m) \quad t \notin \text{dom}(U) \\ G[e_p, E_m, E_e, U]\, xy \rightsquigarrow \langle y, E_m, E_e, U\rangle \quad \mathit{in}(E_e, (v, x)) \end{array}}{G[\text{exists}(v, e_p), E_m, E_e, U]\, xy \rightsquigarrow \langle y, E_m, E_e, U[t \mapsto y]\rangle} \qquad (\text{EXISTS.2})$$

$$\frac{\begin{array}{c} t = (\mathbb{I}(\text{exists}(v, e_p)), xy, E_m) \quad t \notin \text{dom}(U) \\ G[e_p, E_m, E_e, U]\, xy \rightsquigarrow \langle y, E_m, E_e, U\rangle \quad \overline{\mathit{in}(E_e, (v, x))} \end{array}}{G[\text{exists}(v, e_p), E_m, E_e, U]\, xy \rightsquigarrow \langle \text{fail}, E_m, E_e, U[t \mapsto \text{fail}]\rangle} \qquad (\text{EXISTS.3})$$

$$\frac{G[e, E_m, E_e, U]\, xy \rightsquigarrow \langle y, E'_m, E'_e, U'\rangle}{G[\text{scope}(e), E_m, E_e, U]\, xy \rightsquigarrow \langle y, E_m, E_e, U'\rangle} \qquad (\text{SCOPE.1})$$

Example 1. Consider the following V-PEG:

$$\begin{array}{lcl} A & = & \text{bind}_m(v_1, a)\ \text{bind}_m(v_2, b)\ \text{bind}_m(v_1, c)\ \text{bind}_e(v_3, d)\ B \\ B & = & . \end{array}$$

We assume that the start expression is $R(A)$ and the input string is $abcde$. When the parser attempts to match the input string, it begins with the start expression $R(A)$. The parser first attempts to match the three bind_ms and the bind_e. Here, it succeeds after consuming the first four characters, i.e., $abcd$, from the input string and the environments E_m and E_e become $[(v_1, a), (v_2, b), (v_1, c)]$ and $[(v_3, d)]$, respectively.

Next, the parser applies B rule that matches any character. The parser first checks whether the parser already knows the parsing result against the current parsing state (lines 1-3). Here, the current parsing state key is $(B, 4, [(v_1, c), (v_2, b)])$ (line 1). In this case, the parser does not know the parsing result, that is, M_s does not have key as the key (line 2). Thus, the parser attempts to match $R(B)$, which succeeds because $R(B)$ matches any character and it matches the next character e in the input string, and returns the parsing result (5, $[(v_1, a), (v_2, b), (v_1, c)]$, $[(v_3, d)]$) (line 4). Finally, the parser memoizes the parsing result (line 5), and returns it (line 6). As a result, the parser succeeds on the input string $abcde$ after consuming the entire input string.

The time and space complexities of the SG-SPP are both $O(n^{1+|V|})$. We can prove the complexities in the same way as that for packrat parsing, but we omit them due to space constraints. For full details see [2].

7 Conditional Memoization – Intuition

We now explain our key idea of conditional memoization. The key insight is that the parsing results are always the same if the parser tries to match the same nonterminal at the same position with the environments such that the results of the operators affected by the environments are also the same. For example, consider the following rule. We assume

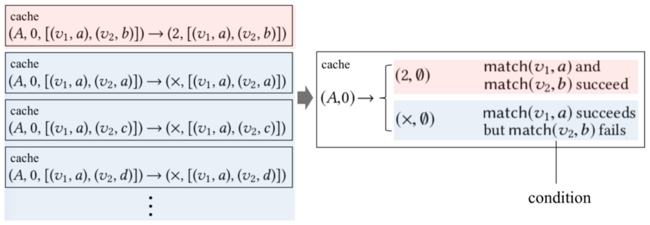
cache
$(A, 0, [(v_1, a), (v_2, b)]) \rightarrow (2, [(v_1, a), (v_2, b)])$
cache
$(A, 0, [(v_1, a), (v_2, a)]) \rightarrow (\times, [(v_1, a), (v_2, a)])$
cache
$(A, 0, [(v_1, a), (v_2, c)]) \rightarrow (\times, [(v_1, a), (v_2, c)])$
cache
$(A, 0, [(v_1, a), (v_2, d)]) \rightarrow (\times, [(v_1, a), (v_2, d)])$
cache
$(A,0) \rightarrow$
$(2, \emptyset)$
match(v_1, a) and match(v_2, b) succeed
$(\times, \emptyset)$
match(v_1, a) succeeds but match(v_2, b) fails
condition

Figure 5. Memoization for parse$(A, 0)$. (Left) SG-SPP and (right) CM-SPP. Environment E_e is omitted for simplicity.

that the rule is a part of a grammar.

$$A \quad = \quad \text{match}(v_1, a)\ \text{match}(v_2, b)$$

Moreover, we assume that the input string is abc, the input position is 0, and E_m is $[(v_1, a), (v_2, a)]$. In this case, the parsing result of A is fail since match(v_1, a) succeeds but match(v_2, b) fails. Then, we consider the situation in which the grammar, input string, and input position are the same as above but the environments are different. Let E_m be $[(v_1, a), (v_2, c)]$. In this case, the parsing result is also fail since the results of the match operators are the same as in the above case, i.e., match(v_1, a) succeeds but match(v_2, b) fails.

We leverage this observation to avoid incorporating the environments in a parsing state. That is, we use the results of match instead of the environments. To give the reader some intuition, Fig. 5 shows the difference in the memoization between SG-SPP and CM-SPP.

We define the results of match as a *condition*, and incorporate this condition into the memoization table. Informally, we define a condition as given hereafter. Let $f_1(v_1, e_1)$, $f_2(v_2, e_2)$, ..., $f_m(v_m, e_m)$ (f_i = match) be match operators that are used during the parsing. Let $b_1, b_2, ..., b_m$ ($b_i \in \{true, false\}$) be results of the match operators. Here, we assume that if $f_i(v_i, e_i)$ succeeds, then b_i is $true$; otherwise, b_i is $false$.

Then, a condition takes the form $f_i(v_i, x_i) \to \mathfrak{b}_i$ where x_i is a substring matched to e_i. This can be read as follows: if $\mathfrak{b}_i$ is *true*, then the operator $f_i(v_i, e_i)$ succeeds; otherwise, $f_i(v_i, e_i)$ fails. By using the conditions, we redefine the memoization as follows: when the parser tries to match the same nonterminal at the same position, the parsing result is also the same if the following conditions hold:

$$\forall i \in \{1, 2, .., m\}.\ f_i(v_i, x_i) \to \mathfrak{b}_i.$$

The aforementioned conditions, however, have a problem that will lead to a contradiction. The problem occurs if there exist match operators such that the results of the match operators are determined regardless of the environments. Let us give a simple example of the problem.

$$A \quad = \quad \mathsf{match}(v_1, a)\ \mathsf{bind}_m(v_1, a)\ \mathsf{match}(v_1, a)$$

In this example, match appears twice. The result of first $\mathsf{match}(v_1, a)$ depends on E_m, whereas the result of the second $\mathsf{match}(v_1, a)$ does not because the result of second one is determined by $\mathsf{bind}_m(v_1, a)$. As a result, we may obtain conditions that contain contradictory terms: $\mathsf{match}(v_1, a) \to$ *true* and $\mathsf{match}(v_1, a) \to$ *false*. Thus, in order to avoid this contradiction, we must ignore the operators in which the results are determined regardless of the environments in making conditions. We call conditions that lead to contradictory terms *deterministic condition* (or simply *deterministic*), and avoid incorporating deterministic conditions into conditions for the memoization.

We summarize the definitions for the conditions described in this section hereafter.

Definition 2. *A condition $\mathfrak{c}$ is 4-tuple $(f, v, x, \mathfrak{b})$ where f is a* match*, v is a variable, x is a string, and $\mathfrak{b}$ is a boolean value.*

For simplicity, we write $\mathfrak{c}$ as follows. Here, *true* is treated as a dummy condition to initialize the condition for the memoization and arbitrary environments meet the condition. In addition, we write $\mathfrak{c}_1 \land \mathfrak{c}_2 \land \ldots \land \mathfrak{c}_m$ as a list of conditions.

$$\begin{array}{lll} \mathfrak{c} ::= & \mathsf{match}(v, x) \to \mathfrak{b} & \text{A condition for match} \\ \quad\ \mid & \textit{true} & \text{A dummy condition} \end{array}$$

Definition 3. *We say that an environments E_m meets conditions $\mathfrak{c}_1 \land \ldots \land \mathfrak{c}_m$ if E_m meets the following conditions:*

$$\forall i \in \{1, 2, \ldots, m\}. \begin{cases} E_m[\![v]\!] = x & (\ \mathfrak{c}_i = \mathsf{match}(v, x) \to \textit{true}\ ) \\ E_m[\![v]\!] \neq x & (\ \mathfrak{c}_i = \mathsf{match}(v, x) \to \textit{false}\ ) \end{cases}$$

Definition 4. *A parsing result is a 4-tuple ($\mathfrak{c}_1 \land \ldots \land \mathfrak{c}_m$, i, ΔE_m, ΔE_e) where $\mathfrak{c}_1 \land \ldots \land \mathfrak{c}_m$ is a list of conditions, i is the next input position or* fail*, ΔE_m is a list of key-value pairs for* match*, and ΔE_e is a list of key-value pairs for* exists.

For example, a parsing result ($\mathsf{match}(v_1, a) \to$*true*, 1, $[(v_2, b)]$, $[(v_3, c)]$) might be read thusly: if E_m meets $\mathsf{match}(v_1, b) \to$ *true*, then the next input position is 1, E_m becomes $E_m :: (v_2, b)$, and E_e becomes $E_e :: (v_3, c)$.

Definition 5. *Let Φ be a function that takes a variable v and a string x. Here, we assume that the key-value pair (v, x) is in E_m. Then, Φ returns the number of executions of* bind_m *before appending the (v, x) to E_m. Note that the number of executions includes the number of indirect binds by the memoization. In addition, let $num_{m,(A,i)}$ be the numbers of executions of* bind_m *at call* $\mathsf{parse}(A, i)$.

Condition $\mathsf{match}(v, x) \to \mathfrak{b}$ *that occurs at call* $\mathsf{parse}(A, i)$ *is deterministic if $(v,\_)$ is in E_m and $num_{m,(A,i)} < \Phi(v, E_m[\![v]\!])$.*

Let e_{p1} and e_{p2} be parsing expressions that match a substring x. Intuitively, $(v,\_)$ is in E_m and $num_{m,(A,i)} < \Phi(v, x)$ for $\mathsf{match}(v, e_{p1})$ means that there exists $\mathsf{bind}_m(v, e_{p2})$ between calling of $\mathsf{parse}(i, A)$ and the execution of $\mathsf{match}(v, e_{p1})$. For example, let us consider the following grammar.

$$\begin{array}{lll} A & = & \mathsf{bind}_m(v_1,a)\ B \\ B & = & \mathsf{bind}_m(v_2,b)\ \mathsf{match}(v_2, b)\ \mathsf{match}(v_1, a) \end{array}$$

We assume that the start expression is A and input string *abba*. When the parser attempts to match B at input position 1, $num_{m,(B,1)}$ is 1. In this case, $\mathsf{match}(v_2, b) \to$ *true* is deterministic for parsing state $(B, 1)$ since $num_{m,(B,1)} < \Phi(v_2, b)$, i.e., 1 < 2. On the other hand, $\mathsf{match}(v_1, a) \to$ *true* is not deterministic since $num_{m,(B,1)} = \Phi(v_1, a)$, i.e., 1=1. Then, after the parser returns to A, $\mathsf{match}(v_1, a) \to$ *true* becomes deterministic for parsing state $(A, 0)$ since $num_{m,(A,0)} < \Phi(v_1, a)$, i.e., 0 < 1. Hence, we should avoid incorporating $\mathsf{match}(v_2, b) \to$ *true* and $\mathsf{match}(v_1, a) \to$ *true* into the condition for memoization of $(B, 1)$ and $(A, 0)$, respectively.

Finally, we revisit the counter-example in §3. In the example, the condition becomes *true*, i.e., the parsing result is totally independent of the environments. This is because the match operator fails to match `Name`, and the failure only depends on the input position. Consequently, the parser for the HTML grammar can run in the same way as the packrat parser for PEGs.

8 CM-SPP

8.1 Algorithm Overview

We now give an overview of CM-SPP. In the same way as in §6, we give the pseudocode of the parse functions for parsing nonterminal A. The pseudocode is shown in Alg. 2.

Global variables. In addition to the same global variables as SG-SPP, CM-SPP has three more global variables: *c_list*, num_m, and *num_list*. *c_list* is a list that stores the lists of conditions for each parsing state during the parsing. When the parser finishes call $\mathsf{parse}(A, i, E_m, E_e)$, the last element of *c_list* is the conditions for parsing state (A, i) (line 14). num_m corresponds to $num_{m,(A,i)}$ described in §7. Initially, num_m is 0. *num_list* is a list that stores integers that correspond to num_ms for each parsing state during the parsing.

Checking the memoization table. The first step of CM-SPP is to check whether or not the parsing result already

Algorithm 2: parse(A, i, E_m, E_e)

```
1:  $key = (A, i)$
2:  $p$ = checkMemo($key$)
3:  if $p$ is not $empty$ :
4:      $num_m = num_m + |p.\Delta E_m|$
5:      propagate($p.cs$)
6:      return $(p.i, E_m :: p.\Delta E_m, E_e :: p.\Delta E_e)$
7:  $c\_list = c\_list :: true$
8:  $num\_list = num\_list :: num_m$
9:  $(j, E'_m, E'_e)$ = parse$(R(A), i, E_m, E_e)$
10: if $j$ is fail :
11:     $\Delta E_m = [], \Delta E_e = []$
12: else :
13:     $\Delta E_m = E'_m[|E_m|..|E'_m| - 1], \Delta E_e = E'_e[|E_e|..|E'_e| - 1]$
14: $\mathfrak{c}_1 \wedge \ldots \wedge \mathfrak{c}_m = c\_list[|c\_list|-1]$
15: $c\_list = c\_list[0..|c\_list|-2]$
16: $num\_list = num\_list[0..|num\_list|-2]$
17: $M_C =$
    $M_C[key \mapsto M_C[key] \cup \{(\mathfrak{c}_1 \wedge \ldots \wedge \mathfrak{c}_m, j, \Delta E_m, \Delta E_e)\}]$
18: return $(j, E'_m, E'_e)$
```

exists in the memoization table by using a checkMemo function(line 2). Here, we define the memoization table as $M_C : N \times I \rightarrow \mathcal{P}$ where $\mathcal{P}$ is a set of distinct parsing results. The checkMemo function returns a parsing result p in M_C such that the parsing state *key* meets the conditions if it exists, and otherwise returns *empty*. If there exists p in M_C, the parser updates the states and returns the parsing result. The parser first increases num_m by the size of ΔE_m due to the indirect binds (line 4). Then, the parser updates the conditions in *c_list* (line 5), and returns the parsing result (line 6). The details of propagate at line 5 are described in §8.2. This process for checking the memoization table is the same as that for packrat parsing. However, the approach is completely different. Specifically, although the packrat parsing simply returns a value of the memoization table, CM-SPP returns the value iff there exists a parsing result that meets the conditions in the current environments.

Example 2. Let us consider the following grammar.

$$A \quad = \quad \text{bind}_m(v, .)\ \&B?\ \&\text{bind}_m(v, .)\ B$$
$$B \quad = \quad \text{match}(v, b)$$

We assume that the start expression is A and the input string is *ab*. When the parser tries to match the second B, the input position is 1 and E_m is [(v,a), (v,b)]. The parsing result in the memoization table for B at position 1 is ($true \wedge \text{match}(v, b) \rightarrow false$, fail, $\emptyset$, $\emptyset$). The parser evaluates the condition before starting to match the second B. However, in this case, E_m does not meet the condition since the result of match(v, b) is *true*. As a result, the parser starts to match the second B.

Construct parsing results. After checking the memoization table, CM-SPP begins preparing for the construction of conditions (lines 7-8). In CM-SPP, a condition is constructed for each nonterminal during the parsing. The condition is stored in *c_list*, and initially the condition is *true* (line 7). Here, we represent the appending of *true* to *c_list* and num_m to *num_list* as *c_list* = *c_list* :: *true* and *num_list* = *num_list* :: num_m in the same manner as the appending of environments. After preparation, CM-SPP attempts to match the expression $R(A)$ (line 9). During the parsing $R(A)$, the conditions in *c_list* are modified incrementally. Then, we construct the parsing result for the memoization (lines 10-14). The parsing result comprises conditions (line 14), the next input position or fail (line 9), and the difference between the environments before and after parsing $R(A)$ (lines 10-13). CM-SPP caches the parsing results into memoization table M_C in the same way as that for packrat parsing (line 17). Finally, our algorithm returns the parsing result (line 18).

Algorithm 3: propagate$(\mathfrak{c}_1 \wedge \ldots \wedge \mathfrak{c}_m)$

```
1:  for $i$ in 2 .. $m$ : ▷ Here, we assume $\mathfrak{c}_1$ = $true$.
2:    ptr = |$c\_list$| - 1
3:    while ptr ≥ 0 :
4:        if $E_m[\![\mathfrak{c}_i.v]\!] \neq \bullet$ and $num\_list$[ptr] <
    $\Phi(\mathfrak{c}_i.v, E_m[\![\mathfrak{c}_i.v]\!])$ :
5:            break
6:        if $c\_list$[ptr] has match$(\mathfrak{c}_i.v, \_) \rightarrow true$ :
7:            ptr = ptr - 1
8:            continue
9:        if $\mathfrak{c}_i.\mathfrak{b}$ is $true$ :
10:           eliminate all conditions such as match($v$,_) →
    $false$ from $c\_list$[ptr]
11:       $c\_list$[ptr] = $c\_list$[ptr] ∧ $\mathfrak{c}_i$
12:       ptr = ptr - 1
```

Example 3. We consider the same grammar shown in Example 2. We assume the input string is *ab* and consider matching the second B. After matching the second B, the input position is 2 and the environments are the same as before matching the second B. In addition, condition match(v,b) → *true* is constructed during the parsing. Hence, the parsing result for parsing state (B, 1) is (*true* ∧ match(v,b) → *true*, 2, $\emptyset$, $\emptyset$).

8.2 Construct Conditions

We now explain in more detail how CM-SPP constructs conditions by incrementally modifying the conditions in *c_list*. The conditions are modified by using the propagate function.

Alg. 3 represents the pseudocode for the propagate function. The propagate function takes a list of conditions $\mathfrak{c}_1 \wedge \ldots \wedge \mathfrak{c}_m$. The objective of the propagate function is to propagate $\mathfrak{c}_1 \wedge \ldots \wedge \mathfrak{c}_m$ to the conditions in *c_list* without constructing deterministic conditions. Alg. 3 iteratively confirms that condition $\mathfrak{c}_i$ should be incorporated into the conditions in *c_list* (line 1). To verify this, the algorithm first checks whether or not $\mathfrak{c}_i$ is deterministic or not (line 4). Then, the algorithm checks whether or not the condition *c_list*[ptr] has $\mathtt{match}(\mathfrak{c}_i.v, \_) \rightarrow \mathit{true}$ because, if so, the algorithm does not add $\mathfrak{c}_i$ to *c_list*[ptr] for efficiency reasons (line 6). Moreover, if the result of condition $\mathfrak{c}_i.\mathfrak{b}$ is *true*, then the algorithm eliminates all conditions such as $\mathtt{match}(v, \_) \rightarrow \mathit{false}$ before adding $\mathfrak{c}_i$ to *c_list*[ptr] (lines 9-10). Although we can simply add $\mathfrak{c}_i$ to *c_list*[ptr], we replace them for efficiency reasons. Finally, the algorithm adds $\mathfrak{c}_i$ to *c_list*[ptr] (line 15).

There are two operators that call the propagate function: a nonterminal and match. As shown in Alg. 2, the parse function for a nonterminal calls the propagate function at line 6. The parse function for match calls the propagate function before returning the parsing result.

To understand how the propagate function works, let us consider the following grammar.

$$
\begin{aligned}
A &= \mathtt{bind}_m(v_1, a)\ B \\
B &= \mathtt{bind}_m(v_2, b)\ C \\
C &= \mathtt{bind}_m(v_3, c)\ \mathtt{match}(v_2, b)\ \mathtt{match}(v_1, a)\ \mathtt{match}(v_3, c)
\end{aligned}
$$

We assume that the start expression is A, the input string is *abcbac*, and the input position is 0. When the parser for the grammar tries to match C, *c_list* = [*true*, *true*, *true*]. The conditions in *c_list* correspond to the initial conditions for parsing state $(A, 0)$, $(B, 1)$, and $(C, 2)$ in order. In this grammar, there are three points to using the propagate function. First, after parsing $\mathtt{match}(v_2, b)$, the parser must call the propagate function to propagate condition $\mathtt{match}(v_2, b) \rightarrow \mathit{true}$ to the conditions in *c_list*. Let $\mathfrak{c}_1$ be $\mathtt{match}(v_2, b) \rightarrow \mathit{true}$. Then, $\mathfrak{c}_1$ is not deterministic for parsing state $(C, 2)$ but is for parsing state $(B, 1)$. Hence, *c_list* becomes [true, true, $\mathit{true} \land \mathfrak{c}_1$]. Second, after parsing $\mathtt{match}(v_1, a)$, the parser calls propagate with $\mathtt{match}(v_1, a) \rightarrow \mathit{true}$. Let $\mathfrak{c}_2$ be $\mathtt{match}(v_1, a) \rightarrow \mathit{true}$. Then, $\mathfrak{c}_2$ is not deterministic for $(C, 2)$ and $(B, 1)$ but is for $(A, 0)$. Hence, *c_list* becomes [true, $\mathit{true} \land \mathfrak{c}_2$, $\mathit{true} \land \mathfrak{c}_1 \land \mathfrak{c}_2$]. Third, after parsing $\mathtt{match}(v_3, c)$, the parser calls propagate with $\mathtt{match}(v_3, c) \rightarrow \mathit{true}$. Let $\mathfrak{c}_3$ be $\mathtt{match}(v_3, c) \rightarrow \mathit{true}$. In this case, $\mathfrak{c}_3$ is deterministic for $(C, 2)$. Thus, propagate does not modify *c_list*.

8.3 Complexity

The space and time complexities of CM-SPP are $O(n^{1+|V|})$ and $O(n^{3+2|V|})$, respectively. In addition, CM-SPP runs in linear time and space against malicious inputs described in §4.2. For full details see [2].

9 Evaluation

9.1 Experiment Setup

The experiments compare the running time and space consumption across two grammars and three parsing algorithms and measured the performance. To conduct the experiments, we implement a parser generator based on grammars written in V-PEGs that generates the parsers on the basis of the above three parsing algorithms. In addition, we use two grammars: (1) HTML — which is based on an ANTLR4 grammar[4] and has match operators and (2) Java — which is based on a Nez grammar[5] and has exists operators. To measure the performances of the algorithms themselves, our implementation does not incorporate the optimizations for PEG parsers such as [12].

The experiments are conducted on a MacBook Pro with a 3.5 GHz Intel i7 and 16 GB of RAM running macOS 10.13.

9.2 Comparing Running time and Space Consumption

To highlight the effect of conditional memoization on both running time and space consumption, we first compare the performance of CM-SPP with that of the other algorithms. Fig. 6 shows the results.

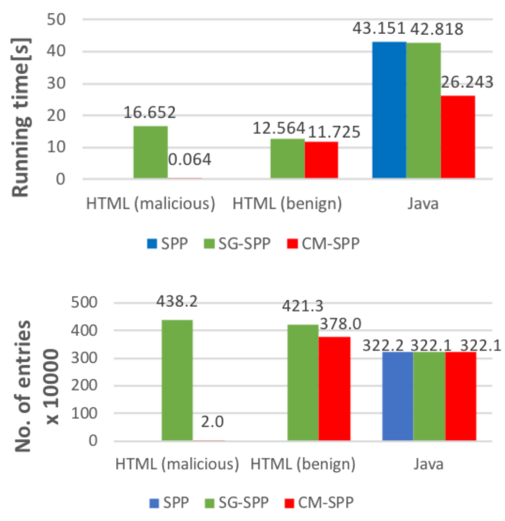

Figure 6. Comparison results. (Top) running time and (bottom) space consumption.

In Fig. 6, space consumption denotes the number of entries cached in the memoization table. Since CM-SPP may have more than one parsing results for entry, we define the space consumption as the number of conditions in the memoization table. We omitted the results of SPP for HTML (malicious) and HTML (benign) due to the exponential behavior; in 4 cases a timeout occurred (set to one hour).

In this experiment, we use two types of inputs for the HTML parsers: *malicious* and *benign*. The malicious input comprises 500 opening tags and is used to evaluate the performance of the parsers for the malicious input that may

[4]https://github.com/antlr/grammars-v4/blob/master/html/HTMLParser.g4
[5]https://github.com/nez-peg/nez-grammar/blob/master/java8.nez

lead to exponential behavior. On the other hand, the benign input is used to evaluate the performance of the parsers for normal input. For the benign inputs, we use a sampling of 43 HTML files taken from Alexa Top 500 Global Sites [1]. For inputs for the Java parsers, we use a sampling of 47 Java source files taken from various open source repositories. All input files were loaded into random access memory (RAM) before parsing.

For this experiment, the parsers based on CM-SPP outperform the other parsers. In particular, in the case of the HTML grammar with the malicious input, the parser based on CM-SPP improves the running time and space consumption by 260x and 217x, respectively, compared with the HTML parser based on SG-SPP. In addition, in the case of the Java grammar, the parser based on CM-SPP improves the running time by 1.6x compared with the Java parser based on SG-SPP. The difference of the performance improvement in the Java parsers comes from the size of entries in a memoization table. As shown in line 5 of Alg. 1, SG-SPP caches full environments, i.e., E_m and E_e, whereas, as shown in line 17 of Alg. 2, CM-SPP caches only the differences of environments, i.e., ΔE_m and ΔE_e. The difference of the sizes improves the cost to copy the environments into the memoization table.

In addition to this experiment, we have also evaluated the performance of the C parser; the results—not shown, due to space limitation—are similar to the ones of the Java parser, i.e., the parser based on CM-SPP outperforms the parser based on the other algorithms. For interested readers, the parsers are given at [2].

10 Related Work

There has been much work on parsing algorithms for extensions that allow some context-sensitive features to be expressed [4][6][13][18][19][22][23][24]. Here, we focus on existing works that are most closely related to ours.

The idea of SPP was first introduced by Ford [8][9]. In his work, he incorporated the desired state into the packrat parser. Although the extension enables us to parse context-sensitive syntaxes, it does not guarantee polynomial time.

Laurent and Mens [16] presented principled stateful parsing and implemented the algorithm as a parser combinator library named *Autumn*. To recognize the context-sensitive features, Autumn allows the generation of a copy of a parsing state and the state to be returned during the parsing. Although Autumn allows us to handle some context-sensitive features, it requires exponential complexity in the worst case.

Grimm [12] presented *Rats!*, a parser generator constructed on PEGs that has a global state to support some context-sensitive grammars. He also developed a technique for managing the global state in the parsing to recognize the syntax of typedef names in C/C++, and our formalism for exists operators is based on this idea.

Kuramitsu [15] proposed SPEGs, an extension of PEGs for recognizing context-sensitive features in real-world grammars, and SPP for SPEGs. He implemented his algorithm as a parser generator named Nez. As described in §3, his algorithm incurs exponential running time in the worst case.

11 Conclusion

We demonstrated that existing SPP incurs exponential running time even on real-world grammars. To overcome the exponential behavior, we proposed a new grammar, V-PEG, and two parsing algorithms for the grammar, SG-SPP and CM-SPP. Our proposal guarantees polynomial running time and significantly improves both the running time and space consumption. We evaluated our algorithms by comparing them to existing SPP. The evaluation showed that our algorithms significantly outperform an existing SPP in terms of both running time and space consumption. In particular, in the case of HTML grammar with the malicious input, CM-SPP improved the running time and space consumption by 260x and 217x, respectively.

References

[1] Alexa top 500 global sites. URL https://www.alexa.com/topsites.

[2] Supplementary material for this paper is available in the following github repository. URL: https://github.com/NariyoshiChida/cc2020.

[3] Adams, M. D. Principled parsing for indentation-sensitive languages: Revisiting landin's offside rule. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2013), POPL '13, ACM, pp. 511–522.

[4] Adams, M. D., and Ağacan, O. S. Indentation-sensitive parsing for parsec. *SIGPLAN Not. 49*, 12 (Sept. 2014), 121–132.

[5] Adar, W. Regular expression denial of service - redos, 2017. URL https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.

[6] Aho, A. V. Indexed grammars&mdash;an extension of context-free grammars. *J. ACM 15*, 4 (Oct. 1968), 647–671.

[7] Davis, J. C., Coghlan, C. A., Servant, F., and Lee, D. The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2018), ESEC/FSE 2018, ACM, pp. 246–256.

[8] Ford, B. Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2002), ICFP '02, ACM, pp. 36–47.

[9] Ford, B. A practical linear-time algorithm with backtracking. In *Master's thesis* (2002), MIT.

[10] Ford, B. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2004), POPL '04, ACM, pp. 111–122.

[11] Gietzen, J. Pegasus: Super-easy peg parsing for .net. URL http://otac0n.com/Pegasus/.

[12] Grimm, R. Better extensibility through modular syntax. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 38–51.

[13] Jim, T., Mandelbaum, Y., and Walker, D. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages (New York, NY, USA, 2010), POPL '10, ACM, pp. 417–430.

[14] Kuramitsu, K. Nez : Open grammar language and tools. URL http://nez-peg.github.io/.

[15] Kuramitsu, K. A symbol-based extension of parsing expression grammars and context-sensitive packrat parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (New York, NY, USA, 2017), SLE 2017, ACM, pp. 26–37.

[16] Laurent, N., and Mens, K. Taming context-sensitive languages with principled stateful parsing. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering* (New York, NY, USA, 2016), SLE 2016, ACM, pp. 15–27.

[17] Levine, J. R. *lex & yacc*. O'Reilly, 1992.

[18] Mehlhorn, K. Parsing macro grammars top down. *Information and Control 40*, 2 (1979), 123 – 143.

[19] Parr, T., Harwell, S., and Fisher, K. Adaptive ll(*) parsing: The power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2014), OOPSLA '14, ACM, pp. 579–598.

[20] Shen, Y., Jiang, Y., Xu, C., Yu, P., Ma, X., and Lu, J. Rescue: Crafting regular expression dos attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (New York, NY, USA, 2018), ASE 2018, ACM, pp. 225–235.

[21] Späth, C., Mainka, C., Mladenov, V., and Schwenk, J. Sok: XML parser vulnerabilities. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)* (Austin, TX, Aug. 2016), USENIX Association.

[22] Thurston, A. D., and Cordy, J. R. A backtracking lr algorithm for parsing ambiguous context-dependent languages. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research* (Riverton, NJ, USA, 2006), CASCON '06, IBM Corp.

[23] van Eijck, J. Sequentially indexed grammars. *J. Log. and Comput. 18*, 2 (Apr. 2008), 205–228.

[24] Warth, A., and Piumarta, I. Ometa: An object-oriented language for pattern matching. In *Proceedings of the 2007 Symposium on Dynamic Languages* (New York, NY, USA, 2007), DLS '07, ACM, pp. 11–19.