Supplementary material

Nariyoshi Chida
nariyoshichidamm@gmail.com

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages*;

ACM Reference Format:
Nariyoshi Chida. 2020. Supplementary material. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

A Preliminaries (Full version)

A.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 $M : N \times I \to (I \cup \{\mathsf{fail}\})$ 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 $I \cup \{\mathsf{fail}\}$ 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. $M(A, i) = \mathsf{fail}$ denotes that A cannot match any substring starting at i.

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

```
XML  =  '<' Name '>' XML '< /' Name '>'
     |  '<' Name '/>'
     |  ''
Name =  [a − zA − Z]+
```

When the packrat parser attempts to match the input string *<a/>*, it begins with the first alternative, i.e., '<' `Name` '>' `XML` '< /' `Name` '>' . The parser first attempts to match a '<' character. Here, it succeeds after consuming the first character, i.e., *<*, from the input string. Next, the parser applies `Name` rule that matches a sequence of one or more alphabet letters. It also succeeds after consuming the second character, i.e., *a*, from the input string. Here, the result for applying `Name` rule at position 1 is memoized. Then, the parser attempts to match a '>' character, which fails because the next character in the input string is */*. This causes the parser to backtrack to position 0, which is where the current alternative started.

Then, the parser tries the second alternative, i.e., '<' `Name` '/>'. The parser attempts to match a '<' character, which succeeds. Then, the parser applies `Name` at position 1. At this point, the parser already knows that `Name` succeeds at position 1 after consuming a character because the result of applying `Name` at position 1 is memoized on the first attempt. Thus, the parser can simply update the position to 2 and attempt to match the next part of the pattern, i.e., '/>'. Finally, the parser attempts to match '/>' and it succeeds after consuming the last two characters */>*. As a result, the parser succeeds on the input string *<a/>* after consuming the entire input string.

A.2 Stateful Packrat Parsing

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

Revisiting the XML grammar in §**??**, the grammar can accept invalid inputs that the opening and closing tags are different. For example, the grammar accepts the invalid input *<a>*. To avoid this, using the extension, we can rewrite the XML grammar as follows:

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

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 $\mathsf{bind}(v, e)$ 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 $\mathsf{match}(v, e)$ 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 $\mathsf{scope}(e)$ operator eliminates all variables bound in e from the list.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

Stateful packrat parsing 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>. In this case, `parse(XML, 0)` returns 7 since the nonterminal `XML` matches the entire string. Note that the position is 0-indexed.

Fig. **??** shows the behavior of stateful packrat parsing for the XML grammar described in this section. We assume that the input string is <a>.

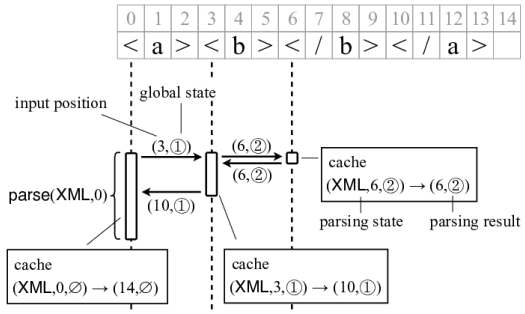

Figure 1. Behavior of stateful packrat parsing for XML grammar

Here, the following are the global states in Fig. **??**.

$$\text{①}\ [(v,a)] \qquad \text{②}\ [(v,a),(v,b)]$$

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, stateful packrat parsing caches three parsing results. In the rest of this paper, we express the behavior of stateful packrat parsing in the same manner.

B Complexity of SG-SPP

We show that the running time and space consumption of the SG-SPP is polynomial. As with packrat parsing, the cost model for SG-SPP is as follows:

$$\text{size of a memoization table} \times \text{cost of a parse function for an operator}$$

First, we show that the size of the memoization table is polynomial. Here, the size of a memoization table is denoted as $|N| \times |I| \times |\mathcal{E}_m|$. $|N|$ and $|I|$ are polynomial since $|N|$ is a constant value and $|I|$ is $O(n)$ where n is the length of the input string. $|\mathcal{E}_m|$ is $O(n^{|V|})$. This is because for each variable $v \in V$, the number of distinct substrings bound to v is $O(n)$ since bind operator $\mathsf{bind}_m(v, e_p)$ takes a parsing expression e_p, and the substrings that match e_p only depend on the input positions. Moreover, the number of variables is $|V|$. Hence, the size of the memoization table is $O(n \times n^{|V|}) = O(n^{1+|V|})$. Note that $|V|$ is constant.

Second, we show that the costs of the parse functions for operators are the same as those of the parse functions for PEGs, i.e., $O(1)$. The costs of the parse functions for the common operators with PEGs are $O(1)$ time if the other operators, i.e., bind_m, bind_e, match, exists, and scope, are also $O(1)$ time. Thus, we focus on the costs of the parse functions for the other operators. To begin with, we consider the costs of the parse functions for bind_m and bind_e. The costs are $O(1)$ time since the parse functions simply append the key-value pairs. Next, we consider the costs of the parse functions for match and exists. The costs of the functions for match depend on the sizes of the environment since the operators search for a key-value pair in the environment. Thus, we show that the sizes of E_m can be constant. match only uses the latest value in E_m for each variable. As a result, we can express E_m as a bijective function and the size is at most $|V|$. The costs of the functions for exists(v, e_p) depend on the time to check whether the key-value pair (v, x), where x is a substring that matched to e_p, is in E_e. This can be achieved in $O(1)$ time by using a hash table. Note that we can calculate the hash value of x in $O(1)$ time by using a rolling hash. Finally, we consider the cost of the parse function for scope. scope(e) restores the environments to the states prior to parse e. The costs of restoring E_m is constant since the size of E_m is constant. In addition, the costs of restoring E_e can be constant by using fully persistent tries[**?**]. Hence, the cost of the parse function for scope is $O(1)$ time.

Consequently, the time complexity of SG-SPP is $O(n^{1+|V|})$, i.e., in polynomial, time. Also, the space complexity is $O(n^{1+|V|})$ due to the size of the memoization table being $O(n^{1+|V|})$.

C Complexity of CM-SPP

First, we show that the worst-case time and space complexities of CM-SPP are polynomial. We assume that the input string length is n.

The space complexity is worst-case $O(n^{1+|V|})$. Since the space complexity depends on the size of the memoization table, we show that the size of the memoization table is $O(n^{1+|V|})$. The size of the memoization table is denoted as $|N| \times |I| \times |\mathcal{P}|$. Here, $\mathcal{P}$ is a set of distinct parsing results, and the size is $O(n^{|V|})$ since a parsing result is constructed by at least one environment and the number of distinct environments is at most $n^{|V|}$ as described in §**??**. Note that we can

express E_m as a bijective function and the size is at most $|V|$. Thus, the space complexity is $O(|N| \times n \times n^{|V|}) = O(n^{1+|V|})$.

The time complexity is worst-case $O(n^{3+2|V|})$. The time complexity is modeled by (size of a memoization table) $\times$ (cost of a parse function for an operator). As mentioned above, the size of a memoization table is $O(n^{1+|V|})$. The most costly parse function is that for a nonterminal due to the functions `checkMemo` and `propagate`. Here, we focus only on the cost of `propagate` since it is higher than the cost of `checkMemo`. The cost of `propagate` is modeled by (the number of iterations at line 1) $\times$ (the number of iterations at line 3) in Alg. 3. The number of iterations at line 1 is $O(|V| \times n)$ due to the maximum number of conditions. The number of iterations at line 3 is $O(n^{1+|V|})$ since the number of iterations depends on the size of *c_list* and *c_list* is the same size as a memoization table in the worst case. Hence, the cost of `propagate` is $O((|V| \times n) \times n^{1+|V|})$. Consequently, the time complexity is $O(n^{1+|V|} \times ((|V| \times n) \times n^{1+|V|})) = O(n^{3+2|V|})$.

Second, we show that CM-SPP runs in linear time and space against malicious inputs described in §4.2. The conditions for all parsing states are *true* against the malicious inputs since the inputs do not lead to the calls match or do not match to e_p of $\mathsf{match}(v, e_p)$. As a result, the size of the memoization table and the cost of the parse function for a nonterminal are $O(n)$ and $O(1)$, respectively. Thus, CM-SPP runs in linear time and space against the malicious inputs.

D Empirical Complexity

We also perform an experiment to show the asymptotic behavior of CM-SPP. To show this, for the grammars and inputs used in §9.2, we plot the running times and the number of entries versus file sizes in Fig. **??**. Note that we scale up and down the malicious input by increasing and decreasing the number of opening tags. We have yet to see nonlinear behavior in practice despite the theoretical worst-case time and space complexities being $O(n^{3+2|V|})$ and $O(n^{1+|V|})$, respectively.

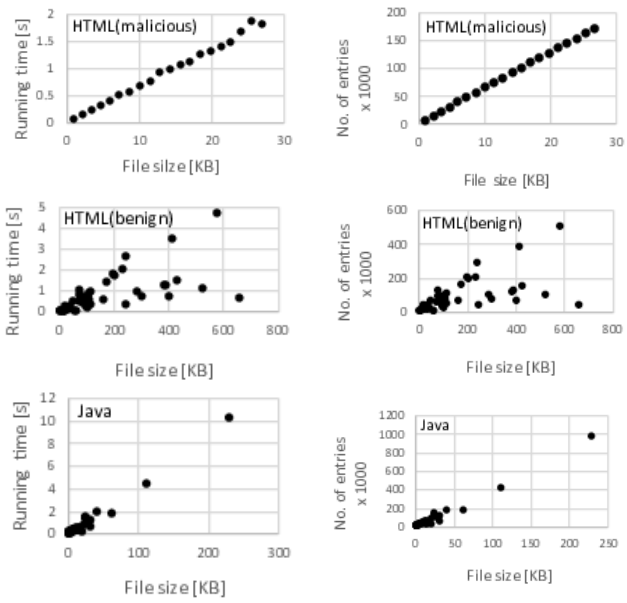

Figure 2. Experimental results for empirical complexity. (Left) time and (right) space.