Implemented C Library for Constructing Packrat Parser with Statically Allocated Memory

Yuta Sugimoto¹,a) Atusi Maeda¹

Received: July 23, 2017, Accepted: September 28, 2017

Abstract: Packrat parsing is a recursive descent parsing method with backtracking and memoization. Parsers based on this method require no separate lexical analyzers, and backtracking enables those parsers to handle a wide range of complex syntactic constructs. Memoization is used to prevent exponential growth of running time, resulting in linear time complexity at the cost of linear space consumption. In this study, we propose CPEG – a library that can be used to write parsers using Packrat parsing in C language. This library enables programmers to describe syntactic rules in an internal domain-specific language (DSL) which, unlike parser combinators, does not require runtime data structures to represent syntax. Syntax rules are just expressed by plain C macros. The runtime routine does not dynamically allocate memory regions for memoization. Instead, statically allocated arrays are used as memoization cache tables. Therefore, programmers can implement practical parsers with CPEG, which does not depend on any specific memory management features, requiring fixed-sized memory (except for input string). To enhance usability, a translator to CPEG from an external DSL is provided, as well as a tuning mechanism to control memoization parameters. Parsing time compared to other systems when parsing JavaScript Object Notation and Java source files are given. The experimental results indicate that the performance of CPEG is competitive with other libraries.

Keywords: Packrat parsing, DSL, C language, parsing

1. Introduction

Packrat parsing [3] is a recursive descent parsing method that can handle a wide variety of grammars using backtracking. To construct parsers with this method, a notation called parsing expression grammars (PEGs) is used to describe grammars. Incorporating memoization enables linear time parsing. Moreover, since these parsers can execute unbounded lookahead, they do not require separate lexical analyzers.

A domain-specific language (DSL) is a computer language designed to express data and/or programs of some specific domains using vocabulary or notations specially chosen for that purpose. Grammar description for parsing, including PEG, can be viewed as a DSL designed for parsing. Internal DSLs use syntactic constructs of a particular general-purpose programming language (host language) to express domain-specific knowledge. Since external DSLs are distinct programming languages, they are not restricted by the syntax of other languages, although they require dedicated parsers.

In this paper, we propose CPEG, a C library which, by describing grammars in PEG-like notation, enables implementation of parsers using packrat parsing. The design goals of CPEG include brief and lightweight implementation of the library and independence from memory-management schemes, which ensures incorporation into a wide range of C programs. The core of CPEG itself does not require a dynamic memory-allocation library such as malloc. However, all the input text must be accessed as a single array, by reading into the buffer or being mapped to the buffer in advance. To restrict total memory usage including input strings, the grammar must be tweaked to prevent backtracking and/or the size of input must be bounded. This is a limitation common to any parser relying on PEG.

2. Background

2.1 Parsing Expression Grammar

Parsing expression grammar [4] is a recognition-based formal grammar. In PEG, grammar rules are expressed in the form N ← e, where N is a nonterminal and e is a parsing expression. Parsing expressions are summarized in Table 1.

Unlike context-free grammars, PEG is characterized by ordered choice operator /, which prioritizes alternatives. In parsing expression e₁ / e₂, e₁ is attempted first, and if it failed, the parser then backtracks and attempts e₂. In this way, ambiguity is eliminated and the parser always derives a unique result.

Also, and-predicate & and not-predicate !e enable unbounded lookahead without consuming input. In LL(k) or LR(k) grammar, the number of lookahead symbols are bounded; thus, parsers using them generally require lexical analyzers to split an input string into tokens. In contrast, the PEG-based parsing algorithm does not require separate lexical analyzers.

2.2 Packrat Parsing

Packrat parsing is a recursive descent parsing method that recognizes input based on grammars described in PEG. Generally,
parsers using this method associate a grammar rule $N \leftarrow e$ with a function that receives an input string and attempts to match the string with $e$. The matching result is represented as the remaining portion of the input string after the match (and optionally semantic value). Choice operator '/' is implemented using backtracking. The result of this function is memoized to enable linear-time parsing.

In the most naive form of implementation, a memoization table occupies a storage size proportional to the product of the input size and number of nonterminals. Many studies have focused on reducing the size of a memoization table (Refs. [7], [11], [13], [14], [19], [20]). The design goal of CPEG is to be simple, predictable, and efficient in implementation. The goal also includes independence from memory-management schemes (e.g., malloc). As described later, since memoization tables do not depend on any memory-management schemes, it is relatively easy to put a bound on memory usage (if grammars and/or input string length are properly limited) or incorporate specific memory-management systems such as garbage collectors.

### 3.1 Defining Grammar Rules

To write parsers using CPEG, users define functions (parsing functions) that correspond to nonterminals that appear on the left-hand side of grammar rules, and write grammar descriptions using parsing expressions.

To define parsing expressions, macros `BEGIN_RULE`, `END_RULE` are used as

```c
BEGIN_RULE(nontermina)) {
  parsing expression;
} END_RULE
```

The `BEGIN_RULE` macro is expanded into a function header and prologue part which include the definition of a memoization table and code to check whether input is already memoized. The `END_RULE` is expanded into an epilogue that stores the results of parsing for the rule into the memoization table.

### 3.2 Writing Down Parsing Expressions Using CPEG

Each element of a parsing expression corresponds to macros in CPEG, as listed in Table 2. Concatenation (SEQ) and ordered choice (ALT) can be simple macros with two arguments, but we implemented them to accept a variable (up to ten) number of arguments for convenience.

The following global variables are used for parsing.

- `input` a pointer that points to the head of input text (which is a null-terminated string array).
- `pos` an integer variable used to indicate the current position in the input. `input[pos]` means character at position `pos` in `input`.
- `val` a semantic value is assigned to this variable after parsing.
- `failed` true if parsing failed, false if succeeded.

The body of the rule surrounded by `BEGIN_RULE` and `END_RULE` may include arbitrary C code if it does not affect the value of the variables above. In the following, we define the macros that correspond to each PEG element. In either case, `pos` does not change when parsing fails.

### Table 1 Parsing expression.

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Parsing expression</th>
<th>Macros</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty string</td>
<td>$\epsilon$</td>
<td>EPSY()</td>
</tr>
<tr>
<td>literal string</td>
<td>$a$</td>
<td>ARY()</td>
</tr>
<tr>
<td>nonterminal</td>
<td>$N$</td>
<td></td>
</tr>
<tr>
<td>concatenation</td>
<td>$e_1e_2$</td>
<td></td>
</tr>
<tr>
<td>ordered choice</td>
<td>$e_1/e_2$</td>
<td></td>
</tr>
<tr>
<td>zero or more repetition</td>
<td>$e^*$</td>
<td></td>
</tr>
<tr>
<td>and-predicate</td>
<td>$&amp;e$</td>
<td>ANDPRE()</td>
</tr>
<tr>
<td>not-predicate</td>
<td>$\sim e$</td>
<td>NOTPRE()</td>
</tr>
</tbody>
</table>

### Table 2 Parsing expression and CPEG macros.

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Parsing expression</th>
<th>Macros</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty string</td>
<td>$\epsilon$</td>
<td>EPSY()</td>
</tr>
<tr>
<td>any character</td>
<td>.</td>
<td>ARY()</td>
</tr>
<tr>
<td>literal character</td>
<td>$a$</td>
<td>CHAR('a')</td>
</tr>
<tr>
<td>literal string</td>
<td>&quot;abc&quot;</td>
<td>STR(&quot;abc&quot;)</td>
</tr>
<tr>
<td>concatenation</td>
<td>$e_1e_2$</td>
<td>SEQ(e1, e2)</td>
</tr>
<tr>
<td>ordered choice</td>
<td>$e_1/e_2$</td>
<td>ALT(e1, e2)</td>
</tr>
<tr>
<td>zero or more</td>
<td>$e^*$</td>
<td>REPEAT(e)</td>
</tr>
<tr>
<td>one or more</td>
<td>$e^+$</td>
<td>REPEAT1(e)</td>
</tr>
<tr>
<td>option</td>
<td>$?e$</td>
<td>OPTION(e)</td>
</tr>
<tr>
<td>and-predicate</td>
<td>$&amp;e$</td>
<td>ANDPRE()</td>
</tr>
<tr>
<td>not-predicate</td>
<td>$\sim e$</td>
<td>NOTPRE(e)</td>
</tr>
</tbody>
</table>
3.3.1 Literals

In principle, a combination of literal characters \( \text{CHAR}(c) \) and concatenation \( \text{SEQ}(c_1, c_2, \ldots) \) is sufficient to express a grammar rule matching a string. Also, \( \text{CHAR}(c) \) and \( \text{ALT}(c_1, c_2, \ldots) \) can express a rule matching any element in a set of characters (charset). However, this is rather cumbersome, so for convenience, we provide notations for literal strings and character sets, as in many other implementations.

### Example

Example of grammar description using STR. A rather complicated combination of SEQ and CHAR can be rewritten to be a concise expression with STR.

**Fig. 1** Example of STR.

```
SEQ(CHAR('a'), CHAR('b'), CHAR('c'));
```

**Fig. 2** Grammar of additive-multiplicative expressions.

```
#include "cpeg.h"
DECLARE_RULES(E, T, P);
cpeg_charset_t digit;
BEGIN_RULE(E)
  ALT(SEQ(T0), CHAR('+'), E()),
  T());
END_RULE
BEGIN_RULE(T)
  ALT(SEQ(P0), CHAR('*'), T()),
  P());
END_RULE
BEGIN_RULE(P)
  ALT(SEQ(CHAR('+'), E(), CHAR('/'))),
  REPEAT(CHARSET(gets()));
END_RULE
```

**Fig. 3** Code to parse additive-multiplicative expression.

```
expression 'c_1' '/c_2' '/\ldots' '/c_n'. In other words, the notation represents an expression that succeeds if and only if an element of the charset \( c_1, c_2, \ldots, c_n \) matches the next input character. Also, notation \( [c_1-c_2] \) represents a parsing expression that succeeds when the next input character is an element of the charset that have a character code greater than or equal to \( c_1 \) and less than or equal to \( c_2 \).

Our CPEG provides macro CHARSET(s), which takes a value of type cpeg_charset_t representing a charset, and denotes a matching to an element of that set. (Since the library assumes the 8-bit character set JIS X0201 for the input text to CPEG, type cpeg_charset_t is implemented as a flag array of size 256.)

The following functions initialize the variables of cpeg_charset_t.

- **cs_simple(set, ch)** Add ch to a charset set.
- **cs_range(set, ch1, ch2)** Add all characters from ch1 to ch2 into a charset set.
- **cs_union(set, base_set)** Add all the characters in the given charset base_set to another charset set.
- **cs_complement(set)** Change a charset set to the complement of the charset set.

**Fig. 2** Grammar of additive-multiplicative expressions.

```
#include "string.h" /* for strlen */
int main(int argc, char argv[1]) {
    input = argv[1];
    cs_range(digit, '0', '9');
    E();
    /* Succeeds if all of the input is parsed */
    return pos = strlen(argv[1]);
}
```
In the first line of Fig. 3, file cpeglib.h, in which macros and data types are defined, is included. After that, rule names are declared using the DECLARE_RULES macro, which is expanded into function prototypes. In the main function, the character set digit is initialized and the start symbol E is called to execute the parsing process.

3.4 Library Implementation Based on Macros

In the design of CPEG, we decided to use C macros for writing a PEG-based parsing expression because (1) macros do not require any dynamically allocated memory to express grammar rules, (2) relatively fast execution will be achieved, and (3) C code for actions and semantic predicates can be inserted in grammars.

When a set of grammar rules is expressed by data structures of the host language, as in parser combinators where objects for terminals and nonterminals are combined with operators to compose complex grammar rules, dynamic memory allocation is required in general to construct grammar rules at runtime. In CPEG, however, grammar rules are described by C macros and expanded into C code at compile time, eliminating the need of dynamic memory allocation for grammar rules.

In addition, since grammar rules are expressed in a control structure in C language instead of data structures interpreted at runtime, the execution speed would be faster.

The core definition for SEQ, concatenation of parsing expressions, is shown in Fig. 4. The STMT macro is an idiom used to define macros consisting of multiple statements [21]. Also, the core part of the ordered choice (ALT) is defined in Fig. 5.

Including arguments e1, e2 of these macros and the macros themselves, all parsing expressions obey the protocol that “if succeeded, set failed to false and assign the parsing result (the semantic value) to val; otherwise, reset pos to the previous value and set failed to true.” Although the expansion introduces a number of assignments to failed and conditional branches on the value of failed, most of them are eliminated by optimization with the C compiler.

An arbitrary expression or statement can be written as e1, e2 in the macros above, as far as it does not violate the protocol.

Figure 6 shows the code to parse and calculate additive-multiplicative expression of the grammar in Fig. 2. Before including file cpeglib.h, a semantic value type CPEG_VALUE_TYPE is defined as int. In the grammar rules, semantic values are calculated by copying the previous value of val and setting val to the result of addition or multiplication of the value.

The value of val can be examined in C code and reflected to the value of failed, effectively implementing semantic predicate [17], which uses the result of semantic analysis to direct parsing behavior.

Figure 7 shows the code that successfully parses a string composed of 'a' and 'b' only when the number of 'a' is greater than or equal to that of 'b'. Using this feature, programmers can write a grammar rule that succeeds only if a given identifier is registered in the symbol table, for example.
3.5 Implementation of Memoization Table

As mentioned earlier, CPEG maps each nonterminal into a C function. In our methodology, the memoization table is allocated statically as a static storage local to the corresponding C function. This static allocation not only eliminates the overhead of dynamic allocation, but also it is necessary to achieve our design goal that CPEG does not depend on memory-allocation functions such as malloc.

The size of memoization tables is fixed at compile time and can be specified by CPEG_MEMO_SIZE (the default value is 256 entries). Each entry of those tables contains the following three fields (Fig. 8):

- frompos: the value of pos where parsing started.
- topos: the value of pos where parsing finished. When parsing failed, FAIL(−1) is assigned to.
- v: the value of val (meaningful only if parsing succeeded).

Since memoization tables have a fixed size, some parsing results may not be memoized if the input is very large. In CPEG, a memoization table is treated as a cache of a result. Therefore, memoization tables have a finite size, and entries are accessed through an index pos%CPEG_MEMO_SIZE. If the frompos field of that entry is equal to pos, then the access is considered as a hit. In that case, if topos is FAIL, the table reveals that the parsing attempt already failed for the rule at pos and the parser can immediately return from the function after setting failed to true. Otherwise, the parser changes pos to topos, sets val to the value in the v field, and returns.

In the miss case, i.e., frompos of the entry at the index pos%CPEG_MEMO_SIZE is not pos, the right-hand side of the rule is executed and the result is stored in the entry of the table. When the entire function succeeded, the value of pos at the time the function is called is stored in frompos, the value of pos at the end of parsing is stored in topos, and the semantic value val is stored in v. When parsing failed, the value of pos at the time the function is called is stored in frompos and FAIL is stored in topos. In either case, the old values in the entry are overwritten. Initially, the frompos of the entry at index zero is set to NO_POS, which is a negative value, to avoid a wrong cache hit.

This implementation technique can be considered as a simplified version of the elastic sliding window [11]. In our technique, the memory size for memoization can be bounded to a constant value, but the linear execution time is not theoretically guaranteed since the parser may recalculate a previously recorded value, which was overwritten by the result for another position, which corresponds to the cache line replaced with the value of a different memory address. However, the recalculation only affects computational cost, not the correctness of the result. Also, a very small cache size has been reported to be sufficient in practice to achieve reasonable performance ([11], [19], [20]).

Figure 9 shows the definition of the PROLOGUE macro, which is included in BEGIN_RULE described in Section 3.1, and the definition of the other macros referred from PROLOGUE. The memoization table for a nonterminal is statically declared and initialized in MEMO_TABLE. When the function corresponding to a nonterminal is invoked, the entry corresponding to pos is examined. If the cache access hits, the function immediately returns the memoized value (MEMO_CHECK). Otherwise, the body of the function is executed after the index of the entry where PROLOGUE stores the result is assigned to first_pos.

The definition of the EPILOGUE macro is shown in Fig. 10. This macro is included in END RULE and it implements the memoization mechanism together with PROLOGUE. First, first_pos, the position where parsing started, is stored in the frompos field of the memoization table entry. If the parsing fails, assign FAIL into topos field. Otherwise, assign pos and val into topos and v, respectively.

When the macro CPEG_DO_MEMOIZE is undefined or defined as a constant zero, then PROLOGUE and EPILOGUE are replaced with empty macros that effectively do nothing.

3.6 External DSL Support

Since it is fairly straightforward to translate PEG into the CPEG internal DSL, only a little effort is required to learn the CPEG notation. Still, the syntactic restriction of C macros makes grammar description somewhat redundant. To improve productivity in writing a large set of grammar rules, and to promote the reuse of grammar rules written for other PEG-based systems, we implemented a tool that generate a C source program using the CPEG notation from an external DSL closer to PEG.

This tool supports the Mouse [18] grammar description format as its input file (excluding actions). The Java parser used in the experiments discussed in Section 4 was automatically translated by this tool into CPEG (Fig. 12) from the Java 1.7 grammar rules distributed together with Mouse (Fig. 11).

Using this tool, a programmer can first generate a skeleton...
CPEG parser from relatively short grammar description. Then she can add her own semantic actions to the output so that she can easily build a large parser.

### 3.7 Profiling and Tuning Memoization Performance

In CPEG, the size of a memoization table (measured in number of entries) for each function is fixed at runtime. Because of this design decision, the parsing is not guaranteed to terminate in linear time. Our CPEG provides a scheme to modify the table size if necessary. Although a larger table size does not guarantee linear-time-complexity for unbounded input size, a huge table size is rarely required in practice. In fact, Refs. [19] and [20] showed that it is sufficient to reserve just two linearly searched cache entries per each nonterminal to eliminate most occurrences of backtracking when parsing source code in C and Java. Also, Ref. [11] reported that, even for an input that exhibits a relatively bad behavior with maximum backtrack distance of 247 kB (jQuery), a memoization table with only 32 entries, which is accessed using a modulo operation like ours, is practically sufficient.

These studies also reported that there are a number of nonterminals for which memoization tables never hit. Memoization is completely useless for them.

In CPEG, the size of a memoization table can be changed by defining \( \text{CPEG}_\text{MEMO} \) SIZE, which can be redefined for each function by combining \#undef and \#define.

To disable memoization, the macro \( \text{CPEG}_\text{DO MEMOIZE} \) is defined to zero. Doing this suppresses all array declarations and read/write accesses of memoization tables. If memoization is not effective or desirable, this feature saves memory and reduces execution time. Since this macro can also be redefined, a programmer can control whether each function is to be memoized or not on a per-function basis.

\( \text{CPEG}_\text{DO COUNT} \) is a simple support for measuring memoization performance for each nonterminal. By setting this to a non-zero value, the \( \text{BEGIN RULE}(N) \) macro additionally declares variables \( \text{cpeg}_N \) called and \( \text{cpeg}_N \) missed, both of which are of type long and record the number of invocations of \( N \) and the number of times probing into the memoization table of \( N \) is missed, respectively. Currently, programmers must manually write the code for printing the values of these variables.

Table 3 shows the measured counts for the parser generated from a grammar artificially designed to cause an extremely large number of backtracking (Fig. 13), with input string "1+1+...+1" (29 characters long, containing 15 '1' characters and 14 '+' characters). During the measurement, we changed the table size for memoization. The results suggest that memoization is effective for both nonterminals \( E \) and \( P \). The table with one entry is sufficient for \( E \), and 16 entries may be insufficient (the performance may be improved when more entries are available) for \( P \).

The hit rates for all the nonterminals were zero when the input was parsed by the JavaScript Object Notation (JSON) parser written using CPEG for the experiments in Section 4. The memoization was totally unnecessary in that case. Table 4 shows the call count and hit rate of the memoization table for nonterminals

<table>
<thead>
<tr>
<th>Memoization-table size</th>
<th>( E ) miss/call</th>
<th>( P ) miss/call</th>
</tr>
</thead>
<tbody>
<tr>
<td>No memo</td>
<td>32,767/32,767</td>
<td>81,918/81,918</td>
</tr>
<tr>
<td>1</td>
<td>15/29</td>
<td>29/31</td>
</tr>
<tr>
<td>16</td>
<td>15/29</td>
<td>22/31</td>
</tr>
<tr>
<td>256</td>
<td>15/29</td>
<td>15/31</td>
</tr>
<tr>
<td>1,024</td>
<td>15/29</td>
<td>15/31</td>
</tr>
</tbody>
</table>
Table 4 Hit rate of memoization table for Java 1.7.

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Call count</th>
<th>Hit rate [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>LetterOrDigit</td>
<td>17,262,258</td>
<td>1.75</td>
</tr>
<tr>
<td>DOT</td>
<td>8,453,728</td>
<td>72.71</td>
</tr>
<tr>
<td>Spacing</td>
<td>6,325,628</td>
<td>0.11</td>
</tr>
<tr>
<td>Digits</td>
<td>4,314,321</td>
<td>66.13</td>
</tr>
<tr>
<td>Identifier</td>
<td>3,447,671</td>
<td>20.46</td>
</tr>
<tr>
<td>AT</td>
<td>3,442,313</td>
<td>66.84</td>
</tr>
<tr>
<td>LBRK</td>
<td>2,926,624</td>
<td>25.72</td>
</tr>
<tr>
<td>HexNumeral</td>
<td>2,870,238</td>
<td>49.70</td>
</tr>
<tr>
<td>INC</td>
<td>2,762,779</td>
<td>0.02</td>
</tr>
<tr>
<td>DEC</td>
<td>2,751,926</td>
<td>0.02</td>
</tr>
<tr>
<td>Keyword</td>
<td>2,742,373</td>
<td>0.00</td>
</tr>
<tr>
<td>PLUS</td>
<td>2,734,821</td>
<td>0.02</td>
</tr>
<tr>
<td>MINUS</td>
<td>2,698,830</td>
<td>0.02</td>
</tr>
<tr>
<td>Letter</td>
<td>2,507,360</td>
<td>0.00</td>
</tr>
</tbody>
</table>

of the Java parser written in CPEG when it parsed JDK 1.7 source code. The table lists the first ten nonterminals in the order of call count. The size of the memoization table was the default 256 entries. A nonterminal with a high call count and low (zero or almost zero) hit rate may be a good candidate for performance tuning since turning memoization off for such a nonterminal would probably make a parser run faster.

4. Performance Evaluation

In this section, we evaluate the execution speed of CPEG in a practical situation. Here, a recognizer is a program that parses an input string and determines accept or reject, but does not create any data structures such as parse trees. A parser is a program that parses an input string and computes some semantic values including parse trees.

The computing environment used for the experiments was shown in Table 5. Unless otherwise noted, we measured user CPU time as execution time. For all experiments, we repeated program execution ten times after one warm-up run. We show the average execution time of ten executions (or a scatter plot of ten execution times).

4.1 JSON Parser

We implemented recognizers for JSON[2] using CPEG and other libraries and compared their execution time when the same JSON text is given as input.

For input, we used the largest dataset (citys) in the JSON Serialization Benchmarks [1], and more complex data set (repos). The size of citys is 2.5 MB, and that of repos is 342 kB. Each of data sets was duplicated and concatenated several times to build larger input files since both of them are too small to reliably measure execution time.

We compared the parser written using CPEG to the ones generated by Bison and Flex (shown Bison in graph) and the parser written using cpp-peglib [9], which is a PEG library for C++. Although there could be a room for improvement since no backtracking occurs for the JSON parser written in CPEG as described in Section 3.7, we used the default CPEG parameter settings (memoization enabled, 256 memoization table entries).

We show the measured results for citys in Fig. 14, and those for repos in Fig. 15. Our CPEG performed better than the other implementations.

4.2 Java Parser

Figure 16 illustrates the measured results of the Java parsers. In the graph, MOUSE indicates the result of a recognizer generated by Mouse from the Java 1.7 grammar file distributed as a sample file with Mouse, Cpeg indicates a recognizer generated from the same grammar file converted to C code using CPEG, and Xtc indicates a parser generated using Rats! [7] and distributed as a part of the xtc compiler, which accepts Java 1.7 source code. The source files of JDK 1.7 (contents of jdk1.7.0_75.src.zip) are used as input. The graph shows user CPU time when the first 100, 200, . . ., and 1,000 input files were given to the parsers in the order displayed by find dir -name '*.java'.

Table 5 Experimental Environment.

<table>
<thead>
<tr>
<th></th>
<th>CPU:</th>
<th>OS:</th>
<th>Memory:</th>
<th>C compiler:</th>
<th>JDK version:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.6 GHz Intel Core i5</td>
<td>OS X El Capitan ver.10.11.4</td>
<td>8GiB 1600 MHz DDR3</td>
<td>gcc ver.4.2.1</td>
<td>1.8.0_131</td>
</tr>
</tbody>
</table>

Fig. 14 JSON(citys).

Fig. 15 JSON(repos).
Since the start symbol of Java grammar is a compilation unit, i.e., one file, all the parsers were reinvoked whenever starting to parse a new file. Hence, the user CPU time for Mouse and xtc include the startup time of the Java virtual machine (VM), which occupies a significant portion of the measured time. Since Mouse and xtc have facilities to measure only the parsing time, excluding the VM startup time, we also conducted the experiment by using these features. Figure 17 shows the results of Mouse with the \texttt{-t} option given to the \texttt{mouse}.TryParser class and the results of xtc with the \texttt{-performance -measureParser -java7 -ast} option given to the \texttt{xtc.lang.Java} class. The numbers indicates the real-time for parsing, measured using \texttt{System.currentTimeMillis()}. These times are much shorter than user CPU time including VM startup. For CPEG, the figure shows the real-time of the entire execution of the parser. Note that xtc is a parser that constructs parse trees, while the other two are recognizers.

We did not specially tune CPEG. We used the default memoization parameters. The experiments showed that CPEG ran faster than Mouse and xtc although the implementation languages differ.

There are 7,448 files in the JDK 1.7 source distribution. However, we used only up to 1,000 of them for measuring Mouse and xtc (repeated ten times for each size) due to time limitation. Figure 18 shows the results of CPEG (both user CPU time and real time) for up to 7,000 files. It reveals that there is no particular bias for the first 1,000 files.

Figure 19 illustrates the parsing time of the Java recognizer written using CPEG relative to the size of the memoization table.
(in terms of number of entries). The graph shows the average parsing time and the interval of ±1SD (standard deviation). The parsing time was measured by using 1,000 Java files for input, as in the previous experiments. As mentioned in Section 3.7, the accesses to the memoization table hits for only several nonterminals in Java parsers. However, from Fig. 19, we could draw no clear relation between the size of the memoization table and the parsing time. The performance benefit obtained by a larger table size is not supposed to be significant compared to other factors and/or fluctuations. One hypothesis is that the parser may show lower locality of references with a larger table size. To verify this, we used Ref. [15] and measured the hit rate of the L2 and L3 caches of the CPU. The results are shown in Fig. 20. Again, we could find no clear relation.

5. Conclusion

We proposed CPEG, a C library for writing parsers, which exploits the macro facility of C language to describe grammar rules (unlike parser combinators, in which grammar rules are dynamically allocated as runtime data structures), and statically allocate tables for memoization. As a result, CPEG does not rely on any specific memory-management schemes and can be easily used to implement parsers in a wide range of C programs.

Our comparison between our library and other existing libraries with respect to the parsing time of JSON and Java source files justifies the performance of CPEG is not slower than the other libraries.

5.1 Related Work

There are parser combinator libraries for C++ that support PEG [5], [9]. These libraries have a number of advantages such as collision avoidance by namespaces, expressiveness by operator overloading, and flexible but safe use of data types by parametric types. However, some developers might find CPEG preferable since it supports C instead of C++ and does not require any dynamic allocation, which is in common with most parser combinators, but requires only statically allocated memory.

Implementation of memoization tables using forms of ring buffers or sliding windows are already known [11], [13], [14]. CPEG differs from the existing systems in that it does not dynamically resize tables, it relies solely on static-memory management, and its implementation is very simple. In particular, if CPEG_MEMO_SIZE is a power of two, highly efficient code will be generated for modulo operation because of the optimization by the C compiler. Consequently, the overhead of memoization would be minimal in CPEG.

Mouse [18] also uses a fixed-size cache for memoization. Based on the observation that the backtracking due to PEG can be reduced by a small number of the entries (the authors reported that two entries are sufficient), Mouse generates PEG-based parsers that use zero (no memoization) to 9 12 cache entries. In pathological cases such as those shown in 3.7, such a small number of entries may not be sufficient.

Kuramitsu [11] argued that since it is difficult to intuitively predict whether a nonterminal requires memoization, dynamic adaptation is preferable. In some practical cases, however, the behavior is predictable from the statistics obtained from a number of test cases. In those cases, eliminating memoization beforehand will reduce overheads better than dynamically detecting the need for memoization. The existence of such cases (and how frequently if exist) is a question that requires further study.

5.2 Future Prospect

Grammar-specific debugging functionality would provide better supports than typical C debuggers to find and correct erroneous grammar description. It could greatly help easily implement parsers. Also, a user-friendly interface for performance measurement is needed.

Because of the redundancy of the relatively low-level notation of CPEG, we provided a translator tool from our external DSL based on the Mouse format. This tool serves as a good example to show the practicality of CPEG. It is an interesting idea to use CPEG as a target of translation from other higher-level DSLs since this approach preserves the advantages of CPEG while enabling further functionality and ease of grammar description.

Acknowledgments The authors would like to thank Professor Shigeru Chiba, University of Tokyo, for his constructive comments to the draft of the paper.

References


Yuta Sugimoto graduated from the College of Information Science, University of Tsukuba in 2017. 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 at the Faculty of Engineering, Information and Systems, University of Tsukuba. His research interests include implementation of programming languages, runtime systems, and dynamic resource management. He is a member of ACM and JSST.