Regular Paper

Implementation of C Library for Constructing Packrat Parser with Statically Allocated Memory

Yuta Sugimoto[1,a)] Atusi Maeda[1]

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 th 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. Incorporated 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 includes 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.

[1] Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki 305–0006, Japan
[a)] sugimoto@ialab.cs.tsukuba.ac.jp

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 $N \leftarrow e$, 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 e_1/e_2, e_1 is attempted first, and if it failed, the parser then backtracks and attempts e_2. In this way, ambiguity is eliminated and the parser always derives a unique result.

Also, and-predicate $\&e$ 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,

Table 1 Parsing expression.

empty string	:	ϵ
literal string	:	"a"
nonterminal	:	N
concatenation	:	$e_1 e_2$
ordered choice	:	e_1/e_2
zero or more repetition	:	$e*$
and-predicate	:	$\&e$
not-predicate	:	$!e$

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]).

2.3 DSLs for Parsing

We now summarize the common implementation techniques of parsers or parser generators from the viewpoint of grammar description as a DSL.

2.3.1 Parser Combinators

Some parsing libraries for languages, such as C++ or Haskell, exploit a kind of internal DSL called a *parser combinator*, which has a set of primitive parsers implemented as first-class functions or objects and *combinators* to compose complex parsers. Popular parser combinator libraries include Boost.Spirit [8] and cpp-peglib [9] for C++, and parsec [12] for Haskell. They provide simple parsers that correspond to terminals and nonterminals, with constructors that correspond to operators in grammar rules, which can be used to describe grammar rules in terms of expression of host languages to implement parsers. Generally, debugging of parsers rely on debugging functionalities of host languages.

2.3.2 Parser Generators

There are numerous systems called *parser generators* including Yet Another Compiler-Compiler (yacc) [10] developed on Unix in the 1970s, Bison [6], which is upward compatible to yacc, and many others ([7], [16], [18]). A parser generator translates input files that describe grammar rules written in notation specially designed for that purpose, and outputs parser programs in source code of languages such as C or Java.

The notation for an input file of a parser-generator system can be regarded as an external DSL that has its own syntax, which differs from general-purpose programming languages, but optimized for the very purpose of writing parsers. Users, however, must learn new grammar and how to use those tools that are not familiar to them.

3. Proposed Library

Our CPEG can be used to construct parsers by writing grammar rules in some internal DSLs hosted by C and closely corresponding to PEG.

Table 2 Parsing expression and CPEG macros.

Meaning	Parsing expression	Macros
empty string	ϵ	`EMPTY()`
any character	.	`ANY()`
literal character	'a'	`CHAR('a')`
literal string	"abc"	`STR("abc")`
concatenation	$e_1 e_2 \ldots$	`SEQ(e1, e2, ...)`
ordered choice	$e_1/e_2/\ldots$	`ALT(e1, e2, ...)`
zero or more	$e*$	`REPEAT0(e)`
one or more	$e+$	`REPEAT1(e)`
option	$e?$	`OPTION(e)`
and-predicate	$\&e$	`ANDPRE(e)`
not-predicate	$!e$	`NOTPRE(e)`

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

```
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.

EMPTY: EMPTY is a parsing expression that matches an empty string. It always succeeds and never changes pos.

ANY: matches any character. Succeeds unless pos is past the end of input and advances pos by one.

CHAR: succeeds when argument character matches input[pos], and advances pos by one.

SEQ: represents the concatenation of parsing expressions. Argument expressions are executed in order. It succeeds only if all of arguments succeed. If one of them fails, the whole SEQ expression fails, and pos will be rolled back to the point before SEQ is entered.

STR: if an argument string matches the input, pos will be advanced by the length of the string. The STR("$c_1c_2 \ldots c_n$") has the same meaning as SEQ(CHAR('c_1'), CHAR('c_1'), $\cdots$, CHAR('c_n')).

ALT: the first argument expression is executed and if it succeeds, the whole ALT expression succeeds. If failed, next argument is tried. If all expression fails, the whole ALT expression fails

REPEAT0: represents repetition of zero or more times and takes any parsing expression as an argument. It is executed until it fails. The REPEAT0 always succeeds.

REPEAT1: REPEAT1(e) represents repetition of one or more times. It is the same as SEQ(e, REPEAT0(e)).

OPTION: OPTION(e) represents an optional expression. It is the same as ALT(e, EMPTY()).

ANDPRE: represents an and-predicate and takes any parsing expression e. If execution of e succeeds, ANDPRE also succeeds; otherwise, ANDPRE fails. In either case, pos does not advance.

NOTPRE: represents a not-predicate and takes any parsing expression as an argument. If it succeeds, NOTPRE fails; otherwise, NOTPRE succeeds. In either case, pos does not advance.

CHARSET: if input character is included in the argument, which is a set of characters (in type cpeg_charset_t, described later), it advances pos by one.

3.3 Literal Strings and Character Sets

In principle, a combination of literal characters CHAR(c) and concatenation SEQ($e_1, e_2, \ldots$) is sufficient to express a grammar rule matching a string. Also, CHAR(c) and ALT($e_1, e_2, \ldots$) 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 charsets, as in many other implementations.

3.3.1 Literal Strings

The macro STR represents an expression that succeeds when its argument matches the string starting from input[pos] and increases pos by the length of the string.

Figure 1 illustrates an example of grammar description using STR. A rather complicated combination of SEQ and CHAR can be rewritten to be a concise expression with STR.

3.3.2 Character Sets

In many variants of PEG, notation $[c_1c_2 \cdots c_n]$ (where $c_1c_2 \cdots c_n$ are arbitrary characters) is used to abbreviate a parsing expression 'c_1' / 'c_2' / $\cdots$ / '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.

Figure 3 shows an example of using charset. This CPEG code implements the PEG for parsing additive and multiplicative expressions of decimal numbers (**Fig. 2**).

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

$$\downarrow$$

```
STR("abc");
```

Fig. 1 Example of STR.

$$E \leftarrow T \text{ '+' } E \;/\; T$$
$$T \leftarrow P \text{ '*' } T \;/\; P$$
$$P \leftarrow \text{ '(' } E \text{ ')' } \;/\; [0–9]+$$

Fig. 2 Grammar of additive-multiplicative expressions.

```
#include "cpeglib.h"
DECLARE_RULES(E, T, P);
cpeg_charset_t digit;

BEGIN_RULE(E) {
  ALT(SEQ(T(), CHAR('+'), E()),
      T());
} END_RULE

BEGIN_RULE(T) {
  ALT(SEQ(P(), CHAR('*'), T()),
      P());
} END_RULE

BEGIN_RULE(P) {
  ALT(SEQ(CHAR('('), E(), CHAR(')')),
      REPEAT1(CHARSET(digit)));
} END_RULE

#include <string.h> /* for strlen */
int main(int argc, char *argv[]) {
  input = argv[1];
  cs_range(digit, '0', '9');
  E();
  /* Succeeds if all of the input is parsed */
  return pos != strlen(argv[1]);
}
```

Fig. 3 Code to parse additive-multiplicative expression.

```
#define STMT(block) do block while (0)

#define SEQ_I(e1, e2)                              \
  STMT(                                            \
      {                                            \
        cpeg_pos_t first_pos = pos;                \
        e1;                                        \
        if (!failed) {                             \
          e2;                                      \
          if (failed) pos = first_pos;             \
        }                                          \
      } )
```

Fig. 4 C macro definition for concatenation (two-arg version).

```
#define ALT_I(e1, e2)                              \
  STMT(                                            \
      {                                            \
        e1;                                        \
        if (failed) {                              \
          failed = false;                          \
          e2;                                      \
        }                                          \
      } )
```

Fig. 5 C macro definition for ordered choice (two-arg version).

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.

```
#define CPEG_VALUE_TYPE int
#include "cpeglib.h"
DECLARE_RULES(E, T, P);

cpeg_charset_t digit;
BEGIN_RULE(E) {
  int v1;
  ALT(SEQ(T(), v1 = val, CHAR('+'), E(), val += v1),
      T());
} END_RULE

BEGIN_RULE(T) {
  int v1;
  ALT(SEQ(P(), v1 = val, CHAR('*'), T(), val *= v1),
      P());
} END_RULE

BEGIN_RULE(P) {
 int v1;
 ALT(SEQ(CHAR('('), E(), v1 = val, CHAR(')'), val = v1),
     SEQ(v1 = 0,
         REPEAT1(SEQ(CHARSET(digit),
                     v1 = v1 * 10 + (val - '0'))),
         val = v1));
} END_RULE

#include <string.h> /* for strlen */
#include <stdio.h> /* for printf */
int main(int argc, char *argv[]) {
  input = argv[1];
  cs_range(digit, '0', '9');
  E();
  printf("%d\n", val);
  return pos != strlen(argv[1]);
}
```

Fig. 6 Code for calculating additive-multiplicative expression.

```
BEGIN_RULE(S)
     {
       int num_a = 0, num_b = 0;
       REPEAT0(SEQ(ALT(SEQ(CHAR('a'), num_a++),
                       SEQ(CHAR('b'), num_b++))));
       failed = num_a < num_b;
     }
END_RULE
```

Fig. 7 Semantic predicate example.

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 `fail`, 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.

```
typedef struct memo_entry {
  cpeg_pos_t frompos;
  cpeg_pos_t topos;
  cpeg_value_t v;
} cpeg_result_t;
```

Fig. 8 Structure of memoization-table entry.

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 was 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 *true* memoization table, 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

```
#define PROLOGUE() MEMO_TABLE(); MEMO_CHECK(pos); \
  cpeg_pos_t first_pos = pos;

#define MEMO_TABLE()  \
 static cpeg_result_t memo[CPEG_MEMO_SIZE] = {{NO_POS}};

#define MEMO_CHECK(arg) STMT({                              \
  if(memo[arg%CPEG_MEMO_SIZE].frompos == arg) {             \
    val = memo[arg%CPEG_MEMO_SIZE].v;                       \
    failed = (memo[arg%CPEG_MEMO_SIZE].topos ==FAIL);\
    if(!failed) pos = memo[arg%CPEG_MEMO_SIZE].topos;\
    return;                                                 \
  }                                                         \
 })
```

Fig. 9 C macro definition for `PROLOGUE`.

```
#define EPILOGUE() STMT({                                  \
  memo[first_pos%CPEG_MEMO_SIZE].frompos = first_pos; \
  if (failed){                                             \
    memo[first_pos%CPEG_MEMO_SIZE].topos = FAIL;      \
  } else {                                                 \
    memo[first_pos%CPEG_MEMO_SIZE].topos = pos;       \
    memo[first_pos%CPEG_MEMO_SIZE].v = val;           \
  }                                                        \
})
```

Fig. 10 C macro definition for `EPILOGUE`.

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 `EPILOGUE` 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, `firstpos`, 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

```
CompilationUnit
   = Spacing PackageDeclaration? ImportDeclaration*
     TypeDeclaration* EOT
   ;

PackageDeclaration
   = Annotation* PACKAGE QualifiedIdentifier SEMI
   ;

ImportDeclaration
   = IMPORT STATIC? QualifiedIdentifier (DOT STAR)? SEMI
   / SEMI
   ;

...
```

Fig. 11 Excerpt of Java 1.7 grammar in the Mouse format.

```
DECLARE_RULES(n_CompilationUnit,
              n_PackageDeclaration,
              n_ImportDeclaration, ...);
cpeg_charset_t cs0;
cpeg_charset_t cs1;
cpeg_charset_t cs2;
...
BEGIN_RULE(START) {

/* cs0:[ \t\r\n\f] */
  cs_simple(cs0, '␣');
  cs_simple(cs0, '\t');
  cs_simple(cs0, '\r');
  cs_simple(cs0, '\n');
  cs_simple(cs0, '\f');
...
  n_CompilationUnit();
} END_RULE

BEGIN_RULE(n_CompilationUnit) {
  SEQ(n_Spacing(),
      OPTION(n_PackageDeclaration()),
      REPEAT0(n_ImportDeclaration()),
      REPEAT0(n_TypeDeclaration()),
      n_EOT());
}
END_RULE

BEGIN_RULE(n_PackageDeclaration) {
  SEQ(REPEAT0(n_Annotation()),
      n_PACKAGE(),
      n_QualifiedIdentifier(),
      n_SEMI());
} END_RULE

BEGIN_RULE(n_ImportDeclaration) {
  ALT(SEQ(n_IMPORT(),
          OPTION(n_STATIC()),
          n_QualifiedIdentifier(),
          OPTION(SEQ(n_DOT(),
                     n_STAR())),
          n_SEMI()),
      n_SEMI());
} END_RULE
...
```

Fig. 12 CPEG translated from Java 1.7 grammar (excerpt).

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 `CPEG_MEMO_SIZE`, which can be redefined for each function by combining `#undef` and `#define`.

To disable memoization, the macro `CPEG_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.

`CPEG_DO_COUNT` is a simple support for measuring memoization performance for each nonterminal. By setting this to a nonzero value, the `BEGIN_RULE(N)` macro additionally declares variables `cpeg_N_called` and `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.

$$E \leftarrow P\ \text{'+'}\ E\ \text{';'}\ /\ P\ \text{'+'}\ E\ /\ P$$
$$P \leftarrow [0\text{–}9]$$

Fig. 13 Grammar with extreme backtracking.

Table 3 Memoization performance for extremely backtracking grammar.

Memoization-table size	*E* miss/call	*P* miss/call
No memo	32,767/32,767	81,918/81,918
1	15/29	29/31
16	15/29	22/31
256	15/29	15/31
1,024	15/29	15/31

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 4 Hit rate of memoization table for Java 1.7.

Nonterminal	Call count	Hit rate [%]
LetterOrDigit	17,262,258	1.75
DOT	8,453,728	72.71
Spacing	6,325,628	0.11
Digits	4,314,321	66.13
LPAR	4,171,972	36.25
Identifier	3,447,671	20.46
AT	3,442,313	66.84
LBRK	2,926,624	25.72
HexNumeral	2,870,238	49.70
INC	2,762,779	0.02
DEC	2,751,926	0.02
Keyword	2,742,373	0.00
PLUS	2,734,821	0.02
MINUS	2,698,830	0.02
Letter	2,507,360	0.00

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.

Table 5 Experimental Environment.

CPU:	1.6 GHz Intel Core i5
OS:	OS X El Capitan ver.10.11.4
Memory:	8 GiB 1600 MHz DDR3
C compiler:	gcc ver.4.2.1
C compiler option:	-O3
JDK version:	1.8.0_131

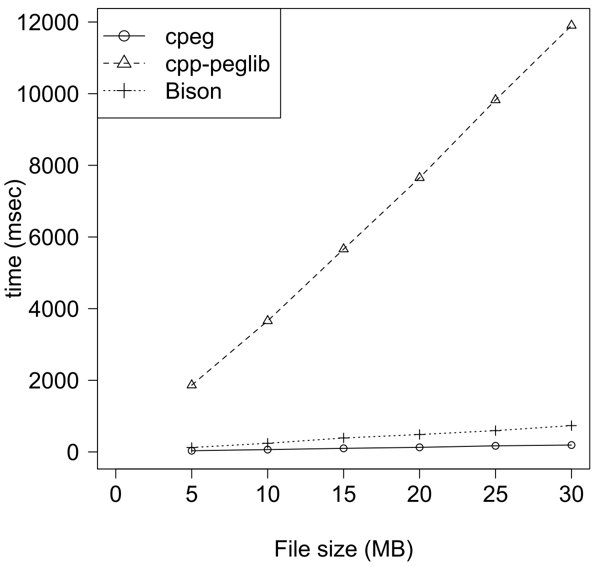


Fig. 14 JSON(citys).

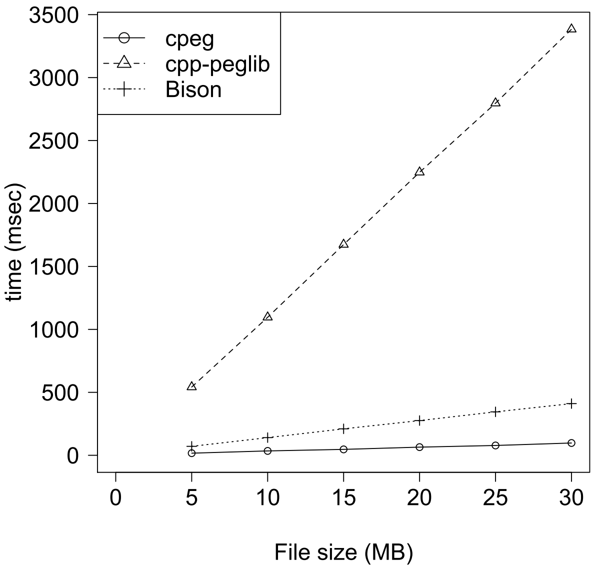


Fig. 15 JSON(repos).

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'`.

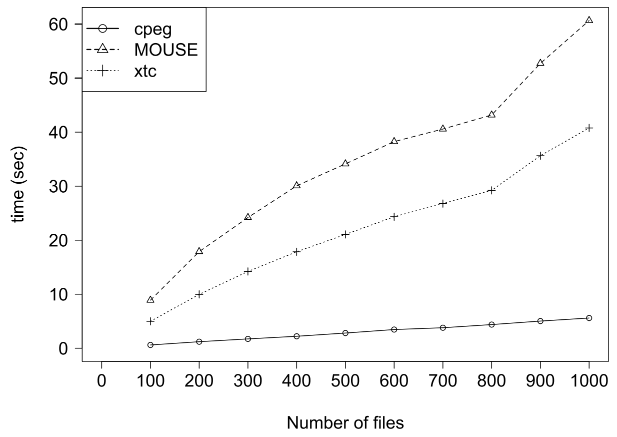


Fig. 16 Java parsers (user CPU time).

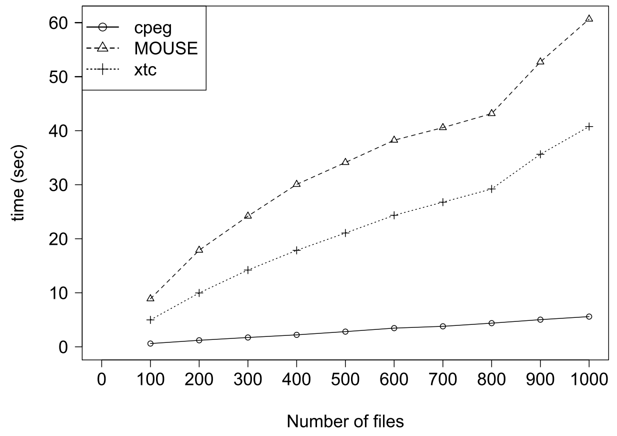


Fig. 17 Java parsers (real-time excl. VM startup).

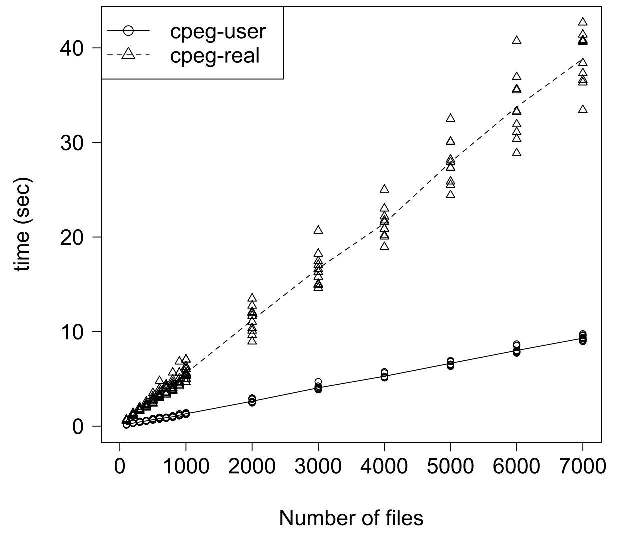


Fig. 18 Java parser (CPEG only, up to 7,000 files).

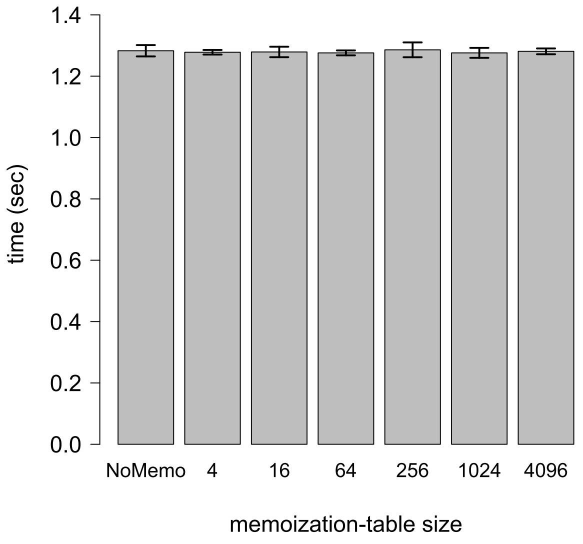


Fig. 19 Parsing time relative to memoization-table size.

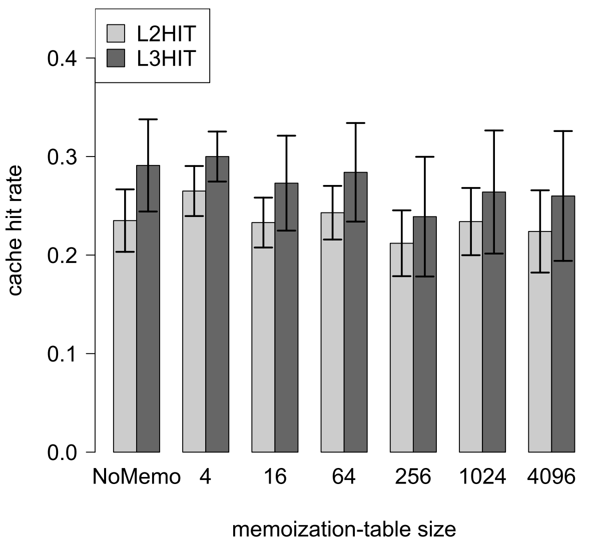


Fig. 20 Cache hit rate relative to memoization-table size.

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 `-t` option given to the `mouse.TryParser` class and the results of xtc with the `-performance -measureParser -java7 -ast` option given to the `xtc.lang.Java` class. The numbers indicates the real-time for parsing, measured using `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 [*1].

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 [*2] cache entries. In pathological cases such as those shown in 3.7, such a small number of entries may not be sufficient.

[*1] The source code is published at https://github.com/ialab/cpeglib/.

[*2] Note that the entries do not form a ring buffer nor a modulo-indexed array. They are linearly searched. The input position is a search key.

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

[1] Bloschetsov, A.: JSON Serialization Benchmarks (online), available from ⟨https://github.com/bura/json-benchmarks⟩ (accessed 2017-06-24).

[2] ECMA: ECMA-404: The JSON Data Interchange Format, ECMA International (online), available from ⟨http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf⟩ (accessed 2017-06-22).

[3] Ford, B.: Packrat Parsing: Simple, Powerful, Lazy, Linear Time, Functional Pearl, *Proc. 7th ACM SIGPLAN International Conference on Functional Programming, ICFP '02*, New York, NY, USA, pp.36–47, ACM (online), DOI: 10.1145/581478.581483 (2002).

[4] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '04*, New York, NY, USA, pp.111–122, ACM (online), DOI: 10.1145/964001.964011 (2004).

[5] Frey, D.: Taocpp/PEGTL: Parsing Expression Grammar Template Library (online), available from ⟨https://github.com/taocpp/PEGTL⟩ (accessed 2017-06-24).

[6] FSF: GNU Bison – The Yacc-compatible Parser Generator, Free Software Foundation (online), available from ⟨https://www.gnu.org/software/bison/manual/⟩ (accessed 2017-06-22).

[7] Grimm, R.: Better Extensibility Through Modular Syntax, *Proc. 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06*, New York, NY, USA, pp.38–51, ACM (online), DOI: 10.1145/1133981.1133987 (2006).

[8] Guzman, J.D.: Home of The Boost.Spirit Library (online), available from ⟨http://boost-spirit.com/home/⟩ (accessed 2017-06-23).

[9] Hirose, Y.: C++11 Header-Only PEG (Parsing Expression Grammars) Library (online), available from ⟨https://github.com/yhirose/cpp-peglib⟩ (accessed 2017-06-23).

[10] Johnson, S.C.: Yacc: Yet Another Compiler-Compiler, *UNIX Programmer's Manual*, Vol.2, Bell Telephone Laboratories, 7th edition, pp.353–387 (1978).

[11] Kuramitsu, K.: Packrat Parsing with Elastic Sliding Window, *Journal of Information Processing*, Vol.23, No.4, pp.505–512 (online), DOI: 10.2197/ipsjjip.23.505 (2015).

[12] Leijen, D. and Meijer, E.: Parsec: Direct Style Monadic Parser Combinators for the Real World, Technical Report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht (2001).

[13] Mizushima, K., Maeda, A. and Yamaguchi, Y.: Packrat Parsers Can

Handle Practical Grammars in Mostly Constant Space, *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '10*, New York, NY, USA, pp.29–36, ACM (online), DOI: 10.1145/1806672.1806679 (2010).

[14] Mizushima, K., Maeda, A. and Yamaguchi, Y.: A Space Complexity Calculation Method of Optimized Packrat Parsers, *Information Processing Society of Japan Transactions on Programming* (*PRO*), Vol.4, No.2, pp.77–91 (online), available from ⟨http://ci.nii.ac.jp/naid/110008616678/⟩ (2011).

[15] opcm: Processor Counter Monitor (PCM) (accessed 2017-06-25).

[16] Parr, T. and Fisher, K.: LL(*): The Foundation of the ANTLR Parser Generator, *Proc. 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, New York, NY, USA, pp.425–436, ACM (online), DOI: 10.1145/1993498.1993548 (2011).

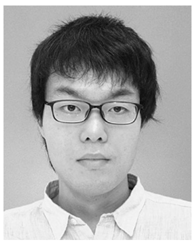
[17] Parr, T.J. and Quong, R.W.: Adding Semantic and Syntactic Predicates to LL(k): Pred-LL(k), *Computational Complexity*, pp.263–277, Springer-Verlag (1994).

[18] Redziejowski, R.R.: Mouse: From Parsing Expressions to a Practical Parser (online), available from ⟨http://www.romanredz.se/Mouse/⟩ (accessed 2017-06-24).

[19] Redziejowski, R.R.: Parsing Expression Grammar As a Primitive Recursive-Descent Parser with Backtracking, *Fundam. Inf.*, Vol.79, No.3-4, pp.513–524 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1366071.1366090⟩ (2007).

[20] Redziejowski, R.R.: Some Aspects of Parsing Expression Grammar, *Fundam. Inf.*, Vol.85, No.1-4, pp.441–451 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2365896.2365924⟩ (2008).

[21] SEI-CERT: SEI CERT C Coding Standard, Software Engineering Institute, Carnnegie Mellon University (online), available from ⟨https://www.securecoding.cert.org/confluence/display/c/⟩ (accessed 2017-06-24).



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 JSSST.