Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space

Kota Mizushima
Graduate School of Systems and Information Engineering, University of Tsukuba, Tennodai 1-1-1 Tsukuba, Ibaraki, JAPAN
mizusima@ialab.cs.tsukuba.ac.jp

Atusi Maeda
Graduate School of Systems and Information Engineering, University of Tsukuba, Tennodai 1-1-1 Tsukuba, Ibaraki, JAPAN
maeda@cs.tsukuba.ac.jp

Yoshinori Yamaguchi
Graduate School of Systems and Information Engineering, University of Tsukuba, Tennodai 1-1-1 Tsukuba, Ibaraki, JAPAN
yamaguti@cs.tsukuba.ac.jp

Abstract

Packrat parsing is a powerful parsing algorithm presented by Ford in 2002. Packrat parsers can handle complicated grammars and recursive structures in lexical elements more easily than the traditional $LL(k)$ or $LR(1)$ parsing algorithms. However, packrat parsers require $O(n)$ space for memoization, where n is the length of the input. This space inefficiency makes packrat parsers impractical in some applications. In our earlier work, we had proposed a packrat parser generator that accepts grammars extended with *cut* operators, which enable the generated parsers to reduce the amount of storage required.

Experiments showed that parsers generated from *cut*-inserted grammars can parse Java programs and subset XML files in bounded space.

In this study, we propose methods to automatically insert *cut* operators into some practical grammars without changing the accepted languages. Our experimental evaluations indicated that using our methods, packrat parsers can handle some practical grammars including the Java grammar in mostly constant space without requiring any extra annotations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Parsing; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems—Parsing

General Terms Languages, Algorithms, Performance

Keywords parsing expression grammars, packrat parsing, cut operators, memoization, backtracking, parser generator

1. Introduction

Today, so-called scripting languages (e.g., Perl, PHP, Ruby, etc.) are widely used in many systems such as Web applications. The grammars of these languages are typically more complicated than those of traditional programming languages. For example, newlines are used as statement terminators in Ruby, just like semicolons are used as statement terminators; however, newlines within expressions are treated as whitespaces and ignored. In addition, some languages have lexical elements that can include recursive syntactic structures. For example, string literals in Ruby can contain arbitrary expressions.

These syntactic features cannot be handled easily using LL-family grammars, such as that used by JavaCC[1], or using LR-based grammars, such as that used by Yacc[12] and many other parser generators. The implementation of these features using traditional LL- or LR-based parser generators requires many ad-hoc approaches and complicates the parser descriptions, making parsers difficult to maintain. Ruby's grammar file, for example, exceeds 8,000 lines of code including a special stateful lexical analyzer that can recursively invoke the parser.

In contrast, packrat parsers can handle complicated grammars more easily than traditional LL and LR parsers. Furthermore, packrat parsers can handle grammars easily in cases where conflicts occur in traditional LL and LR parsers, such as dangling-if-else statements. Despite their power and flexibility, packrat parsers guarantee linear time parsing. However, packrat parsers are space-inefficient and require linear space in parsing for memoization relative to input size.

In our earlier work[13], we proposed the addition of *cut* operators to parsing expression grammars (PEGs)[9], on which packrat parsing is based, to overcome its disadvantage. The concept of *cut* operators, which we borrowed from Prolog[6], enables grammar writers to control backtracking. By manually inserting *cut* operators into a PEG grammar, an efficient packrat parser that can dynamically reclaim unnecessary space for memoization can be generated. To evaluate the effectiveness of *cut* operators, we implemented a packrat parser generator called *Yapp* that accepts *cut* operators in addition to ordinary PEG notations. The experimental evaluations showed that the packrat parsers generated using grammars with *cut* operators inserted can parse Java programs and subset XML files in mostly constant space, unlike conventional packrat parsers.

In this paper, we describe methods that achieve the same effect in some practical grammars without manually inserting *cut* operators. In our methods, a parser generator statically analyzes a PEG grammar to find the points at which the parser generator can insert *cut* operators without changing the meaning of the grammar and then inserts *cut* operators at these points.

The remainder of this paper is organized as follows. First, we introduce the background of our work. We mainly focus on parsing expression grammars (PEGs) and packrat parsing, on which our work is based. Then, we introduce the notion of *cut* operators,

PASTE'10, June 5–6, 2010, Toronto, Ontario, Canada.

ε	:	Empty string
`" "`	:	String literal
`[ ]`	:	Character class
.	:	Wildcard (Any character)
(e)	:	Grouping
N	:	Nonterminal
$e_1\ e_2$	:	Sequence
$e_1\ /\ e_2$	:	Ordered-choice
$e*$	:	Zero-or-more repetition
$\&e$	:	And-predicate (Positive lookahead)
$!e$	:	Not-predicate (Negative lookahead)
$e+$	:	One-or-more repetition
$e?$	:	Zero-or-more.

Figure 1. Expressions Constituting a Parsing Expression

which was proposed in our earlier work, to overcome the disadvantages of packrat parsing. We describe the motivation, definition, properties, effectiveness, and problems of *cut* operators. Then, we describe methods for automatically inserting *cut* operators, which is the main contribution of this paper. Then, we show the evaluation results in terms of memory efficiency and parsing speed. Finally, we conclude this paper and present future prospects.

2. Background

2.1 Parsing Expression Grammars (PEGs)

PEGs is a recognition-based formal syntactic foundation that was first presented by Ford[9] to describe the syntax of formal languages. PEGs are a formalization of recursive descent parsing with backtracking. A PEG consists of rules, represented by $N \leftarrow e$, where N is a nonterminal symbol and e, an expression (called parsing expression). A parsing expression consists of the following elements, as shown in figure 1:

We do not consider $e+$ in later sections because it can be desugared to $e\ e*$ easily. Additionally, we distinguish ε from string literal, which cannot be empty string in this paper for the brevity. In the actual use of PEGs, the empty string literal `""` is allowed.

It appears that PEGs are similar to Extended Backus-Naur Forms (EBNFs)[14]. However, it should be noted that $e_1\ /\ e_2$ is *an ordered choice*, not *an unordered choice*, $e_1 \mid e_2$ in EBNFs. $e_1\ /\ e_2$ does not indicate strings expressed by e_1 or e_2. $e_1\ /\ e_2$ means an action in which e_1 is evaluated at first and e_2 is evaluated only when e_1 fails. Therefore, generally, $e_1\ /\ e_2\ \neq\ e_2\ /\ e_1$. In addition, it should be noted $e*$ is not similar to the operators used in regular expressions (REs). $e*$ does not mean a greedy match in REs but a possessive match. For example, `"a"` $*$ `"a"` in PEG does not express $\{$`"a"`, `"aa"`, $\ldots\}$ but $\emptyset$ because once `"a"`$*$ succeeds, the parser does not backtrack even if successive expressions `"a"` do not succeed. $\&e$ and $!e$ are lookahead expressions. $!e$ succeeds if e does not succeed and $\&e$ succeeds if e succeeds. Note that $!e$ and $\&e$ don't change the position in an input of a parser even if the expressions succeed.

PEGs can express all deterministic $LR(k)$ languages and some non-context-free languages[9].

2.2 Packrat Parsing

Packrat parsing is a parsing algorithm that was first presented by Ford[8]. Roughly speaking, packrat parsing is a combination of PEG-based recursive descent parsing with memorization. A packrat parser takes each nonterminal of a PEG as a parsing function of the nonterminal and carries out parsing on an input by calling the function. A parsing function takes a start position in an input as an argument and returns a parse result, which is failure or success. A parsing function must be pure in a packrat parser. That is, the same parsing function called at the same position returns the same result. This fact allows parsing functions to be memoized. Memoization of all parsing functions in a packrat parser guarantees that the packrat parser parses any input in linear time.

Despite the guarantee of linear time parsing, packrat parsing is powerful. Packrat parsing can rapidly handle wide-ranged grammar PEGs can express, including all deterministic $LR(k)$ languages and some non-context-free languages. In addition, packrat parsing does not require a separate lexer because when one of the choices fails in parsing, a packrat parser can backtrack to the other choice, unlike traditional LL or LR predictive parsers. Furthermore, because packrat parsers are simple, they can be implemented easily. Although packrat parsing is a simple, powerful, and linear-time parsing algorithm, it has a major disadvantage in that packrat parsers require $O(n)$ space for memoization in parsing because they memoize all intermediate results. Because of this disadvantage, packrat parsers are considered to be unsuitable for large file parsing (e.g. XML streams)[8].

Rats![10], which generates packrat parsers in Java, supports several optimizations to improve execution performance and memory efficiency. For example, *Rats!* merges successive memoized fields into objects called *chunks* to decrease the heap size. By *chunks* and many other optimizations, parsers generated by *Rats!* achieve an execution performance comparable to that of LL parsers generated by *ANTLR*[15]. However, *Rats!* does not resolve the fundamental problem that packrat parsers require $O(n)$ space.

3. Cut Operators

In this section, we describe *cut* operators. *Cut* operators, which are conceptually borrowed from Prolog, were proposed in our earlier work[13] to overcome the disadvantage that packrat parsers require $O(n)$ space in parsing. First, we illustrate the usage of *cut* operators by an example using a simple PEG. Then, we present an informal definition of the *cut* operator. This description mainly concerns the points at which a *cut* operator can be inserted and the behavior of a *cut* operator. Next, we discuss the properties of *cut* operators. We describe how packrat parsers can remove unnecessary space for memoization dynamically using *cut* operators. Then, we explain the result of an experimental evaluation of *cut* operators. Finally, we describe the problems faced with the use of *cut* operators.

3.1 Motivation

We consider the following PEG expressing a programming language's control statements (rules about spacing, the definitions of E and I are omitted):

$$\begin{array}{rcl} S & \leftarrow & \texttt{"if"}\ \texttt{"("}\ E\ \texttt{")"}\ S\ (\texttt{"else"}\ S)? \\ & / & \texttt{"while"}\ \texttt{"("}\ E\ \texttt{")"}\ S \\ & / & \texttt{"print"}\ \texttt{"("}\ E\ \texttt{")"}\ S \\ & / & \texttt{"set"}\ I\ \texttt{"="}\ E\ \texttt{";"};. \end{array}$$

When one of `"if"`, `"while"`, `"print"`, or `"set"` succeeds, other choices never succeed. When one of these choices is being evaluated, the packrat parser normally saves information for backtracking in preparation of failure. However, in this case, it is wasteful because after matching one of the keywords in a choice, it would not be possible to succeed with other choices. In such a case, we want the parser to avoid saving backtrack information.

By inserting *cut* operators ($\uparrow$) as follows, grammar writers can instruct the parser to dispose of the backtrack information as soon as it becomes unnecessary:

$$
\begin{array}{lcl}
S & \leftarrow & \texttt{"if"} \uparrow \texttt{"("}\, E\, \texttt{")"}\, S\, (\texttt{"else"}\, S)?\\
 & / & \texttt{"while"} \uparrow \texttt{"("}\, E\, \texttt{")"}\, S\\
 & / & \texttt{"print"} \uparrow \texttt{"("}\, E\, \texttt{")"}\, S\\
 & / & \texttt{"set"}\, I\, \texttt{"="}\, E\, \texttt{";"};.
\end{array}
$$

The above PEG instructs the parser to not backtrack to the other choices when one of the `"if"`, `"while"`, or `"print"` succeeds by *cut* operators. Note that a *cut* operator is meaningless in the last choice because no further choices are left.

Another motivation for introducing cut operators is error-reporting. In a backtracking parser, it is more difficult to report the exact point that the parser failed than a predictive parser because local failure is recovered by backtracking and the point that the parser failed at last is reported. Cut operators can improve readability of error messages because cut operators suppress backtracking.

3.2 Definition

Cut operator is a 0-arity operator and is rendered as ↑ or ^ (in ASCII). A *cut* operator can be inserted in the following two cases. First, you can insert a *cut* in the left-hand side of an ordered choice e_1 / e_2. That is, an expression of the form $e_1 \uparrow e_2 / e_3$ is a valid expression. However, an expression of the form $e_1 / e_2 \uparrow e_3$ is not valid. Another place you can insert a *cut* is the body of a repetition $e*$. That is, an expression of the form $(e_1 \uparrow e_2)*$ is a valid expression. However, an expression of the form $(e_1\ e_2)* \uparrow e_3$ is not allowed.

The semantics of a *cut* operator are informally described as follows:

- $e_1 \uparrow e_2 / e_3$:
  1. e_1 is evaluated.
  2. If e_1 fails in 1., e_3 is evaluated. Otherwise, e_2 is evaluated and e_3 *is never evaluated* even if e_2 fails. Contrast this with the case where *cut* is not inserted, in that case e_3 is evaluated when e_2 fails.

  A *cut* operator can be considered to be similar to an if-then-else statement in conventional programming languages (if e_1 then e_2 else e_3).
- $(e_1 \uparrow e_2)*$:
  1. e_1 is evaluated.
  2. If e_1 fails in 1., the entire expression succeeds. Otherwise, e_2 is evaluated.
  3. If e_2 fails in 2., the *entire expression fails*. Otherwise, the parser goes back to 1. and repeats the evaluation.

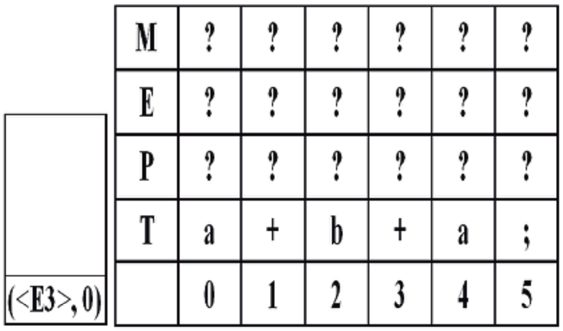
  Because $e*$ can be desugared to N and $N \leftarrow e\ N / \varepsilon$, where N is a fresh nonterminal, $(e_1 \uparrow e_2)*$ can also be desugared to N and $N \leftarrow e_1 \uparrow e_2\ N / \varepsilon;$.

In other words, if an evaluator comes through a *cut* operator, it does not backtrack to the other choice.

3.3 Properties

Existing packrat parsers memoize all intermediate results in preparation for backtracking. Eliminating the possibility of backtracking with *cut* operators, also enables a packrat parser to remove unnecessary space for memoization. In this subsection, we describe how a packrat parser can remove unnecessary space. As an example, we use the following PEG with *cut* operators, which expresses simple mathematical expressions:

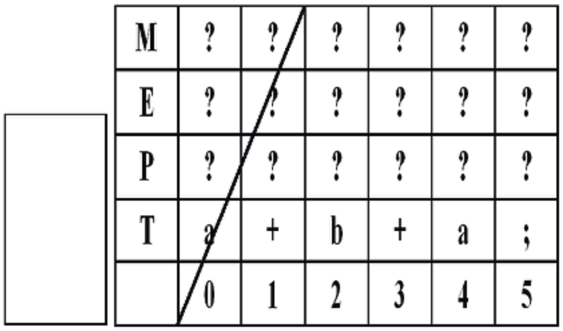
$$
\begin{array}{lcl}
M & \leftarrow & E\, \texttt{";"};\\
E & \leftarrow & \texttt{<E1>}\, P\, \texttt{"+"} \uparrow \texttt{<E2>}\, E \,/\, \texttt{<E3>}\, P;\\
P & \leftarrow & \texttt{"a"} \,/\, \texttt{"b"};.
\end{array}
$$



(<E3>, 0)

M	?	?	?	?	?	?
E	?	?	?	?	?	?
P	?	?	?	?	?	?
T	a	+	b	+	a	;
	0	1	2	3	4	5

Figure 2. Backtracking Stack and Memoization Table At (`<E1>`,0). The first column indicates nonterminals, the last row indicate positions in the input, and an element in the row of T indicates characters constituting the input. Otherwise, an element in the table is `?`, which indicates that it has not been parsed yet, or i, which is the index of the rest.



M	?	?	?	?	?	?
E	?	?	?	?	?	?
P	?	?	?	?	?	?
T	a	+	b	+	a	;
	0	1	2	3	4	5

Figure 3. Backtracking Stack and Memoization Table At (`<E2>`,2).

For the purpose of illustration, we consider a packrat parser's state as a pair (`<X>`, i) and a stack of pairs as (`<X>`,i), where `<X>` indicates a point in a PEG and i indicates a position (0-origin) in a input string. The stack is used for backtracking. In addition, we consider the space for memoization of the packrat parser as a table in which row indices are nonterminals, column indices are positions, and elements are parse results. We ignore the call stack of nonterminals to simplify the explanation.

For the input `a+b+a;`, suppose that the parser start to parse the input from M. At (`<E1>`, 0), the backtracking stack and the memoization table are as shown in Figure 2.

Figure 2 shows that the parser has to push (`<E3>`, 0) for backtracking before P `"+"` E is evaluated. Because the backtracking stack is not empty and the position 0 is pushed to the stack, the parser cannot discard space for memoization at this point. However, at (`<E2>`, 2), the backtracking stack and the memoization table are as shown in Figure 3.

Figure 3 shows that the backtracking stack is empty at that point by the evaluation of the *cut* operator. Therefore, the parser would never backtrack to a position before 2 in future and it can discard the space for memoization that is indicated by the diagonal line in Figure 3.

Generally, when the backtracking stack of a parser is empty at position n, the parser can discard all regions in the memoization table whose column indices are less than n. *Cut* operators help a packrat parser to empty out its own backtracking stack because an evaluation of a *cut* operator decreases the size of the backtracking stack. If *cut* operators are inserted appropriately in the definition of grammars, the packrat parser generated from the grammar would require only almost constant space for memoization.

3.4 Effectiveness

In our earlier work, we developed a packrat parser generater called *Yapp*, which involves *cut*-enhanced PEG, to evaluate the effectiveness of our idea. Our experimental evaluations are similar to those described in section 6. Evaluation result shows that the parsers generated from the grammars of Java and an XML subset PEG, both with manually inserted *cut* operators, require almost constant space regardless of the input size in contrast with parsers generated from the grammars in which *cut* operators are not inserted. The term ”almost constant space” means that we have no problem with taking the memory consumption as constant space practically. Furthermore, the parsers generated from *cut*-inserted grammars achieved better execution throughput than those generated from the grammars without *cut* operators.

3.5 Problem

We have now seen that appropriate use of *cut* operators can drastically improve the memory efficiency of packrat parsers. However, inserting *cut* operators by hand is tedious and error-prone process. Careless grammar writers may unintentionally change the meaning of grammars. Consider the following use of *cut* operators:

$$\begin{array}{lcl} M & \leftarrow & E\,\texttt{";"}; \\ E & \leftarrow & P\,\texttt{"+"}\ \uparrow\ E\,/\,P; \\ P & \leftarrow & \uparrow\,\texttt{"a"}\,/\,\texttt{"b"};. \end{array}$$

In the evaluation of P, the *cut* operator before `"a"` is definitely evaluated. As a result, `"b"` in P is never evaluated and P expresses only `"a"`. Consequently, M expresses only `"a;"`, `"a+a;"`, `"a+a+a;"`,... despite the grammar writer’s intention that it express `"a"`, `"b"`, `"a+a"`, `"a+b"`, `"b+a"`, `"b+b"`, ... Writing large, practical grammars with correct and efficient use of *cut* operators is nontrivial task, to say the least. It would be better to achieve the same effect without requiring the manual insertion of *cut* operators.

4. Automatic Insertion of Cut Operators

We propose two methods for automatic insertion of *cut* operators to resolve the abovementioned problems. In this section, we present a brief overview of these methods. First, we define the automatic *cut* insertion problem as the problem of defining function Ins which satisfies the following equations:

$$\begin{array}{lcl} e_1\ e_2\ /\ e_3 & = & e_1\ \uparrow\ e_2\ /\ e_3 \text{ if } \mathrm{Ins}(e_1\ e_2\ /\ e_3) \\ (e_1\ e_2)* & = & (e_1\ \uparrow\ e_2)*\ \text{ if } \mathrm{Ins}((e_1\ e_2)*). \end{array}$$

To decide whether $\mathrm{Ins}(e_1\ e_2\ /\ e_3)$ or not, we must decide whether $L(e_1)\ \cap L(e_3)\ =\ \emptyset$ or not, where $L(e)$ represents a language that consists of all strings accepted by a parsing expression e[1].

When e_1 is $.*$, which accepts any string, the problem would be to decide whether $L(e_3) = \emptyset$ or not. Unfortunately, this problem is undecidable[9]. So, we will compute a more conservative approximation of Ins.

4.1 Terminology

Before describing our methods, we define the terminologies used in this section.

- *cursor*: The current position of a parser. A *cursor* is the index of input. The *cursor* move to right with progress of parsing and move to left with backtracking.
- G: The set of all parsing expressions in a PEG.
- $!(X)$, where X is a set of expressions $\{e_1, ..., e_n\}$: It denotes $!(e_1\ /.../\ e_n)$. For example, $!(\{$`"a"`$,$`"b"`$\})$ denotes $!($`"a"` $/$ `"b"`$)$.
- *fail* and *nul* [18]: *fail* is the set of expressions that may fail. For example, `"a"` $\in fail$ and $\varepsilon \notin fail$. *nul* is the set of expressions that may succeed without moving *cursor*. For example, $\varepsilon \in nul$ and `"a"` $\notin\ nul$. The method for computing *fail* and *nul* is omitted. In this paper, we assume that the judgements of $e\ \in fail$ and $e\ \in\ nul$ may be conservative. That is, if $e\ \notin fail$, e never fails and if $e \notin\ nul$, e can only accept nonempty strings.
- T: The set of nonempty terminal expressions. An expression e of the form `" ... "`,`[ ... ]`, or $. \in T$.
- `[...]` $\subset$ `[...]`: A character class includes another character class. For example, `[ab]` $\subset$ `[abc]`.
- `[...]` $\cap$ `[...]`: Intersection of two character classes. For example, `[ab]` $\cap$ `[bc]` $= [a]$ and `[ab]` $\cap$ `[cd]` $=$ `[]`.
- `x` $\in$ `[...]`: a character is included in a character class. For example, `'a'` $\in$ `[abc]`.
- $s[i]$: The ith character of string literal s. i is 0-origin. For example, `"ab"`$[0]$ is `'a'`.
- $e_1 \leq e_2$ $(e_1, e_2\ \in\ T)$: e_1 is a prefix of e_2. $e_1 \leq e_2 =$

$$\left\{ \begin{array}{ll} \text{true} & \text{if } \mathrm{s?}(e_1) \wedge \mathrm{s?}(e_2) \wedge (e_2 \\ & \text{starts with } e_1) \\ \text{false} & \text{if } \mathrm{s?}(e_1) \wedge \mathrm{s?}(e_2) \wedge (e_2 \text{ does not} \\ & \text{start with } e_1) \\ \text{false} & \text{if } \mathrm{s?}(e_1) \wedge \mathrm{c?}(e_2) \wedge \mathrm{len}(e_1) > 1 \\ e_1[0] = a & \text{if } \mathrm{s?}(e_1) \wedge \mathrm{c?}(e_2) \wedge \mathrm{len}(e_1) = 1 \wedge \\ & e_2 = \texttt{[a]} \text{ where } a \text{ is one character} \\ \text{false} & \text{if } \mathrm{s?}(e_1) \wedge \mathrm{d?}(e_2) \\ e_2[0] \in e_1 & \text{if } \mathrm{c?}(e_1) \wedge \mathrm{s?}(e_2) \\ e_2 \subset e_1 & \text{if } \mathrm{c?}(e_1) \wedge \mathrm{c?}(e_2) \\ \text{false} & \text{if } \mathrm{c?}(e_1) \wedge \mathrm{d?}(e_2) \\ \text{true} & \text{if } \mathrm{d?}(e_1). \end{array} \right\}$$

  where $\mathrm{s?}(e) =$ true if e is a string literal, $\mathrm{c?}(e) =$ true if e is a character class, and $\mathrm{d?}(e) =$ true if e is a wildcard.
- $e_1 \leq_{?} e_2$: e_1 *maybe* prefix of e_2. $e_1 \leq_{?} e_2 =$

$$\left\{ \begin{array}{ll} e_1 \cap e_2 \neq \texttt{[]} & \text{if } \mathrm{c?}(e_1) \wedge \mathrm{c?}(e_2) \\ \text{true} & \text{if } \mathrm{c?}(e_1) \wedge \mathrm{d?}(e_2) \\ e_1 \leq e_2 & \quad\text{otherwise.} \end{array} \right\}$$

- $R*$: The transitive and reflexive closure of relation R.
- $R(e)$, where R is a relation and e is an expression: It means the set of all expressions e_2 that $(e, e_2)\ \in R$.

4.2 Basic Ideas

Our basic ideas for automatic *cut* insertion methods are simple. First, at the choice e_1/e_2 in parsing, if the prefix of input is accepted by only the *first* terminal expressions of e_1, information for

[1] Note that the definition $L(e)$ doesn’t require that *all* of string must be matched to e but a *prefix* of string must be matched to e. Its definition is same as Ford’s definition of Parsing Expression Language (PEL)[9].

backtracking can be avoided after the evaluation of the first terminal expressions of e_2 *failed*. The idea is similar to $LL(1)$ parsing. For example, the following PEG

$$S \quad \leftarrow \quad \texttt{"if"} \ldots / \texttt{"while"} \ldots;$$

can be translated to the following PEG:

$$S \quad \leftarrow \quad !(\texttt{"while"}) \uparrow \texttt{"if"} \ldots / \texttt{"while"} \ldots;.$$

Second, at the start of repetition e_1* in the expression $e_1 * e_2$, if the prefix of input is accepted by only the first terminal expressions of e_1, information for backtracking can be avoided after the evaluation of the first terminal expressions of e_2 failed in the same manner. For example, the following PEG

$$S \quad \leftarrow \quad \texttt{"\{"} \, (\texttt{"if"} \ldots / \texttt{"while"} \ldots) * \texttt{"\}"};$$

can be translated to the following PEG:

$$S \quad \leftarrow \quad \texttt{"\{"} \, (!(\texttt{"\}"}) \uparrow (\texttt{"if"} \ldots / \texttt{"while"} \ldots)) * \texttt{"\}"};.$$

Note that lookahead by not-predicate is needed for the case that contents of choices and repetitions are organized into other rules.

4.3 AC-FIRST

In this subsection, we describe an automatic *cut* insertion method AC-FIRST. AC-FIRST uses the relation FIRST on G[18], which was defined by Redziejowski and is inspired by the notion used in predictive top-down parsers. In [18], the relation First (not FIRST) is defined as follows (The notation is changed from Redziejowski's definition for lucidity):

$e_1 \in \texttt{First}(e)$ iff

$$\begin{array}{lcl} e & = & e_1/e_2 \text{ for some } e_2 \\ e & = & e_2/e_1 \text{ for some } e_2 \in fail \\ e & = & e_1\, e_2 \text{ for some } e_2 \\ e & = & e_2\, e_1 \text{ for some } e_2 \in nul \\ e & = & !e_1 \\ e & = & e_1* \\ e & = & e_1?. \end{array}$$

Using the relation First, the relation FIRST is defined as follows:

$$\texttt{FIRST}(e) \quad = \quad \texttt{First} * (e) \cap T.$$

Intuitively, FIRST(e) can be considered to be the set of terminal expressions of which at least one must be evaluated at the *cursor* when e is evaluated. For example,

$$\texttt{FIRST}(\texttt{"if"} \ldots / \texttt{"while"} \ldots) \quad = \quad \{\texttt{"if"}, \texttt{"while"}\}.$$

Additionally, if the condition $e \notin nul$ holds, FIRST(e) can be considered to be the set of terminal expressions of which at least one must *succeed* first when e *succeeds*. Therefore, If the following condition holds,

$$\begin{array}{l} e_1 \notin nul \wedge e_2 \notin nul \wedge \\ \forall t_1 \in \texttt{FIRST}(e_1), t_2 \in \texttt{FIRST}(e_2). t_1 \not\lesssim_? t_2 \wedge t_2 \not\lesssim_? t_1 \end{array}$$

then an expression $e_1 \ / \ e_2$ can be translated to

$$!(\texttt{FIRST}(e_2)) \uparrow e_1 \ / \ e_2$$

without changing its meaning. Recall that $!(X)$, where X is set of expressions $\{e_1, ..., e_n\}$, means an expression $!(e_1 /.../ e_n)$.

However, an expression $A \ / \ B$, where A and B are nonterminals that have rules $A \leftarrow \texttt{"a"}$ and $B \leftarrow \texttt{"b"}$, respectively, cannot be translated likewise because content of nonterminals are not expanded when computing FIRST. To resolve this problems, we first define another relation ExtFirst as follows:

$e_1 \in \texttt{ExtFirst}(e)$ iff

$$\begin{array}{l} e = N (\text{where } N \text{ is a nonterminal with a rule } N \leftarrow e_1) \\ e_1 \in \texttt{First}(e). \end{array}$$

ExtFirst is the same as First except for the first line to obtain the content of nonterminals. Then, we define relation ExtFIRST as follows:

$$\texttt{ExtFIRST}(e) \quad = \quad \texttt{ExtFirst} * (e) \cap T.$$

Using the relation ExtFIRST, if the following condition holds,

$$\begin{array}{l} e_1 \notin nul \wedge e_2 \notin nul \wedge disjoint(e_1, e_2) \\ disjoint(e_1, e_2) = \\ \quad \forall t_1 \in \texttt{ExtFIRST}(e_1), t_2 \in \texttt{ExtFIRST}(e_2). \\ \quad\; t_1 \not\lesssim_? t_2 \wedge t_2 \not\lesssim_? t_1 \end{array}$$

then an expression $e_1 \ / \ e_2$ can be translated to the following expression:

$$!(\texttt{ExtFIRST}(e_2)) \uparrow e_1 \ / \ e_2.$$

Note that our method may translate an expression e, of which evaluation may not terminate, to another, terminating one, if $e \notin$ *nul* $\wedge$ $\texttt{ExtFIRST}(e) = \emptyset$. Since such possible non-termination should be a bug in the grammar, we do not consider this as a flaw in our method.

4.4 AC-Repetition

Similar to the problem described in the previous subsection, we will give the condition to translate an expresssion containing repetition $e_1 * e_2$ to $(!e_2 \uparrow e_1) * e_2$ without changing its meaning, as follows:

$$e_1 \notin nul \wedge e_2 \notin nul \wedge disjoint(e_1, e_2).$$

We use ExtFIRST to obtain the first terminal expressions of e_2. Using ExtFIRST, an expression $e_1 * e_2$ in which the above condition holds can be translated to the following expression without changing its meaning:

$$(!(\texttt{ExtFIRST}(e_2)) \uparrow e_1) * e_2.$$

However, if there are no expressions following the repetition, the translation is not applicable.

To resolve this problem and translate repetition $e*$ at the tail position, the follower e_2 of the last non-tail invoker of $e*$ must be found. First, we define the relation Invoker as follows:

$e_1 \in \texttt{Invoker}(e)$ iff

$$\begin{array}{lcl} e_1 & = & e/e_2 \\ e_1 & = & e_2/e \\ e_1 & = & e\, e_2 \\ e_1 & = & e_2\, e \\ e_1 & = & !e \\ e_1 & = & e* \\ e_1 & = & e? \\ e_1 & = & N (\text{where } N \text{ is a nonterminal with a rule} \\ & & N \leftarrow e). \end{array}$$

The last line in the above means that the Invoker(e) for an expression e may have several elements. Then, we define the following function H such that $H(e)$ represents the follower expressions of e (c indicates the child of e and V indicates visited nonterminals):

$$
\begin{array}{lcl}
H(e) & = & J(e_1, e, \emptyset) \cup \ldots J(e_n, e, \emptyset) \\
J(e*, c, V) & = & \emptyset \\
J(e?, c, V) & = & \emptyset \\
J(!e, c, V) & = & \emptyset \\
J(\&e?, c, V) & = & \emptyset \\
J(e : \texttt{"~~"}, c, V) & = & J(e_1, e, V) \cup \ldots\ J(e_n, e, V) \\
J(e : \texttt{[~]}, c, V) & = & J(e_1, e, V) \cup \ldots\ J(e_n, e, V) \\
J(e : ., c, V) & = & J(e_1, e, V) \cup \ldots\ J(e_n, e, V) \\
J(e : N, c, V)\ (N \in V) & = & \emptyset \\
J(e : N, c, V) & = & J(e_1, e, V \cup \{N\}) \\
 & & \cup \ldots\ J(e_n, e, V \cup \{N\}) \\
J(e : e_a\ e_b, e_b, V) & = & J(e_1, e, V) \cup \ldots\ J(e_n, e, V) \\
J(e : e_a\ e_b, e_a, V) & = & \{e_1\} \\
J(e : e_a \;/\; e_b, e_a, V) & = & \emptyset \\
J(e : e_a \;/\; e_b, e_b, V) & = & J(e_1, e, V) \cup \ldots\ J(e_n, e, V),
\end{array}
$$

where $e_1, \ldots, e_n \in \texttt{Invoker}(e)$.

For the expression e_1* and the set $X = H(e_1*)$, if size$(X) =$ 1 and the following condition holds,

$$
\begin{array}{c}
e_1 \notin nul \wedge e_2 \notin nul \wedge \\
disjoint(e_1, e_2)\ (X = \{e_2\})
\end{array}
$$

then the expression e_1* can be translated to:

$$(!(\texttt{ExtFIRST}(e_2))\ \uparrow\ e_1)*;.$$

The relation `Invoker` and the function H to find the follower expressions of e are similar to the relation $\texttt{Last}_{ss}$ and $\texttt{FOLLOW}_s$ in [18], respectively, but not the same as them.

For example, $\texttt{FOLLOW}_s(e) \neq H(e)$ where $(e/e_1)\ e_2$.

Additionally, we add the special translation rule to handle the case that $e_2 =!.$ where $X = \{e_2\}$. This pattern appears frequently in PEGs and the abovementioned translation rule cannot handle this pattern. The translation rule enable that the expression e_1* followed by !. to be translated to $(\texttt{\&}(.)\ \uparrow\ e_1)*$.

4.5 Limitations

Because AC-FIRST and AC-Repetition, like $LL(1)$ parsing, perform lookahead of only one terminal expression, these algorithms have limitation similar to it. That is, if the first terminal expressions of a choice can be prefix of the first terminals of another choice, these algorithms cannot insert a cut operator into the PEG. For example, in the following PEG, a cut operator cannot be inserted after ":" because lookahead of arbitrary terminal expressions is needed.

A ← [a-z]+ ":" ... / [a-z]+ ";";

4.6 Compaction of Lookahead Expressions

Lookahead expressions generated by AC-FIRST and AC-Repetition sometimes become quite large. For example, the VariableInitializer rule in Java PEG

VariableInitializer ← ArrayInitializer
/ Expression;

is translated to the following rule using AC-FIRST:

VariableInitializer ← !(
[0-9] / [1-9] / [A-Z] / [_$] / [a-z]
/ "synchronized" / ...) ↑
ArrayInitializer / Expression.

The right-hand side of the above rule is unnecessarily large because the lookahead expression !([a-z] / "synchronized") has the same meaning as !([a-z]). We implemented a method for the compaction of a lookahead expression to solve this problem. The key concept is that when $e_1 \leq e_2$, we can safely replace $!(e_1 \;/\; e_2)$ or $!(e_2 \;/\; e_1)$ with $!(e_1)$. Hence, if $e_i \leq e_j$, e_j can be removed from the lookahead expression $!(\ \ldots \;/\; e_i \;/\; \ldots \;/\; e_j \;/\; \ldots)$. In addition, if $e_j \leq e_i$, e_i is removed from it. Using this method, we can compact the expression !([a-z] / "synchronized") to !([a-z]).

4.7 Suppressing Excess Insertion of Cut Operators

Using our methods, unnecessary *cut* operators are occasionally inserted. Specifically, inserting a *cut* operator into an expression that expresses only fixed-size strings is always unnecessary. An expression that expresses only fixed-sized strings can be detected using the following function F:

$$
\begin{array}{lcl}
F(e_1 \;/\; e2,\ V) & = & F(e_1,\ V) \wedge F(e_2,\ V) \\
F(e_1\ e2,\ V) & = & F(e_1,\ V) \wedge F(e_2,\ V) \\
F(e*,\ V) & = & \text{false} \\
F(e?,\ V) & = & \text{false} \\
F(\&e, V) & = & F(e,\ V) \\
F(!e,\ V) & = & F(e,\ V) \\
F(N,\ V)\ \text{if}(N \in V) & = & \text{false} \\
F(N,\ V) & = & F(e,\ V \cup \{N\})\ \text{where}\ N \leftarrow e \\
F(e)\ \text{if}(e \in T) & = & \text{true.}
\end{array}
$$

The first line says that $e_1 \;/\; e_2$ expresses only fixed-size strings if both e_1 and e_2 express only fixed-size strings. The other lines can be interpreted in the same manner. Note that V, which is the second argument of F, represents visited nonterminals. V is used to detect a recursive expression. If $F(e)$ is true, it is wasteful to insert *cut* operators into e and such an insertion should be avoided.

5. Comparison with Related Work

In this section, we compare our methods with two related works. One is *Rats!*[10], a packrat parser generator which generates Java code. The other is *Mouse*[19], a PEG parser generator which also generates Java code.

Rats! implements several optimizations to improve execution performance and memory efficiency. However, despite these optimizations, parsers generated by *Rats!* require $O(n)$ space in parsing because of memoization. On the other hand, parsers generated by *Yapp* using our methods can parse an input in mostly constant space for memoization. Because our methods are orthogonal to those optimizations exploited in *Rats!*, we can combine the use of our methods and the optimizations to improve the memory efficiency. In fact, we could easily implement *Chunks* optimization in *Yapp*. It should also be possible to implement our method in *Rats!*.

Mouse is a PEG parser generator. Instead of memoizing all intermediate results, parsers generated by *Mouse* have a small, fixed-size cache. Therefore, *Mouse* does not require $O(n)$ space for memoization. But at the same time, *Mouse* does not guarantee linear-time parsing. Our method requires only almost constant space for memoization without sacrificing time linearity.

6. Evaluation

For the evaluation of our methods, First, we implemented our methods as a modification of our parser generator *Yapp*. *Yapp* is written in Java and generates packrat parsers in Java. Then, we generated optimized parsers using *Yapp*. To evaluate our methods, we compared the following parsers that don't construct abstract syntax trees (i.e. recognizers) in terms of heap size and execution performance:

- `AUTO`: generated by *Yapp* from grammars in which *cut* operators are inserted *automatically* using our methods.

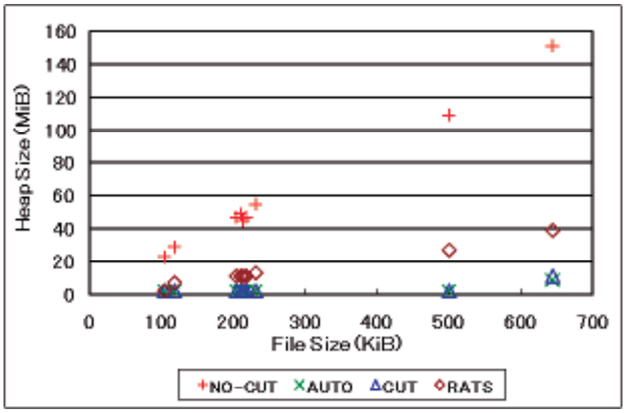


Figure 4. Minimum heap size in parsing Java programs

- CUT: generated by *Yapp* from grammars in which *cut* operators are inserted *manually*.
- NO-CUT: generated by *Yapp* from grammars in which *cut* operators are *not* inserted.
- RATS: generated by *Rats!* 1.14.3.

We used Java 1.4 PEG, JSON, and XML (subset) PEG as examples of grammars. Java is selected as a widely used programming languages. And JSON and XML is selected as widely used languages which sizes could be large in practice. We created Java 1.4 PEG[2] from a Parsing Expression Grammar for Java 1.5[16], XML PEG from the EBNF definition in Extensible Markup Language (XML) 1.0 (Fourth Edition)[5], and JSON PEG from the grammar on Annex A.8 of ECMA-262 (Fifth Edition) [11]. We selected 9 files with size greater than 100 KiB as inputs to the Java 1.4 parsers from Java programs generated from the repository of JavaCC grammars[3], and 38 files as inputs to the XML parsers, each having a size less than 2 MiB, from IJS-ELAN corpus Version 2.0[7]. And we selected 38 files that we translated the IJS-ELAN corpus XML files to JSON files using JSON in Java library[2] as inputs to the JSON parsers.

All evaluations are performed on Intel Core2 Duo 2.4GHz with 2GB RAM running JDK 1.6.0 (client VM) on Windows XP Professional.

6.1 Heap Size

To measure the heap size used in parsing inputs, we used the -Xms command line option, which sets the initial Java heap size, and the -Xmx command line option, which sets the maximum Java heap size. We carried out a binary search to determine the minimum heap size for which the parser can parse an input file without OutOfMemoryError. The results are shown in figure 4, 5, and 6.

Figure 4 and 6 show that AUTO and CUT can parse Java programs and JSON texts in mostly constant space regardless of the input size, in contrast to NO-CUT and RATS. The slight increase of heap sizes of AUTO and CUT in Java when the file size is 644 KiB is that there exists one large Java statement in the file and *cut* operators are not sufficiently inserted for such a statement. We can say that our methods are effective for the Java PEG and the JSON PEG from the result.

But in figure 5, AUTO exhibits almost the same memory consumption as NO-CUT and performs differently from CUT. An observation of the XML PEG suggests that there exists some *cut* operator instances which cannot be inserted automatically but can be inserted manually for the reason described in section 4.5.

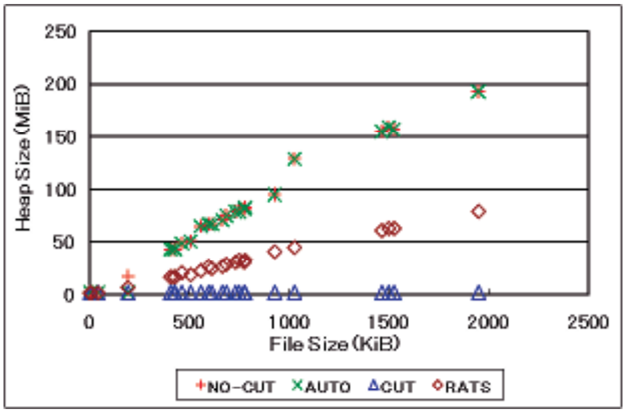


Figure 5. Minimum heap size in parsing XML files

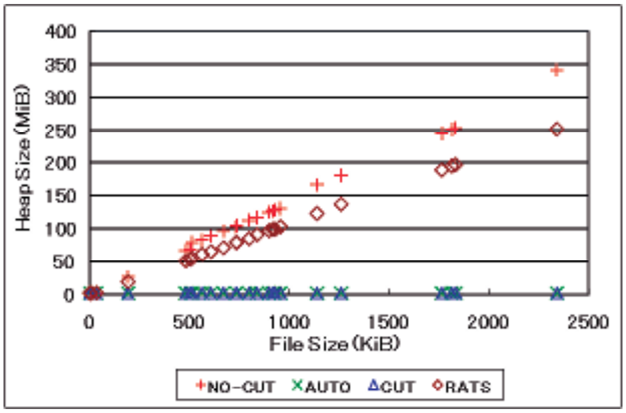


Figure 6. Minimum heap size in parsing JSON files

6.2 Speed

To measure how the speeds of the parsers change with the heap size, we again used the *-Xms* and *-Xms*. We measured the time in which parsers parse all input files successfully, repeated the evaluation 20 times, and selected the median as a result. The results are shown in figure 7, 8, and 9.

Figure 7 and 9 show that the speeds of AUTO and CUT in parsing Java programs and JSON texts are improved significantly as compared to NO-CUT. AUTO performs almost as good as CUT. That is, our methods do a fairly good job as compared to the manual insertion of *cut* operators.

In addition, figure 7 indicates that in Java AUTO and CUT are faster than RATS in a small heap size and figure 9 indicates that in JSON AUTO and CUT are far more faster than RATS. Supposedly, this result is due to a decrease in time for garbage collections. Unfortunately, as shown in figure 8, AUTO performes mostly the same as NO-CUT for XML for the same reason as the case of figure 5.

7. Conclusions

We proposed methods for automatic insertion of *cut* operators into a PEG. Using our methods, we can generate packrat parsers that require only almost constant space for memoization without

[2] We used Java 1.4 parser distributed with *Rats!* for RATS, because the grammar may be optimized for *Rats!*.

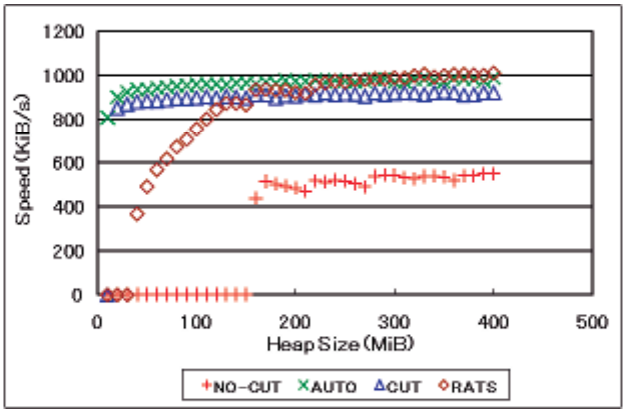


Figure 7. Speed for parsing Java programs. 0 on the y-axis indicates that the parser could not parse all inputs successfully.

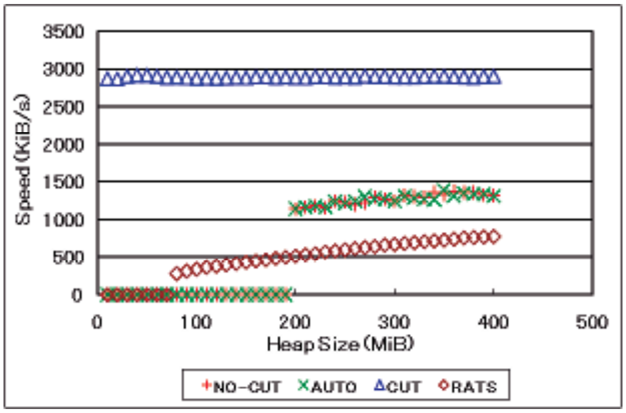


Figure 8. Speed for parsing XML files.

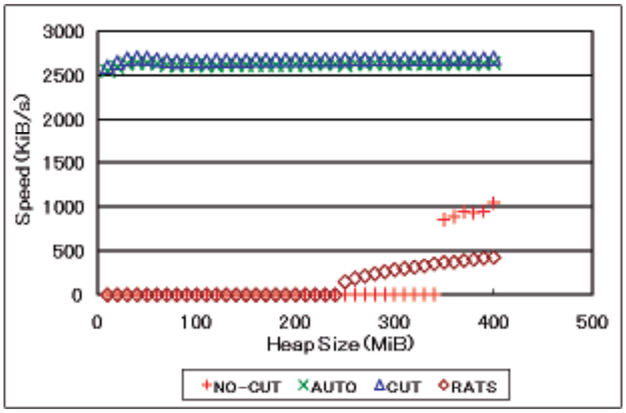


Figure 9. Speed for parsing JSON files.

manual rewriting of the grammar. Packrat parsers are considered unsuitable for parsing large inputs. However, using our methods, packrat parsers can handle large files practically. This is the main contribution of our study. Experimental evaluations suggest that our methods are effective for a Java PEG and a JSON PEG but are unfortunately ineffective for an XML PEG. We believe that this problem can be solved by some extensions to our methods (e.g. increasing the number of lookahead nonterminal expressions like $LL(k)$). We intend to address the problem in future work.

Acknowledgments

We wish to thank the anonymous reviewers for their helpful comments.

References

[1] *javacc: JavaCC Home*, . https://javacc.dev.java.net/.

[2] *JSON in Java*. http://www.json.org/java/.

[3] *A repository of JavaCC grammars*, . https://javacc.dev.java.net/servlets/ProjectDocumentList?folderID=110.

[4] R. Becket and Z. Somogyi. Dcgs + memoing = packrat parsing but is it worth it? In *Practical Aspects of Declarative Languages*, January 2008.

[5] T. Bray, J. Paoli, and C. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C Recommendation*, August 2006.

[6] A. Colmerauer and P. Roussel. The birth of prolog. In *The second ACM SIGPLAN conference on History of programming languages*, pages 37–52. ACM Press, 1993.

[7] T. Erjavec. The ijs-elan slovene-english parallel corpus. *International Journal of Corpus Linguistics*, 7(1):1–20, 2002.

[8] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming*, October 2002.

[9] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Symposium on Principles of Programming Languages*, January 2004.

[10] R. Grimm. Better extensibility through modular syntax. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 19–28, 2006.

[11] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), fifth edition, December 2009.

[12] S. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[13] K. Mizushima, A. Maeda, and Y. Yamaguchi. Improvement technique of memory efficiency of packrat parsing. In *IPSJ Transaction on Programming Vol.49 No. SIG 1(PRO 35) (in Japanese)*, pages 117–126, 2008.

[14] I. S. Organization. *Syntactic metalanguage – Extended BNF*, 1996. ISO/IEC 14977.

[15] T. J. Parr and R. W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25:789–810, 1994.

[16] R. Redziejowski. *Parsing Expression Grammar for Java 1.5*. http://www.romanredz.se/papers/PEG.Java.1.5.txt.

[17] R. Redziejowski. Some aspects of parsing expression grammar. In *Fundamenta Informaticae 85, 1-4*, pages 441–454, 2008.

[18] R. Redziejowski. Applying classical concepts to parsing expression grammar. In *Fundamenta Informaticae 93, 1-3*, pages 325–336, 2009.

[19] R. Redziejowski. Mouse: from parsing expressions to a practical parser. In *Concurrency Specification and Programming Workshop*, September 2009.