Packrat Parsing: A Literature Review

Manish M. Goswami
Research Scholar,
Dept. of Computer Science and Engineering,
G.H.Raisoni College of Engineering,
Nagpur, India

Dr. M.M. Raghuwanshi,
Professor,
Department of Computer Technology,
YCCE,
Nagpur, India

Dr. Latesh Malik,
Professor,
Dept. of CSE,
G.H.Raisoni College of Engg.,
Nagpur, India

***Abstract*—Packrat parsing is recently introduced technique based upon expression grammar. This parsing approach uses memoization and ensures a guarantee of linear parse time by avoiding redundant function calls by using memoization. This paper studies the progress made in packrat parsing till date and discusses the approaches to tackle this parsing process efficiently. In addition to this, other issues such as left recursion, error reporting also seems to be associated with this type of parsing approach and discussed here the efforts attempted by researchers to address this issue. This paper, therefore, presents a state of the art review of packrat parsing so that researchers can use this for further development of technology in an efficient manner.**

***Keywords*—Parsing Expression Grammar; Packrat Parsing; Memoization; Backtracking**

I. INTRODUCTION

Parsing consists of two processes: lexical analysis and parsing. The job of the lexical analysis is to break down the input text (string) into smaller parts, called tokens. The lexical analyzer then sends these tokens to the parser in sequence. During parsing, the parser takes the help of a grammar to decide whether to accept the input string or reject .i.e. whether it is a subset of the accepting language or not. A set of grammar rules or productions is used to define the language of grammar. Each production can then, in turn, compose of several different alternative productions. These productions guide the parser throughout the parsing to determine whether to accept the input string or to reject it. Top-down parsing is a parsing strategy that attempts left-to-right leftmost derivation (LL) for the input string. This can be achieved with prediction, backtracking or a combination of the two. LL(k) top-down parser makes its decisions based on lookahead, where the parser attempts to ”look ahead” k number of symbols of the input string. A top-down parser that uses backtracking instead evaluates each production and its choices in turn; if a choice/production fails the parser backtracks on the input string and evaluates the next choice/production, if the choice/production succeeds the parser merely continues. The bottom-up parsing is a parsing method that instead attempts to perform a left-to-right rightmost derivation (LR) in reverse of the input string. Shift-reduce parsing is widely used bottom-up parsing technique. A shift-reduce parser uses two different actions during parsing: shift and reduce. A shift action takes a number of symbols from the input string and places them on a stack. The reduce action reduces the symbols on the stack based on finding a matching grammar production for the symbols. The decisions regarding whether to shift or reduce are done based on lookahead. Several different parsing techniques have been developed over the years, both for parsing ambiguous and unambiguous grammars. One of the latest is packrat parsing [5]. Packrat parsing is based upon a top-down recursive descent parsing approach with memoization that guarantees linear parse time. Memoization employed in the packrat parsing eliminates disadvantage of conventional top-down backtracking algorithms which suffer from exponential parsing time in the worst case. This exponential runtime is due to performing redundant evaluations caused by backtracking. Packrat parsers avoid this by storing all of the evaluated results to be used for future backtracking eliminating redundant computations. This storing technique is called memoization which ensures guaranteed linear parsing time for packrat parsers. The memory consumption for conventional parse algorithms is linear to the size of the maximum recursion depth occurring during the parse. In the worst case it can be the same as the size of the input string. In packrat parsing, the memory consumption for a packrat parser is linearly proportional to the size of the input string. Packrat parsing is based upon parsing expression grammars (PEGs) which have the property of always producing unambiguous grammars. It has been proven that all LL(k) and LR(k) grammars can be rewritten into a PEG[7]. Thus, packrat parsing is able to parse all context-free grammars. In fact, it can even parse some grammars that are non-context-free [7].

Another characteristic of packrat parsing is that it is scannerless i.e. a separate lexical analyzer is not needed. In packrat parsers, they are both integrated into the same tool, as opposed to the Lex[10]/Yacc[9] approach where Lex is used for the lexical analysis and Yacc for parsing phase of the compiler.The founding work for packrat parsing was carried out in 1970 by A. Birman et. al.[4]. Birman introduced a schema called the TMG recognition schema (TS). Birman’s work was later refined by A. Aho and J. Ullman et. al.[2], and renamed into generalized top-down parsing language (GTDPL). This was the first top-down parsing algorithm that was deterministic and used backtracking. Due to deterministic nature of resulting grammar they discovered that the parsing results could be saved in a table to avoid redundant computations. However, this approach was never put into practice, due to the limited amount of main memory in computers at that time [10,14]. Another characteristic of GTDPL is that it can express any LL(k) and LR(k) language, and even some non-context-free languages[2,3]. Rest of the paper consists of introduction to Parsing Expression grammar

along with discussion on its properties followed by the work carried out by researchers in this area. Finally the paper focuses upon the open problems in packrat parsing and concluded with future work.

II. PARSING EXPRESSION GRAMMAR

As an extension to GTDPL and TS, Bryan Ford introduced PEGs [7]. CFGs (which were introduced mainly for usage with natural language [7]) may be ambiguous and thereby either (1) produce multiple parse tree's, which is not necessary due to only one is needed, and (2) produce a heuristically chosen one, which might not even be correct[8]. However, one of the characteristics of PEGs is that they are by definition unambiguous and thereby provides a good match for machine-oriented languages (since programming languages supposed to be deterministic). It is also shown that PEGs, similar to GTDPL and TS, can express all LL(k) and LR(k) languages, and that they can be parsed in linear time with the memoization technique [7].

A. Definitions and Operators

PEGs, as defined in [7], are a set of productions of the form A <- e where A is a nonterminal and e is a parsing expression. The parsing expressions denote how a string can be parsed. By matching a parsing expression e with a string s, e indicates success or failure. In case of a success, the matching part of s is consumed. If a failure is returned, s is matched against the next parsing expression. Together, all productions form the accepting language of the grammar. The following operators are available in PEG productions: [7,14]

Ordered choice: $e_1/.../e_n$, expression e1,...,en is evaluated in this order, to the text ahead, until one of them succeeds and possibly consumes some text. If one of the expressions succeeded, indicate success. Otherwise indicate failure and input is not consumed.

Sequence: $e_1,...,e_n$, expressions $e_1,...,e_n$, is evaluated in this order, to consume consecutive portions of the text ahead, as long as they succeed. If all succeeded, success is indicated. Otherwise indicate failure and input is not consumed.

And predicate: &e, if expression e matches the text ahead; indicate success otherwise indicate failure. Text is not consumed.

Not predicate:!e, if expression e matches the text ahead, failure is indicated; otherwise, indicate success. Do not consume any text.

One or more: e+, expression e is repeatedly applied to match the text ahead, as long as it succeeds. Matched Text is consumed if any and success is indicated if there is at least one match. Otherwise failure is indicated.

Zero or more: e*, As long as expression e matches text ahead it is applied repeatedly and consumed the matched text (if any). Always report success.

Zero or one: e?, if expression e matches the text ahead, consume it. Always report success.

Character class: [s], character ahead is consumed if it appears in the sting s and success is indicated. Otherwise failure is indicated.

Character range: [c1- c2], if the character ahead is one from the range c1 through c2, consume it and indicate success. Otherwise indicate failure.

String:'s', if the text ahead is the string s, consume it and success is indicated. Otherwise failure is indicated.

Any character: (dot), if there is a character ahead, consume it and indicate success. Otherwise (that is, at the end of input) indicate failure.

B. Ambiguity

The unambiguousness of a PEG comes from the ordered choice property. The choices in CFGs are symmetric, i.e., the choices need not be checked in any specific order. However, the choices in a PEGs are asymmetric, i.e., the ordering of the choices determines in which order they are tested. For a PEG the first expression that matches are always chosen. This means that a production such as A<-a/aa is perfectly valid and unambiguous. However, it only accepts the language {a} on the contrary, a CFG production A->a|aa is ambiguous but accepts the language {a,aa}.

A traditional example that is hard to express with the use of CFGs is the dangling else problem. Consider the following if-statement:

if cond then if cond then statement else statement

This statement can be matched in the following two ways:

if cond then (if cond then statement else statement)
if cond then (if cond then statement) else statement

If the intended matching is the former of the two (in fact, this is how it is done in the programming language C[8]) then the following PEG production is sufficient:

Stmt<- 'if' Cond 'then' Stmt 'else' Stmt

/'if' Cond 'then' Stmt /...

Note: Matching the outermost if with the else-clause is believed not to be possible with a PEG. However, no source to either prove or contradict this statement was found.

Discovering if a CFG production is ambiguous is sometimes a non-trivial task. Similarly, choosing the ordering of two expressions in a PEG production without affecting the accepting language is not always straightforward [7].

C. Left Recursion

Left recursion is when a grammar production refers to itself as its left-most element, either directly or indirectly. Similar to the conventional LL(k) parsing methods, left recursion proves to be an issue for PEGs, and therefore a problem also for packrat parsing [10,7]. Consider the following alteration of the production:

A<-Aa / a

For a CFG, this modification is not a problem. However, for a PEG, parsing of nonterminal A requires testing that A matches, which requires testing that A matches etc, producing

an infinite recursion. However, it was early discovered that a left-recursive production can always be rewritten into an equivalent right-recursive production [1], and thus making it manageable for a packrat parser. However, if there is indirect left recursion involved, the rewriting process may become fairly complex.

D. Syntatic Predicates

PEGs allow the use of the syntactic predicates! and &. Consider the following grammar production:

A <- !B C

Every time this production is invoked it needs to establish if the input string matches a B and if it does, signal a failure. If there is no match the original input string is compared with the nonterminal C. This ability to "look ahead" an arbitrary amount of characters combined with the selective backtracking gives packrat parsers unlimited lookahead [10, 6, 7, 13].

E. Memoization

The introduction of memoization was treated as a machine learning method in 1968 by D. Michie et. al. [11]. By storing calculated results, the machine "learned" it. The next time it was asked for the same result, the machine merely"remembered" it by looking up the previously stored result. The storage mechanism used was a stack. This makes the look-up process become linear. However, insertions of results are constant; they are merely pushed on top of the stack. In packrat parsing, the results are instead stored in a matrix or similar data structure that provides constant time look-ups (when the location of the result is already known) and insertions [10]. For every encountered production this matrix is consulted; if the production has already occurred once the result is thereby already in the matrix and merely needs to be returned; if not, the production is evaluated and the result is both inserted into the matrix and returned. Conventional recursive descent parsers that use backtracking may experience exponential parsing time in the worst case. This is due to redundant calculations of previously computed results caused by backtracking. However, memoization avoids this problem due to the fact that the result only needs to be evaluated once. This gives packrat parsing a linear parsing time in relation to the length of the input string (given that the access and insertion operations in the matrix are done in constant time). Let us look at the following trivial PEG, taken from [10]:

Additive <- Multitive '+' Additive / Multitive

Multitive <- Primary '*' Multitive / Primary

Primary <- '(' Additive ')' / Decimal

Decimal <- [0-9]

With this grammar and the input string 2*(3+4) the following memoization matrix can be produced:

The columns correspond to each position of the input string; the rows correspond to each of the parsing procedures. To make it clear that the rows are in fact procedures they have been given a prefix 'p'. Each cell contains either a number that represents how much of the input string that have been consumed by a previous call to the procedure, or the cell

pAdditive	7	-1	5	3	-1	1	-1	-1
pMultitive	7	-1	5	1	-1	1	-1	-1
pPrimary	1	-1	5	1	-1	1	-1	-1
pDecimal	1	-1	-1	1	-1	1	-1	-1
input	'2'	'*'	'('	'3'	'+'	'4'	')'	$

A Matrix Containing the Parsing Results of the Input String 2*(3+4) contains a '-1' which indicates a failed evaluation. For instance, if backtracking occurs at input position four (where the number '3' is present) and procedure pAdditive is called, the parser first checks if a previous computation is stored in the storage matrix. In this case the number 3 is stored and thereby the parser immediately knows that it can advance three steps on the input string and end up at the seventh character of the input string. If the stored value is '-1' the parser knows that a previous call resulted in a failed parse and can thus avoid continuing with the procedure call and instead return a failure response to the calling function. This illustrates how redundant computations and thereby also potential exponential parsing times are avoided with the help of memoization. Calculating the whole matrix in Table1 would be unnecessary since many of the cells are not needed. The idea behind packrat parsing is not to evaluate all of the cells in the parsing matrix, only the results that are needed [10]. This effectively reduces the amount of memory space required during parsing.

F. Scannerless

Conventional parsing methods are usually divided into two phases: the lexical analysis phase and the parsing phase. The tokenization phase is called lexer, lexical analyzer, tokenizer or scanner. The lexical analysis splits the input string into tokens which hopefully corresponds to the permitted terminals of the grammar. This lexical analysis is important for conventional parsers due to their inability to refer to nonterminals for lookahead decisions [6]. Thus, the parser treats the tokens acquired from the lexical analysis as if they were terminals.

Packrat parsers can on the other hand be scannerless, which means that it requires no lexical analysis. When a scannerless packrat parser evaluates different alternatives, it can rely on already evaluated results. This effectively makes a packrat parser able to use both terminals and nonterminals during lookahead [10, 6]. Large parts of the code base of programs may consist of white spaces, comments and other irrelevant information that is not needed for the semantic analysis. A lexical analyzer can effectively disregard such information by simply opting not to create any tokens for them, thus no specific productions for white spaces and/or comments need to be included in the grammar specification of the parser. For a packrat parser that does not use a lexical analyzer, however, this is not the case. A packrat parser that uses no lexical analyzer, the white spaces and comments need to be incorporated into the productions of the grammar.

As previously mentioned, the conventional parsers treat the created tokens given by the lexical analyzer as if they were terminals, and between each token the lexical analyzer disregards any white spaces or comments. To achieve this

effect for a packrat parser, a production to manage white space or comments can be created and used after each terminal symbol of the grammar. For instance, the grammar for recognizing arithmetic expression altered in the following way to be able to correctly handle white spaces inside an arithmetic expression:

Additive<- Multitive '+' Spaces Additive / Multitive
Multitive<-Primary '*' Spaces Multitive / Primary
Primary <- '(' Spaces Additive ')' Spaces /Decimal
Decimal <- [0-9] Spaces
Spaces <- (' ' / ' \t' / ' \n' / ' \r')*

This grammar is now able to parse an arithmetic expression such as: 2 * (3+ 4).

III. Literature Survey

PEGs are a recently introduced technique for describing grammars by Ford in [5] with implementation of the packrat parser. Theory is based upon strong foundations. Ford [18] showed how PRGs can be reduced to TDPLs long back in the 1970s. It was shown by Roman[24] that primitive recursive-descent parser with limited backtracking alongwith integrated lexing can be used for parsing Java 1.5 where requirement is of moderate performance. PEG is not good as a language specification tool as shown in [25]. The characteristic of a specification is that what it specifies is clearly to be seen. But this is, unfortunately, not valid for PEG. Further it gives reasonable performance when C grammar is slightly modified and also in [16] he studied that classical properties like FIRST and FOLLOW where he demonstrated those can be redefined for PEG and can be obtained even for a large grammar. FIRST and FOLLOW are used to define conditions for choice and iteration that are similar to the classical LL(1) conditions, although they have a different structure and semantics. This is different from classical properties like FIRST and FOLLOW where letters are terminal expressions, which may mean sets of letters, or strings. Checking these conditions gives an idea of useful information like the absence of reprocessing or language hiding which is helpful in locating places that need further examination. The properties FIRST and FOLLOW are kind of upper bounds, and conditions using them are sufficient, but not necessary which may results in false warnings In [17] a virtual parsing machine approach is proposed for implementing PEG which is can be applied to pattern matching. Each PEG is converted directly into its equivalent corresponding program Virtual parsing machine then using scripting language excutes the translated program. Creation and composition of new programs are done on fly.

In [7] Robert grimm parsing technique made practical for object-oriented languages. This parser generator employs simpler grammar specifications. Error reporting is also made easy by this parser generator and shown to be better performing parsers through aggressive optimizations.

In [19] cut operator was introduced to parsing expression grammars (PEGs) and when applied to PEG on which packrat parsing is based. Disadvantage is largely addressed with this approach. Concept of cut operator was borrowed from Prolog [6]. It introduces degree of controlling backtracking. An efficient packrat parser can be developed avoiding unnecessary space for memorization by inserting cut operators into a PEG grammar at appropriate places. To show effectiveness and usefulness of cut operators, a packrat parser generator called Yapp was implemented and used. It accepts Parsing Expression Grammar marked with cut operator. The experimental evaluations showed that the packrat parsers generated using grammars with cut operators inserted can parse Java programs and subset of XML files in mostly constant space, unlike conventional packrat parsers. In [12] automatic insertion of cut operators was proposed that achieves the same effect. In these methods, a statistical analysis is made of a PEG grammar by parser generator in order to find the places where the parser generator can insert cut operators without changing the meaning of the grammar and cut operators are inserted at these identified points. Definite clause grammar rules and memoing can be a possible combination for implementation of packrat parser as shown in [20] .Further it points out that packrat parsing may degrade its performance over plain recursive descent with backtracking, but memoing the recognizers of just one or two nonterminals can sometimes give reasonable performance.

Warth [21] tweaked memoization approach used by packrat parser because of which left-recursion even indirectly or mutually was supported. But some experiments were conducted out to show that this is not the case for typical uses of left recursion. In [8] Coq formalization of the theory of PEGs is proposed and with this as a foundation a formally verified parser interpreter for PEGs, TRX is developed. This gives rise to writing a PEG, together with its semantic actions, in Coq and then a parser can be extracted from it a parser with total correctness guarantees. This ensures that the parser will terminate on all kind of inputs and produces output as a parsing results correct with respect to the semantics of PEGs.

In [27] concept of elastic sliding window is used and it is based upon the observation of worst longest backtrack length. Particularly author noted that if a window in the form of small memorization table slides and covers the longest backtrack then redundant calls are avoided since the storage is sufficient enough to store all the results. Practically, it is difficult to get the longest backtrack before parsing as it is runtime entity. Here window is approximated from empirical investigation and if needed may be expanded during parsing.

[28] introduces derivative parsing with memorized approach algorithm for recognition of PEG. Main problem in this algorithm since derivative parsing attempts all possible parses concurrently is to identify which constructs exactly in the current parse tree can match against or consume the current character. This problem is solved by using concept of a backtracking generation (or generation) as a means to take into account for backtracking choices in the process of parsing. Execution of the algorithm is found to be in worst case quadratic time and cubic space. However, it is stressed in this paper that due to the limited amount of backtracking and recursion in grammars when put in practical use and input, practical performance may be nearer to linear time and constant space and requires experimental validation for the same which is in progress. Table II summarizes the comparative study of major packrat parser generators.

TABLE I. COMPARISON OF VARIOUS PACKRAT PARSER GENERATORS

Name of Parser	Language used for implementation	Working Principle	Memory Utilization	Execution Time
YAPP	Object oriented Language Java	Optimized Packrat Parser with CUT operator	Memory requirement has been cut down as cut operator reduces redundant calls	Moderate amount of parsing time required
RATS	Object oriented Language Java	Packrat Parser with some aggressive optimization	Less memory space as some aggressive optimizations used	Moderate amount of parsing time required
Mouse	Object oriented Language Java	Straightforward Recursive descent Parser implemented using PEG with no memorization support	Least amount of storage used	Amount of parsing time is highest among all the parsers as repeated backtracking is not avoided
PAPPY	Functional Programming Language Huskell	Basic Pakrat Parser with memoization	Use of significant storage space	Moderate amount of parsing time required
Nez	Object oriented Language Java	Packrat Parser implemented with elastic sliding window concept	Significant and least amount of storage is used among all packrat parser which use memoization	Moderate amount of parsing time required

IV. OPEN PROBLEMS IN PACKRAT PARSING

A. Memoization

One of the drawbacks with using packrat parsing is the additional memory consumption when compared with conventional parsing techniques. A tabular approach where the whole m*n matrix is evaluated would require Θ(mn) space, where m is the amount of nonterminals and n is the size of the input string. However, for a packrat parser, only the required cells in the matrix are evaluated, and these cells are directly related to the input string and not the nonterminals of the grammar [22]. In other words, adding more productions to the grammar may not necessarily increase the storage consumption while increasing the size of the input string will always increase the memory consumption. This makes the required size of the memoization matrix for a packrat parser be proportional only to the size of the input string, thus O(n). Even if the space consumption is upper-bounded by the input string and can therefore be written as O(n) there is a ”hidden constant multiple” of n [22].This is because there can be more than n elements in the produced memoization matrix. Conventional LL(k) and LR(k) parsing algorithms only require storage space proportional to the maximum recursion depth that occurs for the given input. This causes these conventional algorithms to have the same worst case memory requirement as a packrat parser. However, a packrat parser is also lower bounded by n and this worst case behavior for LL(k) and LR(k) parsers rarely occurs [10, 22]. In fact, the maximum recursion depth is usually significantly smaller than the size of the input string [22].

B. Maintaining States

A parser for the programming languages C and C++ requires that the parser is able to maintain a global state. The reason is the nature of typedef’s for the two languages. The parser needs to be able to distinguish whether the input is a typedef symbol, an object, a function or an enum constant, and change the global state accordingly if the meaning of a specific token changes. For instance, the following C code requires this feature: [7]

```
T(*b)[4]
```

By only looking at this snippet, the parser has no way of knowing whether T refers to a function call or a typedef name, it is context-sensitive. If it is a function call, the snippet corresponds to accessing the fifth element of the resulting call to function T with the pointer b as input parameter. If T instead is a typedef, the snippet corresponds to b pointing to an array consisting of four elements of type T.

C or C++, however, can still be parsed with a packrat parser that changes its state whenever a variable changes its type [8]. This is because of the requirement that the type of a variable needs to be declared before its usage and therefore no parsing information prior to a definition of a variable is lost. This way, a separate symbol table can be constructed during the parse which keeps track of the type for different tokens. However, for the general case of context-sensitive grammars, packrat parsers may experience exponential parsing time and memory consumption. This is because during parsing a packrat parser assumes that an already evaluated cell of the result matrix is the correct result, and that this value will not have to change. But if a state change occurs the result matrix may have to be re-evaluated to ensure a correct result during backtracking. This can potentially break the guaranteed linear time characteristic due to cells being evaluated multiple times [22].

C. Left recursion:-

Left recursion is an issue in PEG and solution is proposed for the same in [11] by Wrath et al. But this approach fails for some PEGs as shown by Tratt[37] .The solution works for a safe subset of left-recursive PEGs with this approach. By extending this algorithm where allowing left-recursive rules with definite right-recursion to work as expected. In order to parse right-recursive PEGs safely, a number of subtle issues need to be addressed, and the set of right-recursive PEGs safely parseable is less than might originally have been hoped for. Next step obviously is to extend the solutions presented in this paper to tackle with indirect left and indirect right recursion. But this may be quite challenging and may impose further restrictions on valid PEGs. Therefore, this gives rise to a open problem: are PEGs really safe for left-recursion?

D. Error Reporting:-

One important property of parser is to provide good syntax error support. For example, if user enters invalid expression

and to recover from it, it is necessary for parser needs to know if it is parsing an array index or, say, an assignment. It is preferable that the parser should resynchronize by skipping ahead to a token. In the later case, it should skip to a; token since top-down parsers maintains a rule invocation stack and is able to report things like invalid expression in array index. Ambiguous context poses a problem before packrat parser since they are always speculating. In practice, recovery from syntax errors cannot be possible because they cannot detect errors until they have seen the entire input.

E. Specification Tool:-

PEG is looked at as a advanced tool for describing syntax and considered to be better than CFGs and regular expressions. The reason behind this is cited as grammar is unambiguous. But though it is an unambiguous specification of a parser, the language specified by it is whatever that parser happens to accept. But the language we want is easily seen? "Specification" means its meaning must be clear to a human reader. "Prefix capture" in PEG is not immediately visible which the main pitfall is. In addition to left recursion, detection of prefix capture is inevitable for any usable parser generator for PEG. To make sure the grammar defines the language required are there any other conditions that must be detected? This gives rise to think in terms of Parsing Expressions. This raises an argument here that BNF make it to understand easier because it better reflects the working of human mind, or because we are using it since long time?

V. Conclusion and Future Work

In this paper, compressive review is taken about the packrat parsing introduced by Ford in 2002. It presents the state of the art about the development and research about packrat parsing based upon parsing expression Grammar. Specifically this type of parsing is presented by using memoization technique but subsequently it is this memoization which is shown to be hindrance from applying this parsing technique though it guarantee linear time of execution by avoiding redundant calls. Therefore authors focused on this problem so that wide use of this technique is feasible. In addition to this other issues such as left recursion, error reporting also seems to be associated with this type of parsing approach and discussed here about the initiatives made by researchers to address this issue. Future work may be to apply these techniques to wide variety of grammars so that authenticities of these techniques are to be endorsed.

References

[1] Alfred V. Aho, Ravi Sethi, and Je_rey D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Alfred V. Aho and Je_rey D. Ullman. The Theory of Parsing, Translation, and Compiling.Upper Saddle River, NJ, USA, 1972. Prentice-Hall, Inc.

[3] A. Birman and J. D. Ullman. Parsing Algorithms with Backtrack. Journal of Information and Control, 23(1):1{34, 1973.

[4] Alexander Birman. The TMG Recognition Schema. PhD thesis, Princeton University,Department of Electrical Engineering, Princeton, NJ, USA, 1970.

[5] B. Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time - Functional Pearl. In Proceedings of the ACM SIGPLAN International Conference on Functional Programming,ICFP, pages 36{47, New York, NY, USA, 2002. ACM.

[6] Bryan Ford. Packrat Parsing: A Practical Linear-time Algorithm with Backtracking. Master'sthesis, Massachusetts Institute of Technology, Department of Electrical Engeneering and Computer Science, Cambridge, MA, USA, 2002.57

[7] Bryan Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation.In Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Program-ming Languages, POPL '04, pages 111{122, New York, NY, USA, 2004. ACM.

[8] Robert Grimm. Better Extensibility Through Modular Syntax. In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06, pages 38{51, New York, NY, USA, 2006. ACM.

[9] Stephen C Johnson. Yacc: Yet Another Compiler-Compiler. Bell Laboratories Murray Hill, NJ, 1975.

[10] M. E. Lesk and E. Schmidt. UNIX Vol. II. chapter Lex; a Lexical Analyzer Generator,pages 375{387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[11] D. Michie. Memo Functions and Machine Learning. Journal of Nature, 218:19{22, 1968.

[12] Kota Mizushima, Atusi Maeda, and Yoshinori Yamaguchi. Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space. In Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '10, pages 29{36, New York, NY, USA, 2010. ACM.

[13] Terence J. Parr and Russell W. Quong. Adding Semantic and Syntactic Predicates To LL(k): Pred-LL(k). In Proceedings of the 5th International Conference on Compiler Con-struction, CC '94, pages 263{277, London, UK, UK, 1994. Springer-Verlag.

[14] R. R. Redziejowski. Parsing Expression Grammar as a Primitive Recursive-Descent Parser with Backtracking. Journal of Fundamenta Informaticae, 79(3-4):513{524, 2007.

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

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

[17] S. Medeiros and R. Lerusalimschy : A Parsing Machine for PEGs. In Proc. PEPM, ACM (January 2009) 105-110.

[18] 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.

[19] 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.

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

[21] Warth, A., Douglass, J., Millstein, T.: Packrat parsers can support left recursion. In: Proc. PEPM, ACM (January 2008) 103-110.

[22] Warth, A., Piumarta, I.: OMeta: an object-oriented language for pattern matching. In: Proc. Dynamic Languages Symposium, ACM (2007) 11-19.

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

[24] R. Redziejowski. Parsing Expression Grammar for Java 1.5.

[25] http://www.romanredz.se/papers/PEG.Java.1.5.txt.

[26] Adam Koprowski and Henri Binsztok. TRX: A formally veri_ed parser interpreter. In Proceedings of the 19th European Symposium on Programming (ESOP '10), volume 6012 of Lecture Notes in Computer Science, pages 345-365, 2010.

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

[28] AaronMoss: Derivatives of Parsing Expression Grammars. CoRR abs/1405.4841 (2014)