© 2015 IJSRSET | Volume 1 | Issue 5 | Print ISSN : 2395-1990 | Online ISSN : 2394-4099
Themed Section: Engineering and Technology

A Survey of Packrat Parser

Amit Vishwakarma[1], Manish M. Goswami[2], Priyanka Gonnade[3]
[1,3]Department of CSE, Rajiv Gandhi College of Engineering and Research, Nagpur, India
[2] Department of IT, Rajiv Gandhi College of Engineering and Research, Nagpur, India

ABSTRACT

Two recent developments in the field of formal languages are Parsing Expression Grammar (PEG) and packrat parsing. The PEG formalism is similar to BNF, but defines syntax in terms of recognizing strings, rather than constructing them. It is, in fact, precise specification of a backtracking recursive-descent parser. Packrat parsing is a general method to handle backtracking in recursive descent parsers. It ensures linear working time, at a huge memory cost. This paper begins with discussion of PEG and packrat parsing introduced by Bryan Ford Followed by various approaches over improvement of packrat parsing to reduce the memory requirement. This paper also describes the approaches to handle the left-recursion problem for PEG. The described Approaches handle the direct and indirect left-recursion problem for PEG. The paper concludes with the application of packrat parsing and throws a light on future scope in packrat parsing.

Keywords: PEG; Recursive Descent Parser; Packrat Parser; TDPL

I. INTRODUCTION

Parsing is the act of discovering the structure of text with respect to a particular grammar and parser is a program to facilitate this parsing process. In order to create a parser for a particular language, or even just to reason formally about what kinds of strings are meaningful or well-formed in that language, we must have a way of expressing and understanding the language's syntactic structure. For this purpose we commonly use a grammar, which is a concise representation of the structure of one language, expressed in another (ideally very small and simple) language. Being able to express the syntactic structure of a language concisely with a grammar is especially important for programming languages and other languages expressly designed for precision and machine-readability, because grammars can be used to reason about the properties of a language mathematically or with the help of mechanical tools.

The most common type of grammar in use today is the context-free grammar (CFG), typically expressed in the ubiquitous Backus-Naur Form (BNF). A context-free grammar essentially specifies a set of mutually recursive rules that describe how strings in the described language may be written. Each rule or production in a CFG specifies one way in which a syntactic variable or nonterminal can be expanded into a string. Bottom up parsing recognizes the smallest constructs first by applying productions to group tokens, then grouping those constructs into larger constructs and top down does in reverse way. Parsing algorithms such as LR (k) and LL (k) parsing were developed alongside the first generation of high level programming languages to parse subsets of the full class of CFGs. By limiting the class of parseable languages, such algorithms are both time and space efficient, considerations that were of huge practical importance given the performance limitations of hardware available at the time.

Another method of expressing syntax formally is through a set of rules describing how the strings in a language are to be read rather than written. This approach is called recursive descent parsing. Recursive-descent parsers have been around for a while. Already in 1961, Lucas [16] suggested the use of recursive procedures that reflect the syntax of the language being parsed. His design did not allow backtracking; an explicit assumption about the syntax was identical to

what later became known as LL (1). The great advantage of recursive-descent parsers is transparency: the code closely reflects the grammar, which makes it easy to maintain and modify. However, manipulating the grammar to force it into the LL (1) mold can make the grammar itself unreadable. The use of backtracking removes the LL (1) restriction. Complete backtracking, meaning an exhaustive search of all alternatives, may require an exponential time. A reasonable compromise is limited backtracking, also called "fast-back" in [17]. In that approach, we discard further alternatives once a sub-goal has been recognized. Limited backtracking was adopted in at least two of the early top-down designs: the Atlas Compiler of Brooker and Morris [18, 19], and TMG (the TransMoGrifier) of McClure [20]. The syntax specification used in TMG was later formalized and analyzed by Birman and Ullman [21, 22]. It appears in [23] as "Top-Down Parsing Language" (TDPL) and "Generalized TDPL" (GTDPL). TDPL was developed at around the same time most of the classic CFG parsing algorithms were invented, but at that time it was used only as a formal model for the study of certain top-down parsing algorithms. The speed of modern computers means that relatively inefficient approaches to parsing are now often practical. For example, Earley's algorithm [24] can parse the entire class of CFGs; while it is O (n3), even a simple implementation can parse in the low thousands of lines per second [25]. For many people, parsing is a solved problem: there are a wide variety of well understood algorithms, with a reasonable body

THOUGH THE PARSING PROBLEM IS SUPPOSED TO BE SOLVED, COMPILER DESIGNERS STILL FACE SOME LIMITATIONS WHILE DESIGNING THE COMPILER USING EXISTING WIDELY TECHNIQUES:-

1. From the Perspective of Language Extensibility

Using a parser generator to create a parser has an important advantage over a handwritten parser: the grammar provides a concise specification of the corresponding language. As a result, we generally expect it to be easier to modify the machine-generated parser than the handwritten one. However, LALR (1) grammars for the popular Yacc tool [26] and similar parser generators are fairly brittle in the face of change. A grammar writer can avoid the need for disambiguation by factoring such prefixes by hand, but this requires extra effort and obfuscates the language specification.

2. Many sensible syntactic constructs are inherently ambiguous

When expressed in a CFG, commonly leading language designers to abandon syntactic formality and rely on informal metarules to solve these problems. The ubiquitous " dangling ELSE" problem is a classic example, traditionally requiring either an informal meta-rule or severe expansion and obfuscation of the CFG.

3. An additional problem common to both LR and LL

Parser generators are the separation of lexing and parsing:

This can make it unnecessarily hard to add new tokens to a grammar.

4. Limited lookahead Capability

As mentioned above LR (k) and LL (k) algorithms uses k symbols of lookahead in parsing an expression. Typically k is 2 for most of these algorithms because going further requires more resources and complicates the grammar.

II. METHODS AND MATERIAL

A. Related work

Packrat parsing is a novel technique for implementing parsers in a lazy functional programming language. A packrat parser provides the power and flexibility of top-down parsing with backtracking and unlimited lookahead, but nevertheless guarantees linear parse time. Any language defined by an LL(k) or LR(k) grammar can be recognized by a packrat parser, in addition to many languages that conventional linear-time algorithms do not support. This additional power simplifies the handling of common syntactic idioms.

Parsing Expression Grammar (PEG) is a new way to specify syntax, by means of a top-down process with limited backtracking. It can be directly transcribed into a recursive-descent parser. The parser does not require a separate lexer, and backtracking removes the usual LL(1) constraint. This is convenient for many applications, but there are two problems: PEG is not well understood as a language specification tool, and backtracking may result

in exponential processing time. Excessive backtracking does not matter in small interactive applications where the input is short and performance not critical. But, the author had a feeling that the usual programming languages do not require much backtracking

a. Motivation

The idea to choose this topic is to address the following problems arising while implementing packrat parsing.

1. Space Consumption

Probably the most striking characteristic of a packrat parser is the fact that it literally squirrels away everything it has ever computed about the input text, including the entire input text itself. For this reason packrat parsing always has storage requirements equal to some possibly substantial constant multiple of the input size. In contrast, LL (k), LR (k), and simple backtracking parsers can be designed so that space consumption grows only with the maximum nesting depth of the syntactic constructs appearing in the input, which in practice is an often order of magnitude smaller than the total size of the text. Although LL (k) and LR (k) parsers for any nonregular language still have linear space requirements in the worst case, this " average-case" difference can be important in practice. Even with such optimizations a packrat parser can consume many times more working storage than the size of the original input text

Tabling everything consumes main memory at a high rate and so risks starting thrashing, thus dropping the program from DRAM speed to disk speed. While theoretician may say the performance is still linear, that will not prevent complaints from users. The fact that many languages nowadays(including Java and Mercury) include a garbage collector(which must scan the tables at least once in a while, but will not be able to recover memory from them) just makes this even worse For this reason there are some application areas in which packrat parsing is probably not the best choice. For example, for parsing XML streams, which have a fairly simple structure but often encode large amounts of relatively flat, machine-generated data, the power and flexibility of packrat parsing is not needed and its storage cost would not be justified.

b. Objective & Scope of Study

The main objective behind this research work is to reduce the space consumption required for memoization with guarantee of linear parse time. Another aim is to avoiding the mutual recursive function calls. The scope of study is limited to implementation of efficient parser for parsing expression grammar. The efficiency of this parser will be measured from two perspectives mainly reduction in storage requirement for memorization and avoiding the mutual recursive function calls of parser to improve the efficiency directly, it helps to expand the applicability of packrat parsing in broader areas.

Although PEGs are a recent tool for describing grammars introduced by Ford in [1] with implementation of the packrat parser in Haskell programming language called peppy, their theory has solid foundations. Ford [2] showed how they can be reduced to TDPLs from the 1970s. The semantic predicates have also been successfully applied in the ANTLR LL (k) parser.

In [3] Roman shows that primitive recursive-descent parser with limited backtracking and integrated lexing is a reasonable possibility for parsing Java 1.5 where performance is not too critical. Also in [4] he shows that PEG is not good as a language specification tool. The most basic property of a specification is that one can clearly see what it specifies. And this is, unfortunately, not true for PEG. Further with slight modification in C grammar it gives reasonable performance.

And also in [5] he shows that he classical properties like FIRST and FOLLOW can be redefined for PEG and are simple to obtain even for a large grammar. One difference is that instead of letters are terminal expressions, which may mean sets of letters, or strings FIRST and FOLLOW are used to define conditions for choice and iteration that are analogous to the classical LL(1) conditions, although they have a different form and meaning. Checking these conditions produces useful information like the absence of reprocessing or language hiding. This helps to locate places that need further examination. Unfortunately, most results obtained here have the form of implications that cannot, in general, be reversed. The properties FIRST and FOLLOW are kind of upper bounds, and conditions using them are sufficient, but not necessary. This results in false

warnings. In particular, the lookahead operator "!" may trigger a whole avalanche of them. This paper addresses a need for proper handling of this operator as a future work.

In [6] a new approach is proposed for implementing PEGs, based on a virtual parsing machine, which is more suitable for pattern matching. Each PEG has a corresponding program that is executed by the parsing machine, and new programs are dynamically created and composed. The virtual machine is embedded in a scripting language and used by a pattern matching tool.

In [7] Robert grimm parsing technique which has been developed originally in the context of functional programming languages, practical for object-oriented languages. Furthermore, this parser generator supports simpler grammar specifications and more convenient error reporting, while also producing better performing parsers through aggressive optimizations.

In [8] the addition of cut operators was proposed to parsing expression grammars (PEGs), on which packrat parsing is based, to overcome its disadvantage. The concept of cut operators, which was 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, a packrat parser generator called Yapp was implemented 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 [9] methods are proposed that achieve the same effect in some practical grammars without manually inserting cut operators. In these 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.

Paper [10] argues (a) packrat parsers can be trivially implemented using a combination of definite clause grammar rules and memoing, and that (b) packrat parsing may actually be significantly less efficient than plain recursive descent with backtracking, but (c) memoing the recognizers of just one or two nonterminals, selected in accordance with Amdahl's law, can sometimes yield speedups.

Warth [11] presents a modification to the memoization mechanism used by packrat parser implementations that makes it possible for them to support (even indirectly or mutually) left-recursive rules. While it is possible for a packrat parser with this modification to yield super-linear parse times for some left-recursive grammars, experiments were carried out to show that this is not the case for typical uses of left recursion.

Finally, in [15] Coq formalization of the theory of PEGs is described and, based on it, a formal development of TRX: a formally verified parser interpreter for PEGs. This allows writing a PEG, together with its semantic actions, in Coq and then to extract from it a parser with total correctness guarantees. That means that the parser will terminate on all inputs and produce parsing results correct with respect to the semantics of PEGs. Considering the importance of parsing, this result appears as a first step towards a general way to bring added quality and security to all kinds of software.

B. Proposed Work

Packrat Parsing is a variant of recursive decent parsing technique with memoization by saving intermediate parsing result as they are computed so that result will not be reevaluated. It is extremely useful as it allows the use of unlimited look ahead without compromising on the power and flexibility of backtracking. However, Packrat parsers need storage which is in the order of constant multiple of input size for memoization. This makes packrat parsers not suitable for parsing input streams which appears to be in simple format but have large amount of data.

In this project instead of translating productions into procedure calls with memoization, an attempt is made to eliminate the calls by using stack without using memoization for implementation of ordered choice operator in Parsing expression Grammar (PEG). The experimental results show the possibility of using this stack based algorithm to eliminate the need of storage for memoization to improve the performance of packrat parser in terms of storage space.

III. RESULTS AND DISCUSSION

It is expected that the proposed approach improve the performance of packrat parser from two perspectives mainly reduction in storage requirement for memorization and avoiding the mutual recursive function calls of parser.

IV. CONCLUSION

Packrat parsers need storage which is in the order of constant multiple of input size for memoization. This makes packrat parsers not suitable for parsing input streams which appears to be in simple format but have large amount of data. The future work will be translating productions into procedure calls with memoization, an attempt is made to eliminate the calls by using stack without using memoization for implementation of ordered choice operator in Parsing expression Grammar (PEG). The experimental results show the possibility of using this stack based algorithm to eliminate the need of storage for memoization to improve the performance of packrat parser in terms of storage space.

V. REFERENCES

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

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

[3] Redziejowski, R.R.: Parsing Expression Grammar as a primitive recursive-descent parser with backtracking.Fundamenta Informaticae 79,3-4, (2007) pages 513-524.

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

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

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

[7] 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.

[8] 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.

[9] Mizushima, K., Maeda, A., Yamaguchi, Y.: Packrat parsers can handle practical grammars in mostly constant space. In PASTE'10: Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering, Toronto, Ontario, ACM (2010) 29-36.

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

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

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

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

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

[15] 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.

[16] Lucas, P. The structure of formula-translators. ALGOL Bulletin Supplement 16 (September 1961), 1-27.

[17] Hopgood, F. R. A. Compiling Techniques. MacDonalds, 1969.

[18] Brooker, P., and Morris, D. Some proposals for the realization of a certain assembly program. The Computer Journal 3, 4 (1961), 220-231.

[19] Rosen, S. A compiler-building system developed by Brooker and Morris. Commun. ACM 7, 7 (July 1964), 403-414.

[20] McClure, R. M. Tmg - a syntax directed compiler. In Proceedings of the 20th ACM National Conference (24{26 August 1965), L. Winner, Ed., ACM, pp. 262-274.

[21] Birman, A. The TMG Recognition Schema. PhD thesis, Princeton University, February 1970.

[22] Birman, A., and Ullman, J. D. Parsing algorithms with backtrack. Information and Control 23(1973), 1-34.

[23] Aho, A. V., and Ullman, J. D. The Theory of Parsing, Translation and Compiling, Vol. I, Parsing. Prentice Hall, 1972.

[24] Earley, J.: An e_cient context-free parsing algorithm. Communications of the ACM 13(2) (February 1970).

[25] Tratt, L.: Domain specific language implementation via compile-time metaprogramming. TOPLAS 30(6) (2008) 1-40.

[26] J. R. Levine. lex & yacc. O' Reilly, Oct. 1992.

[27] C. Braband, M. I. Schwartzbach, and M. Vanggaard. The METAFRONT system: Extensible parsing and transformation. Technical Report BRICS RS-03-7, BRICS, Aarhus, Denmark, Feb. 2003.

[28] O. Enseling. Build your own languages with JavaCC.JavaWorld, Dec. 2000. Available at http://www.javaworld.com/javaworld/jw-12-2000/ jw-1229-cooltools.html.

[29] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. Software—Practice and Experience, 25(7):789– 810, July 1995.

[30] Alexander Birman. PhD thesis, 1970.

[31] Alexander Birman and Jeffrey D. Ullman. Parsing algorithms with backtrack.Information and Control, 23, 1973.

[32] Daniel J. Salomon and Gordon V. Cormack. Scannerless NSLR(1) parsing of programming languages. In Proceedings of the ACM SIGPLAN' 89 Conference on Programming Language Design and Implementation (PLDI), pages 170– 178, Jul 1989.

[33] Terence John Parr. Obtaining practical variants of LL(k) and LR(k) for k > 1 by splitting the atomic k-tuple. PhD thesis, Purdue University, Apr 1993.

[34] Terence J. Parr and RussellW. Quong. ANTLR: A predicated-LL(k) parser generator. Software Practice and Experience, 25(7):789– 810, 1995.

[35] Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 3rd edition, June 1997.

[36] Simon Peyton Jones and John Hughes (editors). Haskell 98 Report, 1998. http://www.haskell.org.

[37] Laurence Tratt. Direct Left-Recursive Parsing Expression Grammars: Technical report EIS-10-01 Middlesex University, The Burroughs, London, NW4 4BT, United Kingdom (2010).

[38] R. Grimm. Practical packrat parsing. New York University Technical Report, Dept. of Computer Science, TR2004-854, 2004.

VI. BIOGRAPHY

Manish M. Goswami is a Research Assistant in the IT Department, Rajiv Gandhi College of Engineering and Research, Nagpur, India. He is pursuing PhD. His research interests are compiler, Programming language, Theory of Computation.

Priyanka Gonnade Assistant Professor in the CSE Department, Rajiv Gandhi College of Engineering and Research, Nagpur, India. She has received Master of Technology (M.Tech.) degree. She's research interests are Image Processing and Soft Computing.