Performance Improvement of Stack Based Recursive-Descent Parser for Parsing Expression Grammar

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

Dr. M.M. Raghuwanshi
Professor, Yeshwantrao Chauhan College of Engineering, Nagpur, India,

Dr. Latesh Malik
Professor, Dept of CSE, GHRCE, Nagpur, India

Abstract- Recursive Descent parser can be executed in linear time with Packrat parsing method. This is achieved with memoization technique so that redundant parsing in case of backtracking can be avoided by memorizing all parsing results. But this linear time comes at the cost of large heap consumption in memorization which nullifies its benefits. Due to this developers need to make a difficult choice not to use packrat parsing despite it gives guarantee of linear parsetime. Parsing Expression Grammar relies on a limited backtracking; with language defined in the its way as a sequence of statements. Hence grammar will never return farther back than the starting of current statement. In this paper an idea of extending PEG as a primitive recursive descent parser with backtracking is used by eliminating the procedure calls representing production of grammar by using stack and performed optimization using *and cut operator mainly in addition to other optimizations. Experimental results indeed show an improved performance over straightforward implemented recursive descent parser, Mouse tool and also gave encouraged results when compared with packrat parser with memorization, Rats Parser though it cannot beat the later.

Keywords – PEG, Parser, Packrat.

I. INTRODUCTION

Recursive descent parser with backtracking [1] can be efficiently implemented by Packrat parsing technique. Packrat parsing works on the idea of memorization where intermediate results parsed at each distinct position in the input are stored in the memory. Since backtracking is avoided, danger of running the parser in exponential time is avoided and linear time working is guaranteed. Packrat parser is based upon Parsing Expression Grammars or PEG-extended grammars. A PEG [4] is a syntactic foundation that can recognize a wide range of formats, not only programming source code but also data formats such as XML and computer logs. However, many researchers and developers complained that with some inputs benefits of memorization are not visible and found to be less efficient than plain recursive descent parser. The reason behind is huge linear memory consumption, requiring up to 100– 400 bytes of memory for every byte of input [3], [5]. On practical front, memory consumption of this size reduce overall performance of the system , such as disk slashing and repeated garbage collector invocations. This is the reason most of the developers, despite linear time guarantee, have ignored the benefits of memoization [11].
In development experience of Nez grammars, or PEG- extended grammars by authors in [6], it was observed no or very little backtrack activity in parsing data formats, such as XML and JSON. Compared to packrat parsing, a plain recursive descent parser is found to be efficient, while packrat parsing played degraded performance, especially in cases of a large-scale input sizes. In cases of parsing C source code, despite a fact that we can observe significantly increased backtracking activity, a plain parser still shows very competitive performance compared to a packrat parser. The advantage of packrat parsing is still questionable if we take its huge memory consumption into consideration.

Also many researchers have claimed [1], [9], [11], in practice everytime exponential behavior does not seem to happen in practice. However, this may not be true to some inputs. For example, parsing a common source of JavaScript, such as JQuery library, involves a considerable high- level backtracking. Without any support of memoization, the parsing of JQuery libraries takes a literally exponential time. The problem with packrat parsing is that wide range of input variations, especially very large-scale files and low backtracking activity cases are not handled well.
In this paper an idea to eliminate the procedure calls representing production of grammar by using stack [10] is extended to apply some aggressive optimization which would not have been possible if we directly implement straightforward recursive descent parser. Further taking advantage of stack based parser, operator * has been optimized to reduce the no. of Push and pop operations typically encountered in backtracking top down parser. Further Cut operator is used to reduce backtracking activity. Experimental results indeed show an improved performance over straightforward implemented recursive descent parser, Mouse tool. Further heap utilization is also compared with our parser with the parser generator RATS. Because of optimization used, remarkable improvement in heap utilization is seen though it cannot be matched with memorization but nearest to it.

II. STACK BASED PARSER

An idea of Stack based parser [11] is used to eliminate function calls in recursive descent parser by explicit use of stack. Implementing parser to directly deal with the stack helps to apply optimization which will be discussed in next section. This optimization would not have been possible in straightforward recursive descent parser. A traditional parser for PEG described below is composed of parse functions:-

S -> A B d / C S
A -> a / b
B -> a /c
C -> c

The straightforward backtracking parser is as follows:-

```
Main ()
{
 i := 0;
 if S() and I[i] =$ then success;
 else error();
}

S() { A();
      B();
    if (I[i]=d) { i:= i+1 }.
    If any one of these conditions is false then evaluate following else
    return true.
     C()
     S().
    If any one of these the conditions is false then return false else return
    true.
}

A() { if (I[i]==a) { i := i + 1; return     true }
     else if (I[i]==c) {i := i + 1; return true} else return false }

B() { if (I[i]==a) { i := i + 1; return  true}
     else if (I[i]==b) {i := i + 1; return true} else return false }

C() {if (I[i]==c) { i := i + 1; return true} else return false }
```

The procedure calls in above parser can be easily eliminated by converting the function calls into explicit call stack operations using stack push and goto statements as usual to facilitate elimination of procedure calls as shown in [13]

and slightly tweaked here to implement stack based parser . In addition, the body of those functions is partitioned whose corresponding nonterminal is having more than one choice and separately label each partition. stack based parser uses two stacks to simulate recursive descent parsing . For storing the label (call frame) for next nonterminal in the choice, parse stack is used and starting label of alternative choice(backtrack frame) is stored on the backtrack stack if the nonterminal has more than one choices. For example, consider the evaluation of nonterminal S in S->ABd|CS. Before evaluating choice ABd starting label for partition for evaluating choice CS must be stored on stack named as backtrack stack for allowing the parser for backtracking if choice ABd fails to parse. Also, when A is evaluated for ABd but before it label for evaluation of next node corresponding to B in ABd must be pushed onto the stack called as parse stack to return it after A is evaluated. Let L be a line label, s be a parse stack and j be a pointer in the input array I. These 3 variable constitute a triplicate denoted as (L,s,j) as an elementary current descriptor. Current descriptor is denoted by R. New descriptor using the label at the top of the current parse stack is created whenever parse function ends point of nondeterminism is detected. After execution of the algorithm, at input I[i] say, the top element L is popped from the stack s = [s', L] and (L, s', i) is added to R (if it has not already been added). We denote POP(s, i, R) for this action. Then the next descriptor (L', t, j) is removed from R and execution starts at line L' with call stack t and input symbol I[j]. Overall execution is declared as terminated when the R and backtrack stack becomes empty. L_k stands for recording the line label L and the current input buffer index k on the stack for allowing backtracking. [ ] denotes the empty stack, and a function PUSH(s, L_k) which is stack operation the of stack s by pushing on the element L_k. The detail of the algorithm reproduced here from [11] is as follows:

```
i := 0; R := ∅; s := [L_0^0]
Ls: add (Ls1, s, i) to R
L0: if (R ≠∅) {remove (L, s1, j) from R
     if (L = L0 and s1 = [ ] and j = |I|   and b=[]) report success else { s
     := s1; i := j; goto L }
     else report failure.
Ls1: PUSH(s,L_1^i); PUSH(b,L_s2^i) ; goto LA
L1: PUSH(s,L_2^i); goto LB
L2: if (I[i] = d) { i := i + 1; POP(s, i, R) };else POP(bs,i,R); goto L0
Ls2: PUSH(s,L_3^i); PUSH(b,L_s3^i) goto Lc
L3: PUSH(s,L_4^i); goto Ls
L4: POP(s, i,R); goto L0
Ls3: POP(s, i,R); goto L0
LA: if (I[i] = a) { i := i + 1; POP(s, i,R); goto L0 }
   else{ if (I[i] = c) { i := i + 1; POP(s, i,R) goto L0 }
   else POP(bs,i,R); goto L0 }
LB: if (I[i] = a) { i := i + 1; POP(s, i,R); goto L0 }
    else{ if (I[i] = b) { i := i + 1;   POP(s, i,R) goto L0}
    else POP(bs,i,R) goto L0 }
Lc: if (I[i] = c) { i := i + 1; POP(s, i,R); goto L0 else POP(bs,i,R) goto L0 }
```

***s**:Parse Stack **b**:Backtrack Stack*

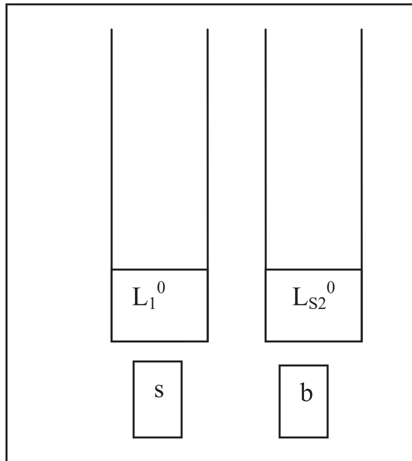

Fig. 1. When input pointer is at pos=0 and Label Ls1 is evaluated

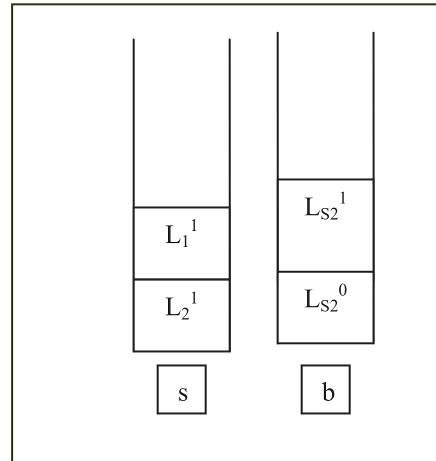

Fig. 3. When input pointer is at pos=1 and Label Ls1 is evaluated

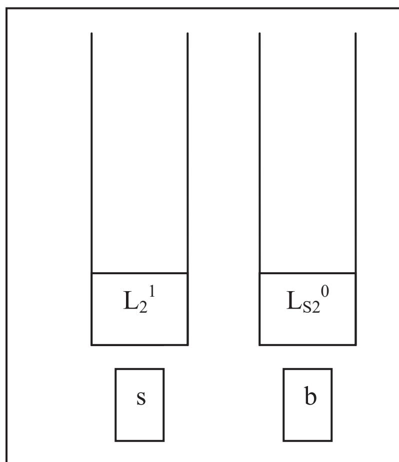

Fig. 2. When input a is matched and Label L1 is evaluated

*A. Optimization of * operator in PEG*

The above stack based parser can be used to optimize * operator in PEG as follows:-

C* can be written in Parsing Expression Grammar as:
C_1->CC_1 / ε.

Let us first see above PEG rule for * is implemented in our parser . Consider the following expression:-

L_{C1}: PUSH (b, L_iC1'); PUSH(s, LC1''); goto L_C;
L_{C1}'' :pop();PUSH (b, LC1'); PUSH(s, LC1''); goto L_C;
L_{C1}': POP(s,j,R);
L_C :

Careful observation clearly indicates that definition of C* gets expanded with reference to our parser, a new backtrack entry is always pushed and discarded it when the pattern C matches. It is noticed that from previous observation when definition of C* in execution, that when a new backtrack entry is pushed, the only thing that changes is the current subject position, thus, instead of discarding the backtrack entry; we could have just updated it. With this changes above snapshot of code can be rewritten as:

```
.
.
.
.
.
LC1:PUSH(b,LC1');goto LC1'';
LC1'':j=i;PUSH(s,LC1'') goto LC;
LC1': POP(s,j,R);
LC:
.
.
.
```

B. Optimization using Cut Operator

Concept of Cut operator was proposed and implemented with packrat parser in [9]. We are applying cut operator to our stack based parser to reduce the backtracking activity to speed up the performance of the parser. To explore the possibility, following PEG expressing a typical programming language's control statements (rules about spacing, the definitions of E and I are omitted) has been picjed up from [9]:

```
S-> "if" "(" E ")" "then " S ("else" S)?;
    / "while" "(" E ")" S;
    / "print" "(" E ")" S;
    / "set" I "=" E;
```

Here associated with nonterminal S, there are four choices. If suppose when evaluating the first choice "if" gets matched, at this point it is very clear that other choices are not going to succeed. As discussed above our stack based parser saves the backtracking information when choice corresponding to "if" set is evaluated in preparation of failure. However, in this case, this preparation of saving information is of no use and going to be wasted as it would not be possible to succeed with other choices. The solution is n such a case, parser is to be informed to avoid saving backtrack information. Grammar writers can tell the parser to dispose of the backtrack information as soon as it becomes unnecessary by inserting cut operators at appropriate places as follows:-.

```
S ->  "if" ^ "(" E ")" S ("else" S)?
      / "while" ^ "(" E ")" S
      / "print" ^ "(" E ")" S
      / "set" I "=" E ";";
```

In our stack based parser, backtrack stack will not be burdened with unnecessary push and pop operations which is anyhow going to be wasted thereby decreasing latency time.

III. Experiment and Result

The resulting program is run on Core 2 Duo 1.8GH, 2 GB RAM machine. First PEG for subset of C programming language is defined. then that PEG is translated to stack based parser generated. The resulting parser is used to parse various C programs. The result is compared with Mouse, recursive descent parser based on PEG. The comparison is and analysis on various parameters are discussed below:-

Effect on Backtracking activity

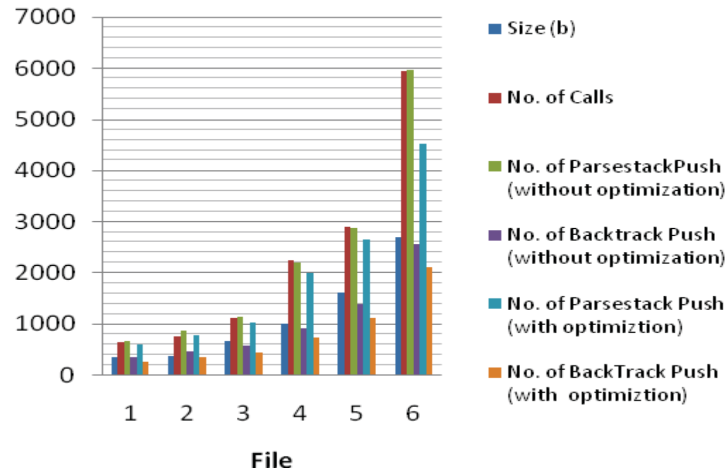

Fig. 4. Effects on Backtracking activity

In the above table fig. 4, programs written in C programming language are given as input with varying file sizes (bytes) numbered 1 to 6 . The input is given to recursive descent based parser, Mouse where performance is measured in No. of calls and Cpu Time. This is compared when the same input is given to our stack based parser with * optimization and performance is measured in terms of no. of Packrat Pushes, no. of backtrack Pushes and CPU Time(sec.). The parameter 'no. of calls' in Mouse parser is obviously compared with parameters 'no. of parsestack pushes' and 'no.of backtrack pushes' in our parser with * optimization since procedure call is nothing but pushing and poping stack frame. The comparison shows the stack based parser with optimization of * operator has an edge over the mouse parser. This is because when the recursive descent parser is executed for procedures for * operator calls are made recursively to expression itself pushing and popping call frames unnecessarily. This is avoided in our parser using * optimization as discussed previously.

Latency

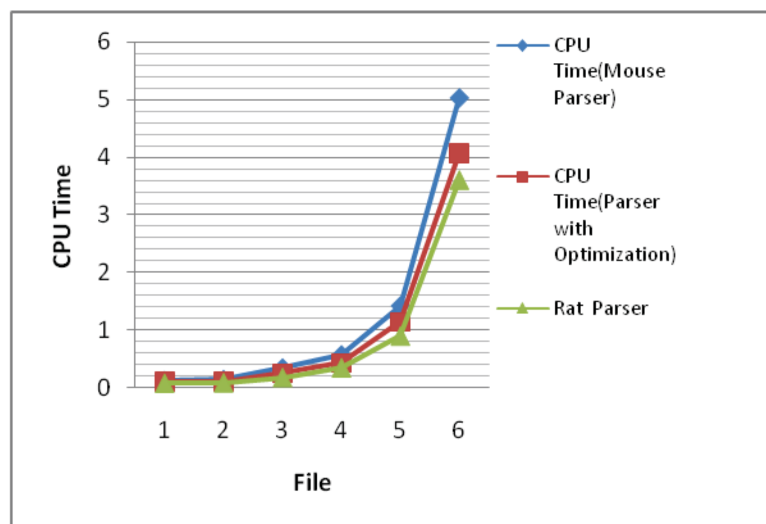

Fig.5. Latency in rats < Latency in Optimized Bactracking Parser < Mouse Parser

Latency in Rats Parser is lowest while Latency in Mouse Parser is highest while latency in our parser is in between them. This is obvious as Rats uses memorization with optimization which speed up the parsing by avoiding redundant calls. Mouse Parser solely depend on pure backtracking approach while our parser though relies on backtracking but its optimized approach help cut down the unnecessary backtracking activities gaining speed up in parsing which caused to reduce latency time. But it is clearly evident that even after applying the optimization to our

parser latency is not minimum Rats Parser still beats our parser by considerable margin. It was expected that as size of the program increases Rats parser outperform our parser but as memorization increases disk slashing and garbage collector invocations are repeated which offset gain in speed obtained from reduced backtracking activities .

Heap Utilization

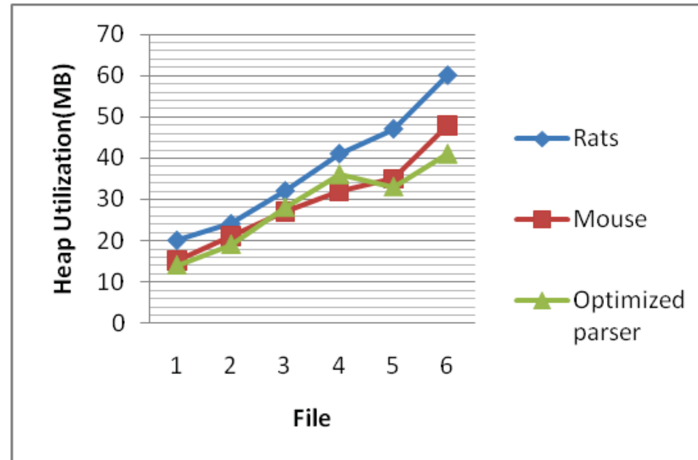

Fig.6. Heap Utilization in Rats > Heap Utilization in Mouse > Heap Utilization in Optimized parser

Heap utilization in Rats is largest compared to others which is obvious as it uses memorization approach to avoid redundandant call which comes at the cost of sacrificing memory. Our parser and Mouse parser since does not use memorization shows less utilization of memory. Heap utilization in these two approaches are quite similar since parsing depends purely on backtracking approach. Optimized parser slightly takes more memory for file no.4 which is due to the fact that there no chance of utilization of cut and * operator in that program.

IV.RELATED WORK

Packrat parsing is a recently introduced innovative technique by Ford [3] which uses memorization for implementing recursive descent parsers with backtracking. Ford's Pappy [3] parser has shown the superiority of the technique requiring linear parse time. Grimm's Rats! [5], aggressively optimized the performance of the parser by applying optimization to reduce memory usage for memoization. However, comparative studies with a plain recursive descent parser have not been performed, perhaps due to the fact that the superiority of packrat parsing seemed trivial in their experiments. However, as practical experiences have increased in various parsing contexts, it is no longer so trivial.In Refs. [1], [9], [11], the effectiveness of memoization in practice put a questin mark on the technique. Warth et al., the developers of OMeta [11], pointed out that the overhead of memoization may outweigh its benefits for the common case.Also, Becket et al., observed that packrat parsing might actually be significantly score less than plain recursive descent parsers with backtracking [1] based on their definite clause grammars (DCGs) analysis. Medeiros et al. proposed an alternative method to packrat parsing [7]. Exponential behavior does not happen in practice was the their main assumption. However, as we have shown before, the parsing of a JQuery library rebuts this assumption.

The major problem with packrat parsing lies in the consumption of huge memory for every byte of input. Backtracking region can be limited by applying extended some PEG notion with a cut operator as shown by Mizushima et al. and presented mostly constant heap consumption with their parser generator, Yapp [8]. Interestingly, the automatic insertion of cut operations suggested by same authors , optimization can be achieved to packrat parsing. However, In high backtracking activity effect of applicability of cut operations remains to be seen.Redziejowski, the author of Mouse, focused on the number of nonterminal calls instead of the input size, and concluded that memoizing the two most recent results succeeded in significantly reducing heap consumption [9]. This approach is similar in terms of expiring older memoized results, while Mouse's approach abandons the linear time parsing property. In our ap- proach, we demonstrate that elastic packrat parsing can handle potential exponential cases. Choosing nonterminals to be memoized was first attempted as a part of Rats!

optimization [5]. The grammar developers can specify the transient attribute to apply their heuristics to control whether nonterminals will be memoized or not. Becket et al. pointed out the heuristic analysis of grammatical properties resulted in very limited improvement [1]. Our dynamic analysis approach, on the contrary, avoids such difficulties and yields clear effects for improved performance.

IV.CONCLUSION AND FUTURE WORK

In this paper, stack based implementation of packrat parser is used so that optimization of PEG based parser is carried out where translated into its corresponding procedure. Cut operator and Choice operator are used to perform the optimization. The performance in terms of time required to recognize the input is compared with Mouse parser and Rats Parser which shows in improvement in terms of latency and memory utilization. Moreover, the parser is easy to construct and reflects Parsing expression Grammar as in traditional recursive descent parser. Future work is to optimize choice operator '/' so that when combined with optimization proposed in this paper can significantly improve the performance and compete with the parser with memorization as in packrat parser.

REFERENCES

[1] Becket, R. and Somogyi(2008): DCGs+Memoing = Packrat Pars- ing but is It Worth It?, Proc. 10th International Conference on Practical Aspects of Declarative Languages, PADL '08, pp.182– 196, Springer-Verlag, Berlin, Heidelberg (online).

[2] http://dl.acm.org/citation.cfm?id=1785754.1785767

[3] Birman,A. and Ullman,J.D(1973).: Parsing algorithms with backtrack, Information and Control, Vol.23, No.1, pp.1–34 (online), DOI: http://dx.doi.org/10.1016/S0019-9958(73)90851-6

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

[5] Ford, B. (2004): Parsing Expression Grammars: A Recognition-based Syn- tactic Foundation, Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POP '04, pp.111–122, ACM (online), DOI: 10.1145/964001.964011)

[6] Grimm, R. (2006): Better Extensibility Through Modular Syntax, Proc. 2006 ACM SIGPLAN Conference on Programming Language De-sign and Implementation, PLDI '06, pp.38–51, ACM (online), DOI: 10.1145/1133981.1133987

[7] Kuramitsu, K(2015): Toward Trans-Parsing with Nez, Informal Proceedings of JSSST Workshop on Programming and Programming Languages

[8] Medeiros, S. and Ierusalimschy, R(2008).: A Parsing Machine for PEGs, Proc. 2008 Symposium on Dynamic Languages, DLS '08, pp.2:1–2:12, ACM (online), DOI: 10.1145/1408681.1408683

[9] Mizushima, K., Maeda, A. and Yamaguchi,Y(2010): Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space, Proc. 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Soft- ware Tools and Engineering, PASTE '10, pp.29–36, ACM (online), DOI: 10.1145/1806672.1806679

[10] Redziejowski, R.R(2007).: Parsing Expression Grammar As a Primitive Recursive-Descent Parser with Backtracking, Fundam. Inf., Vol.79, No.3-4, pp.513–524 (online), available from http://dl.acm.org/ citation.cfm?id=2367396.2367415

[11] M. M. Goswami,M.M.Raghuwanshu,L.Malik(2013,Nov): Stack Based Implementation of Ordered Choice in Packrat Parsing, IJCSMC,Vol. 2, Issue. 11, pg.303 – 309.

[12] Warth, A. and Piumarta, I(2007).: OMeta: An Object-oriented Language for Pattern Matching, Proc. 2007 Symposium on Dynamic Languages, DLS '07, pp.11–19, ACM (online), DOI: 10.1145/1297081.129708

[13] Elizabeth Scott and Adrian Johnstone.GLL Parsing, Volume 253, Issue 7, 17 September 2010, Pages 177–189.