Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 2, Issue. 11, November 2013, pg.303 – 309

RESEARCH ARTICLE

Stack Based Implementation of Ordered Choice in Packrat Parsing

Manish M. Goswami
Research Scholar,
G.H. Raisoni College of Engg.
Nagpur, India

M.M. Raghuwanshi
Professor,
Rajiv Gandhi College of Engg.
Nagpur, India

Latesh Malik
Professor, Dept. of CSE,
G.H.raisoni College of Engineering
Nagpur, India

ABSTRACT: ***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 paper 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 with a guarantee of linear parse time.

Keywords: Parsing; PEG; Packrat; CFG

I. INTRODUCTION

Recursive-descent parsing is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. It was suggested as early as in 1961 by Lucas [1]. The great advantage of a recursive-descent parser is its simplicity and clear relationship to the grammar. For smaller grammars, the parser can be easily produced and maintained by hand. This is contrary to bottom-up parsers, normally driven by large tables that have no obvious relationship to the grammar; these tables *must* be mechanically generated. The problem with constructing recursive-descent parsers from a classical context-

free grammar is that the grammar must have the so-called *LL*(1) property. Forcing the language into the *LL*(1) mold can make the grammar – and the parser – unreadable. The *LL*(1) restriction can be circumvented by the use of backtracking. However, full backtracking may require exponential time. A reasonable compromise is limited backtracking: never try another alternative after one alternative already succeeded on a portion of input. Recently, Ford [2–4] introduced a language for writing recursive-descent parsers with limited backtracking. It is called Parsing Expression Grammar (PEG) and has the form of a grammar that can be easily transcribed into a set of recursive procedures. In addition to backtracking, PEG can directly define structures that normally require a separate ”lexer” or ”scanner”. Together with lifting of the *LL*(1) restriction, this gives a very convenient tool when we need an ad-hoc parser for some application. Theoretically, even the limited backtracking may require a lot of time. In [2, 3], PEG was introduced together with a technique called *packrat parsing*. Packrat parsing handles backtracking by extensive *memoization*: storing all results of parsing procedures. It guarantees linear parsing time at a huge memory cost. There exists a complete parser generator named *Rats!* [4, 5] that produces packrat parsers from PEG. Excessive backtracking does not matter in small interactive applications where the input is short and performance is not critical. Moreover, experiments reported in [7, 8] demonstrated a moderate backtracking activity in PEG parsers for programming languages Java 1.5 and C.

In this paper stack based approach is adopted to transcribe a grammar into explicit call stack Push, Pop statements. in place of procedure calls in packrat parsing. The algorithm is inspired from Generalized LL parsing algorithm[12].Each nonterminal with which production begins is associated with the label where code for evaluation of that nonterminal in parse tree is stored. The data structure stack is used basically for two purposes: one to store the label for the next nonterminal in a choice to be evaluated and second to store the label of alternative choice if the production has more than two choices. Performance of resulting parser is compared with Generalized LL parsing algorithm (non optimized version). Experimental results shows that Packrat parsing implemented in this way(only implementation of ordered choice is considered) shows further scope for optimization which is not possible when implemented using procedure calls. Also, algorithm when compared with GLL parsing algorithm shows improvement on some inputs which is obvious as GLL algorithm tries to explore all the choices while packrat parsing algorithm explores an alternative choice only if first choice fails.

II. BACKGROUND

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* <- *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:

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

It appears that PEGs are similar to Extended Backus-Naur Forms (EBNFs)[9]. However, it should be noted that *e*1 / *e*2 is *an ordered choice*, not *an unordered choice*, *e*1 | *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, $e1 / e2 \neq e2 / e1$. 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", ...} but *Ø* 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[10].

Packrat Parsing

Roughly speaking, packrat parsing is a combination of PEG-based recursive descent parsing with memoization. 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 liner 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). *Rats!*[5], 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*[11]. However, *Rats!* does not resolve the fundamental problem that packrat parsers require *O*(*n*) space.

Call Stacks and elementary descriptors

A traditional parser for Γ0 described below is composed of parse functions:-

S -> ASd | BS
A-> a|b
B->a |c

Main()

{

i := 0

if *pS*() and *I*[*i*] = $ report success

else *error*()

}

pS() *{* Evaluate *pA*() followed by *pS*() and then evaluate ' **if** (*I*[*i*] = *d*) *{ i* := *i* + 1 *}*' .If any one of condition is false then evaluate following else return true.

Evaluate *pB*() followed by *pS*() .If any one of the condition is false then return false else return true. *}*

pA() *{* **if** (*I*[*i*] = *a*) *{ i* := *i* + 1; return true *}*

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

pB() *{* **if** (*I*[*i*] = *a*) *{ i* := *i* + 1 ; return true*}*

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

Of course, Γ0 is not LL (1) parsing algorithm so this algorithm will not behave correctly without some additional mechanism for dealing with non-determinism. This is addressed by converting the function calls into explicit call stack operations using stack *push* and *goto* statements in the usual way. We also partition the body of those functions whose corresponding nonterminal is not LL (1) and separately label each partition.

In practice, then, some *goto* statements will have several target labels, corresponding to these multiple partitions: for example, this will be the case for the nonterminal *S* in Γ0. We use descriptors to record each possible choice, and replace termination in the RD algorithm with execution re-start from the point recorded in the next descriptor on the stack. Instead of calls to the error function, the algorithm simply processes the next descriptor of alternative choice if available on the stack and it terminates when there are no further descriptors to be processed. Finally, it signals error when there is no descriptor available and all input is not processed.

Two stacks are maintained, one is parse stack which is used for storing the label for next nonterminal in the choice and second is backtrack stack which is used to store the starting label of next choice if the nonterminal has more than one productions.

For example, If a node corresponding nonterminal S in S->ASd|BS is evaluated then starting label for evaluating choice BS will be stored on stack named as backtrack stack corresponding choice for backtracking if choice ASd fails and control will be transferred to starting label of choice ASd. In particular A is evaluated but before it label for evaluation of next node corresponding to S in ASd will be pushed onto the stack called as parse stack to return it after A is evaluated.

In detail, an *elementary descriptor* is a triple (*L, s, j*) where *L* is a line label, *s* is a parse stack and *j* is a position in the input array *I*. We maintain a placeholder R for current descriptor. At the end of a parse function and at points of non-determinism in the grammar we create a new descriptor using the label at the top of the current parse stack. When a particular execution of the algorithm stops, 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 use *POP*(*s, i, R*) to denote 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*]. The overall execution terminates when the

R is empty. In order to allow us for backtracking we record both the line label *L* and the current input buffer index *k* on the stack using the notation L_k. At this interim stage we treat the stack as a bracketed list, [ ] denotes the empty stack, and we assume that we have a function *PUSH*(s,L_k) which simply updates the stack *s* by pushing on the element L_k.The details of the algorithm is as follows:-

i := 0; *R* := ∅; *s* := [L_0^0]
L_S : add (L_{S1}, *s*, *i*) to *R*
*L*0: **if** (*R* ≠ ∅) { remove (L, s_1, j) from *R*
if (*L* = *L*0 and *s*1 = [ ] and *j* = |*I*| *and b=[]*) report success
else { *s* := s_1; *i* := *j*; **goto** *L* }
else report failure.

L_{S1} : *PUSH*(s,L_i^1); PUSH(b,L_{S2}^i) ; **goto** L_A
*L*1: *PUSH*(s,L^i_2); **goto** *LS*
L_2: **if** (*I*[*i*] = *d*) { *i* := *i* + 1; *POP*(*s, i,R*) };else POP(b_s,i,R); **goto** L_0

L_{S2} : *PUSH*(s,L^i_3); PUSH(b,L_{S3}^i) **goto** L_B
L_3: *PUSH*(s,L^i_4); **goto** L_S
L_4: *POP*(*s, i,R*); **goto** L_0

L_{S3} : *POP*(*s, i,R*); **goto** L_0

L_A: **if** (*I*[*i*] = *a*) { *i* := *i* + 1; *POP*(*s, i,R*); **goto** L_0 }
else{ **if** (*I*[*i*] = *c*) { *i* := *i* + 1; *POP*(*s, i,R*) **goto** L_0 }
else POP(b_s,i,R); **goto** L_0 }
L_B: **if** (*I*[*i*] = *a*) { *i* := *i* + 1; *POP*(*s, i,R*); **goto** L_0 }
else{ **if** (*I*[*i*] = *b*) { *i* := *i* + 1; *POP*(*s, i,R*) **goto** L_0}
else POP(b_s,i,R) **goto** L_0 }

P:Parse Stack B:Backtrack Stack

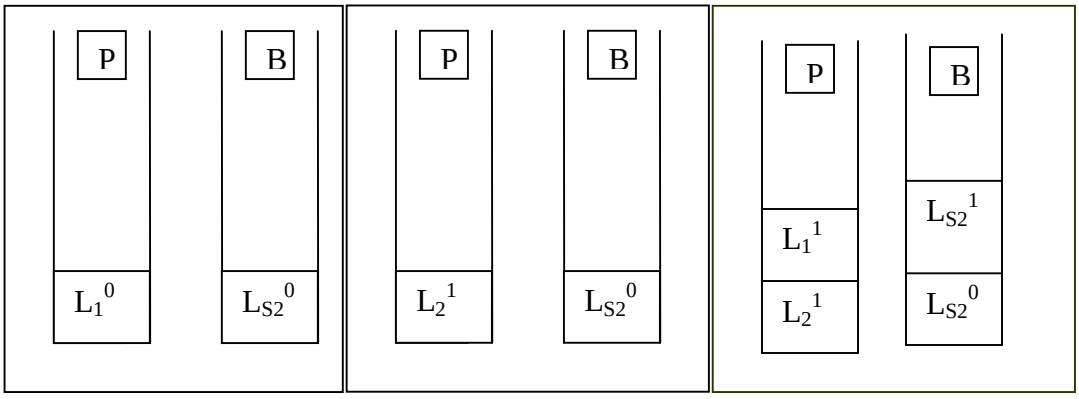

Fig.2: When input pointer is at pos=0 and Label Ls1 is evaluated

Fig.3: When input a is matched and Label L_1 is evaluated

Fig.4: When input pointer is at pos=1 and Label Ls_1 is evaluated

III. EXPERIMENTATION AND ANALYSIS

The Algorithm is implemented in Java. The grammar which is translated to explicit stack calls is reproduced here mentioned in above section:-

$S \rightarrow AS\ d \mid BS \mid \varepsilon$

$A \rightarrow a \mid c$

$B \rightarrow a \mid b$

Though the grammar seems to be simple it reveals many facts pertaining to parsing. The resulting program is run on Core 2 Duo 1.8GH, 2 GB RAM machine. The algorithm is run on a particular input string for 50 times for each parameter and the average is taken. Parameters which were considered into consideration to measure performance are:-

1) No. of Push

2) No. of Pop

3) Time required recognizing the input string.

The performance of above algorithm is measured and compared with GLL algorithm (The non-optimized version of GLL i.e. without graph structured stack is considered here)

As can be seen from the following table, no. of push count, pop count, and parsing time differs for each algorithm for particular input. The explanation for the same is cited as follows on case to case basis:-

Case I: For input $a^{65}bd^{65}$, the stack based implementation of packrat parsing algorithm requires less time as compared to GLL algorithm. This is because in GLL algorithm for all a's, two choices ASd and BS are evaluated. Here only S->Asd is correct choice for recognizing $a^{65}bd^{65}$ .In the proposed algorithm discussed above choice Asd is evaluated for all a's reducing push count and pop count thereby reducing time required to parse the input.

Case II: Here GLL algorithm takes less time. This is because for all b's correct choice i.e. BS is picked up while in the above proposed algorithm for every b, first Asd is evaluated by the property of packrat parsing (ordered choice) which will be obviously failed and then BS will be evaluated resulting into more push and pops thereby requiring more time.

Case III: Here again GLL requires more time compared to above algorithm. This case is similar to Case I.

Input= $a^{65}bd^{65}$			
Sr. No.	Parameter	Stack based implementation of packrat parsing Algorithm	GLL Parsing Algorithm
1	Push_Count	204	6140
	Pop_Count	205	15358
	Parsing Time(ns)	674655	14449780
Input= b^{130}			
2	Push_Count	654	260
	Pop_Count	655	261
	Parsing Time(ns)	1231788	1132462

Input=$a^{10}d$			
3	Push_Count	6650	4092
	Pop_Count	6651	6129
	Parsing Time(ns)	6119207	7073932

IV. CONCLUSION AND FUTURE WORK

In this paper, an algorithm for implementation of packrat parsing algorithm is proposed by eliminating function calls by explicit stack push and pop operations. The performance in terms of time required to recognize the input is compared with GLL parsing algorithm. The advantage that can be seen with this algorithm is the chance of optimization (for example modeling of stack using graph structured stack) so that time required to recognize the input would be reduced which is not possible when the algorithm is implemented using recursive descent parsing style. It would be interesting to see the impact of stack used in the proposed algorithm on the use of memoization in packrat parsing algorithm. If somehow this reduces the memory required for memoization, then packrat parsing algorithm will have linear parsing time without sacrificing the storage.

REFERENCES

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

[2]. Ford, B. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, assachusetts Institute of Technology, September 2002.

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

[4]. Ford, B. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy, 14–16 January 2004), N. D. Jones and X. Leroy, Eds.,ACM, pp. 111–122.

[5]. Grimm, R. Rats! – an easily extensible parser generator http://www.cs.nyu.edu /rgrimm/xtc/ rats.html.

[6]. Grimm, R. Practical packrat parsing. Tech. Rep. TR2004-854, Dept. of Computer Science, New York University, March 2004.

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

[8]. Redziejowski, R. R. Some aspects of Parsing Expression Grammar. *Fundamenta Informaticae 85*, 1–4 (2008), 441–454.

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

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

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

[12] Elizabeth Scott and Adrian Johnstone.GLL Parsing**,** Electronic Notes in Theoretical Computer Science 253 (2010) 177–189.

