Packrat Parsing with Dynamic Buffer Allocation

Nikhil Mangrulkar[1*], Kavita Singh[1], Mukesh Raghuwanshi[2]

[1]*Yeshwantrao Chavan College of Engineering, Nagpur, India.*
[2]*G. H.Raisoni College of Engineering & Management, Pune, India.*

Abstract

Packrat parsing is a type of recursive decent parsing with guaranteed liner time parsing. For this, memoization technique is implemented in which all parsing results are memorized to avoid repetitive scanning of inputs in case of backtracking. The issue with this technique is large heap consumption for memoization which out weigh the benefits. In many situations the developers need to make a tough decision of not implementing packrat parsing despite the possibility of exponential time parsing. In this paper we present our developed technique to avoid such a tough choice. The heap consumption is upper bounded since memorized results are stored in buffer, capacity of which is decided at runtime depending on memory available in the machine. This dynamic buffer allocation is implemented by us in Mouse parser. This implementation achieves stable performance against a variety of inputs with backtracking activities while utilizing appropriate size of memory for heap.

Keywords: Packrat parsing, Parsing Expression Grammars, parser generators, backtracking.

*Corresponding author Email: nmangrulkar@ycce.edu

1. Introduction:

A recursive descent parsing algorithm with backtracking needs to be designed carefully or else, backtracking would cause the time required for parsing to grow exponentially with the input length [1].Packrat parsing, introduced by B. Ford in 2002, is a technique for implementation of recursive descent parser along with backtracking [2]. The primary concept used in packrat parsing is of memoization which works by storing all intermediate results parsed at each distinct position in the input stream. Since each non terminal is called only once, packrat parsing avoids the possibility of exponential time for parsing. Packrat parsers are mainly used with Parsing Expression Grammars or PEGs which also introduced by B. Ford [3-4]. The Context Free Grammars (CFG) can produce multiple tree parse trees which makes it powerful and capable of representing natural languages, which is unnecessary for computer-oriented language since the programming languages are supposed to be deterministic. PEGs avoid ambiguity in grammars by not allowing the ambiguous productions in the rules at first place.

The linear time parsing guarantee given by Packrat parsers comes at a cost of large heap consumption as every intermediate result is stored in memory to avoid reevaluation. As pointed out by some of the researchers, this cost of heap consumption is so high that it nullifies the benefit offered by memoization [5]. This caused many developers to drop the idea of using memoization till recent years. Of late asthe memory available increased, few researchers implemented the PEG based parsers like Rats! and Mouse.

The purpose of this paper is to make optimal use of the resource available by using fixed minimum buffer size if the system on which program is executed is constrained for memory and if sufficient memory is available, use the portion of it for memoization.

Our idea behind dynamic memory allocation of Packrat parsing is based on the fact that if available resources; specifically, memory; is not used optimally, we are failing to take the advantage that Packrat parser offers. If the amount of memory available is not large enough and still large memory size is allotted for memoization, it may hamper the performance of system as CPU will get busy in paging and garbage collection. If too small size of memory is used for memoization even if amount of memory available is large enough, then we will be missing out on implementing linear time parsing.

We take an approach to allot the memory for storing the intermediate results optimally. If sufficient memory is available, some percentage of it will be used for memoization. If free space available is not enough, fix size of memory will be allotted for this purpose. Our experimental results demonstrated promising results.

2. Memoization

D. Michie et. al. [6] introduced the concept of memoization as a method for machine learning. By storing the calculated results, the machine “learned” it. Next time whenever same calculation was requested, the machinemerely “remembered” it by looking up the previously stored results. Stack data structure was used for storing those results which made look-up process linear. The results are merely pushed on top of the stack making insertions of results constant.

In packrat parsing, the results are 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 [7]. 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 since the result only needs to be evaluated once. This gives packrat parsing a linear parsing time in relation to the length of the inputstring (given that the access andinsertion operations in the matrix are done in constant time).

Nariyoshi Chida, *et. al.,* presented Linear Parsing Expression Grammars (LPEGs) in their work in 2017 [8]. LPEGs are a subcategoryof PEGs which are equivalent to DFAs. It is PEGs with syntax limited to right linear.For formulating LPEGsthey excluded patterns of recursive nonterminalswhich were followed by expressions. By this, the syntax of LPEGs became limited to right linear. Robert Grimm introduced Rats! as described in [9]. It translates grammar specification into programming language source code in Java. Redziejowski RR implemented aoperating system independent tool "Mouse" also written in Java[10]. This tool transcribes PEG into executable parser. In this tool they also provided a feature of providing semantics, because of which this tool is not only transcribing the PEGs but also can solve the expressions.

Kimio Kuramitsu presented Packrat parsing with elastic sliding window [11]. He mentioned that there are many claims available in literature that questions the benefits of memoization over its overhead. He also stated that few literatures also claims that plain recursive descent parsers are sufficiently competitive, or even faster than packrat parsers. He pointed out that the performance of Packrat parsing depends on two factors: i) Backtrack activity and ii) Effectiveness of Memoization. He has implemented technique based on sliding window protocol used in networks to handle unlimited length of incoming data. Author have used array data structure to implement the elastic window for memoization of size $W \times N$, where W is window size and N is number of non-terminals. Hashing based index is used to locate the memoized result in the array. From their experimental results they demonstrated that the heap consumption of their technique was less than the traditional Packrat parser. As elaborated [11]; the elastic sliding parsermaintained constant heap consumption, whereas simple Packrat parser had linear consumption.

Manish Goswami et. al.[12] implemented Packrat parsing based parser using Stack data structure. They modified the data structure sued by Packrat parsing based tool Mouse, which is originally implemented using array. They have compared their implementation with Rats!, an another tool based on Packrat parsing and the original Mouse tool. They showed that their modified implementation of Mouse parser performed better than both the other tools.

4. Parsing Expression Grammars:

Parsing Expression Grammars (PEGs) is a newer way of representing rules that can be used for defining syntax for machine-oriented languages. A prioritized choice operator '/' is used by PEGs for specifying possible expansions for a nonterminal. The '/' operator gives multiple patterns to be examined orderly, using the first effective match. A rule using PEGs can be defined by using form represented in Figure 1.

$$E \leftarrow xy / x$$

Figure 1: A Parsing Expression Grammar

In Figure 1 above, *E* is nonterminal which can be reduced if input matches *xy* or *x* which are terminals. Options at the right-hand side of rule can be a terminal, a non terminal or combination of both. To begin with, input will be checked for match with *xy* and only if there is no match, *x* will be checked for a possible match. For this '/' operator is used which as mentioned earlier, works as a priority operator, deciding the sequence of matching. It is worth mentioning here that if the first-choice symbol can reduce to nonterminal, the next options would not be checked in any case. Hence, it is required to decide the sequence of symbols in rule very carefully.PEG can be seen as a proper representation of a top-down parser.

5. Methodology:

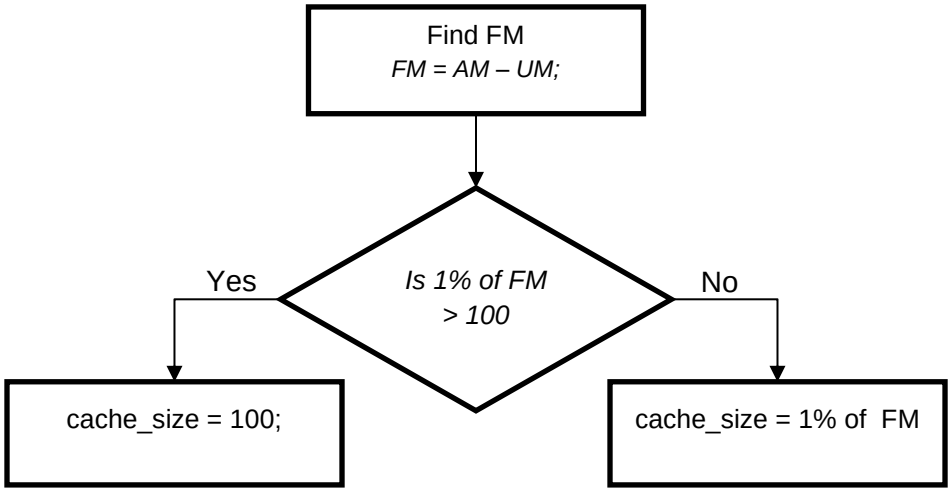

Figure 2: Graphical representation of our approach to decide minimum size of cache

Given below is our algorithm to select minimum size of cache.

Step 1: *Identification of free memory:*
Find FM:
FM= AM-UM;

Step 2: *Setting cache size:*
Initialize cache_size:
If 1% of FM > 100
*set cache_size = 100**
else
set cache_size = 1% of FM

Where, *FM* is free memory
*AM*is Available Memory
*UM*isUsed Memory

Our algorithm is a two-step approach in which step 1 is to identify the free memory available in the system. Once the FM (free memory) is calculated, step 2 is to set the cache size. Here we check the amount of FM available. If 1% of FM is greater that 100 cache size, then maximum cache size is set to 100. If it is less than 100, 1% of FM is set as cache size. The value 100 has been taken into consideration based on the fact that parsing a sufficiently large input string would also not require more than 100 memorization. Also for every 1 byte of input about 400 bytes of memory is required for memorization which is the reason we are selecting only 1% of free memory to avoid excessive use of memory space for memorization. Because of our this approach, it is guaranteed that no more than 1% of free memory will be utilized for memorization if we are constrained for memory and if memory is available, we would take maximum benefit from it by storing every intermediate result.

6. Results and Discussion:

We have implemented our algorithm by modifying current Mouse tool. All the executions has been tested on system having Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz and 8.00 GB (7.89 GB usable) RAM. Figure 3 shows the output generated by the program for "123 + 4567 + 789" as input given to the tool.

```
> 123 + 4567 + 789
5479.0

61 calls: 38 ok, 17 failed, 3 backtracked.
0 rescanned, 3 reused.
backtrack length: max 4, average 3.3.

Backtracking, rescan, reuse:

procedure        ok  fail  back  resc reuse totbk maxbk at
------------ ----- ----- ----- ----- ----- ----- ----- --
Digits            3     0     0     0     3     0     0
Number_0          0     0     3     0     0    10     4 After '123 + '

>
```

Figure 3: Output generated for input 123 + 4567 + 789

It can be observed in Figure 3 above, there were 3 situations where backtracking was done by the tool for the given input. Maximum backtracking required was of 4 places occurred after consuming input tokens "123+".

We are not presenting the performance comparison of our program with other tools like original Mouse and Modified Parser [11] as we need to compare the performance on systems with different size of RAM to verify the effectiveness of our approach. Our implementation is stable and is performing promisingly under varying size of input.

7. Conclusion:

Packrat parsing is a technique for implementing recursive descent parsers with backtracking which gives guaranteed linear time parsing. However, many researchers have questioned the effectiveness of packrat parsing due to overhead of large heap consumption for memorization. Dropping memorization because of this overhead is not a proper solution specially if there is availability of resources which could give us optimum utilization and guaranteed liner time parsing. This paper presented an approach to select the optimum size of heap to be used for storing memorized results. A two-step process is presented which ensures that if resources are available, they will be optimally utilized to provide best performance and along with guaranteed linear time parsing and, if the system is constrained for resource, bare minimum resource will be utilized for memorization to ensure memory is available for smooth functioning of the system.

Conflict of interest: The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1]. Birman A, Ullman JD. "Parsing algorithms with backtrack". In11th Annual Symposium on Switching and Automata Theory (swat 1970) 1970 Oct 28 (pp. 153-174). USA IEEE. https://doi.org/10.1109/SWAT.1970.18

[2]. Bryan Ford, "Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking", Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, USA, 2002. https://doi.org/10.1145/583852.581483

[3]. Ford B. "Parsing expression grammars: a recognition-based syntactic foundation". InProceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages 2004 Jan 1 (pp. 111-122). https://doi.org/10.1145/982962.964011
[4]. B. Ford. "Packrat Parsing: Simple, Powerful, Lazy, Linear Time - Functional Pearl" In Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP, Pittsburgh, Pennsylvania, USA, 2002. ACM SIGPLAN Notices37/9 (2002) 36-47. https://doi.org/10.1145/583852.581483
[5]. Becket R, Somogyi Z. "DCGs+ memoing= packrat parsing but is it worth it?"InInternational Symposium on Practical Aspects of Declarative Languages 2008 Jan 7 (pp. 182-196). Springer, Berlin, Heidelberg.(LNCS, volume 4902). https://doi.org/10.1007/978-3-540-77442-6_13
[6]. MICHIE, D. "Memo Functions and Machine Learning". Nature 218 (1968) 19-22. https://doi.org/10.1038/218019a0
[7]. 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.
[8]. Nariyoshi Chida, Kimio Kuramitsu, "Linear Parsing Expression Grammars", Preprint Version, Sep 2017. https://doi.org/10.1007/978-3-319-53733-7_20
[9]. Grimm R. "Practical packrat parsing" Technical Report TR2004-854, New York University; 2004 Mar.
[10]. Redziejowski RR. "Mouse: from parsing expressions to a practical parser". InConcurrency Specification and Programming Workshop 2009 Sep.
[11]. Kuramitsu, Kimio. "Packrat parsing with elastic sliding window" Journal of information processing 23/4 (2015) 505-512. https://doi.org/10.2197/ipsjjip.23.505
[12]. Manish M. Goswami, M.M. Raghuwanshi, Latesh Malik, "Performance Improvement of Stack Based Recursive-Descent Parser for Parsing Expression Grammar", International Journal of Latest Trends in Engineering and Technology, 6/3 (2016) 302-309.