IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727
PP 19-24
www.iosrjournals.org

An innovative solution to avoid problem associated with top down parsing

Mr. Rakesh Yashwant Gedam[1], Dr. Rakesh Ramteke[2], Dr. Pravin Gundalwar[3]
[1](mr.rakeshgeam1@gmail.com, *Department of computer science, G. H. R. I. I. T., RTM Nagpur University, India)*
[2](.rakeshj.ramteke@gmail.com, *HOD,Department of computer science & IT, NMU, Jalgaon, India)*
3(p_gundalwar@yahoo.com, *Dr. D. Y .Patil IMC Akurdi, Savitri Bai Fule Univesity Pune .India* .)

Abstract: *Compiler has different phases or passes. Each phase having important significance but parser does very crucial role in compilation process. This paper has discusses basic issues in parsing & generation of parse tree or derivation tree for different token identifier by using lexical analyzer and then pass to parser for further process to get intermediate code generation. This paper explains the types of parsers to generate string of executable code. Each approach having significant benefit and drawback, and also suffer with some bottleneck. This paper focuses on certain problem in top down parsing approach & its efficient solution to generate the machine dependent effective code.*

Keywords: *Parser, Lexical Analyzer, Parse Tree, backtracking, left recursion, left factorization*

I. Introduction :

Compiler translate the source code to machine code equivalent code by mean of series of different phases **.**The second phase of the compiler is syntax analysis or parsing. [1].A parsing or syntax analysis is a process which takes the input string W and produces either a parse tree (syntactic structure) or generate syntactic error. Syntax analyzer (parser) basically checks for syntax of the language .Syntax analyzer which takes the token from lexical analyzer and group of them in a such a way that some programming structure (syntax) can be recognized. After grouping the token if at all, any syntax cannot be recognized then syntactic error will be generate and the overall process is called syntax checking of the programming language .

For example sum= num +10; the programming statement is first given to lexical analyzer . The lexical analyzer will divide it into group of tokens. The syntax analyzer takes the token as a input and generate a parse tree. The parse tree dawn above is for some programming statement. It has shown how the statement gets parsed according to their syntactic specification.

II. Current work on Parsing Techniques

There are two parsing technique , these parsing technique work on following principal .

1. The parser scan the input string from left to right and identifies that the derivation is leftmost or rightmost
2. The parser make use of production rule for choosing the appropriate derivation . The different parsing technique use the different approaches in selecting the appropriate rule for derivation. And finally a parse tree is constructed

2.1 Type of Parser

Bottom Up Parser : When the Parse tree can be generated from leaves to root then such type of parser is called *Bottom Up Parser* ,this the parse tree is built in bottom up fashion

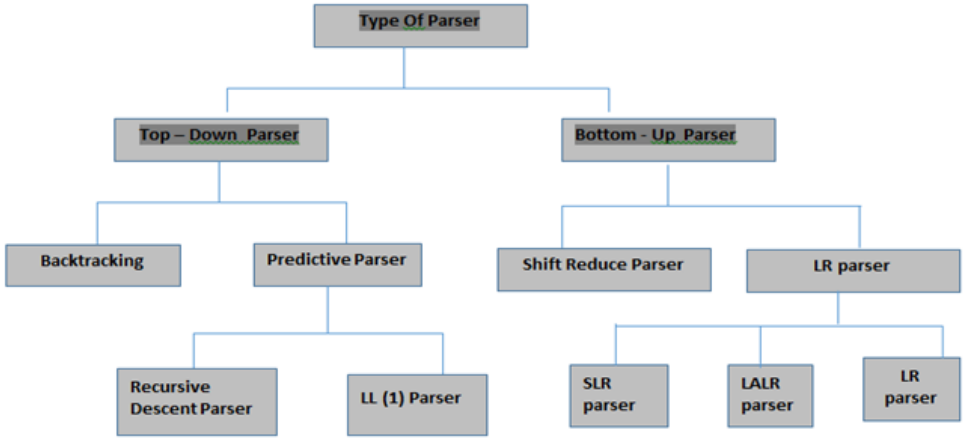
Top Down Parser :

When the Parse tree can be generated from root and expanded to leave then such type of parser is called **Top** *Down Parser* ,as a name implies parse tree can be built from top to bottom

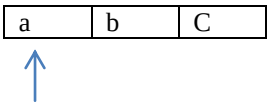
The derivation terminal when the required input string terminate. The leftmost derivation matches this requirement . the main task of top down parser is to find the appropriate production rule in order to produce the correct input string .we understand the process with the help of example .

Consider the grammar

S →a P c ,P →b d | b.



Consider the input string " abc " is show



Input Buffer

Now we construct the parse tree for above grammar deriving the input string . And for this derivation we will make use of top down approach

step 1

.

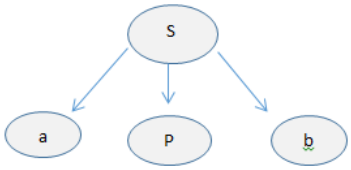


Fig (a)

step 2

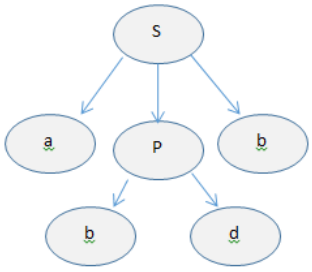


Fig (b)

Step3.

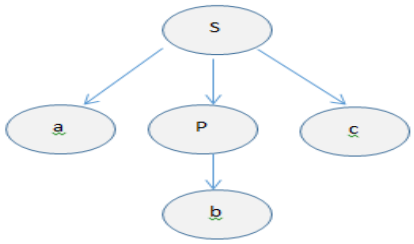


Fig (c)

III. Basic Issue in parsing:

There are three important issue in parsing in front of any parser designer

i) Specification of syntax, its most critical issue in parsing for any parser generator . specification of syntax tells us how to write any programming statement .

Guideline for deal with this issue is

a) This specification should be precise and unambiguous
b) This specification should be in detail , i.e should cover all the detail of programming language
c) This specification should be complete which checked by using Context Free Grammar (CFG) which generate the language L .

ii) Representation of input after parsing, another important issue in parsing is representation of input after parsing, because all subsequence phase of compiler take the information from parse tree being generated . the information suggested by any input programming statement should not be differed after building the syntax tree for it

iii) Parsing Algorithm, is another most crucial issue because based on which we get the parse tree for given input.

IV. Solution for top down parsing:

There are certain problem in top down parsing . in order to implement the parser we need to eliminate those problem . Let Us discuss with those problem and how to remove them

4.1 Backtracking : Backtracking is the technique in which for expansion of non terminal symbol we choose one alternative and if some mismatch occur we try another alternative if any

For example .Consider the grammar S →a P c ,P →b d | b. string " abc "

step 1
.

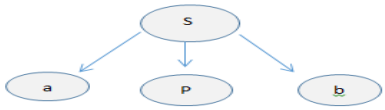


Fig (a)

step 2

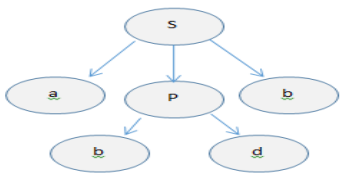


Fig (b)

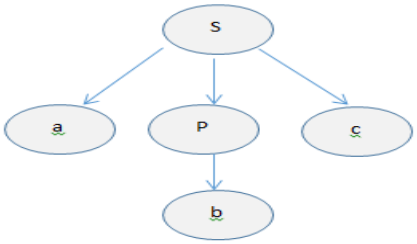
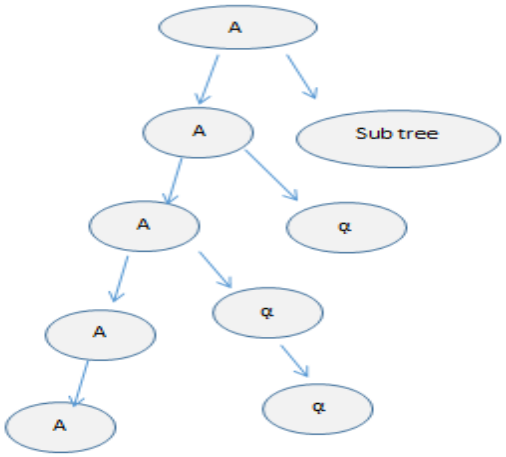


Fig (c)

If for Non terminal there are multiple production rule beginning with same input symbol then to get the correct derivation we need to try all these alternative. secondly, in backtracking we need to move levels upward in order to check the possibilities. This increase lots of overhead in implementation of parsing. hence its necessary to eliminate the backtracking by modifying the grammar.

4.2 Left Recursion: The process of calling itself again and again it called recursion. The left recursion grammar is a grammar which is given below

+ +
A → A α here - → means deriving the input in one or more steps . The A can be non terminal and α denote input string .
Left Recursion is present in the grammar then it creates series problems . because of left recursion the top down parser can enter the infinite loop .



Thus expansion of A cause further expansion of A and due to generate of A, Aα,Aαα,Aααα,……the input pointer will not be advanced . this caused major problem in top down parser and therefore eliminating of left recursion is must .
To eliminating left recursion we need to modify the grammar .
Let , G be a CFG having production rule with left recursion

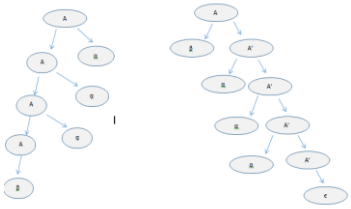
A → A α
A→β ---------- (1)

Then Eliminating left recursion by re writing the production rule as
A→βA'
A'→αA' -----(2)
A'→ϵ

Input

B	A	α	A	

String:

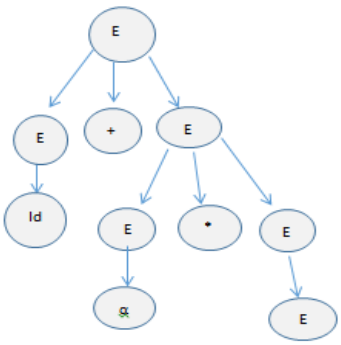


4.3 Left factoring :
If the grammar is left factored then it became suitable for use. Basically left factoring used when it is not clear that which of two alternative is used to expand the non-terminal . By left factoring we may able to retrieve the production in which the decision can be deferred until enough of input is seen to make the right choice.
In general if
A→αβ1| αβ2 is a production then it is not possible to take decision whether go for 1st rule or 2nd rule .
Then Left factorization method is A→αA', A→β1|β2. For example : consider the following grammar
S→iEtS|iEtSeS|a
E→b.
The left factored grammar become
S→iEtSS'|a

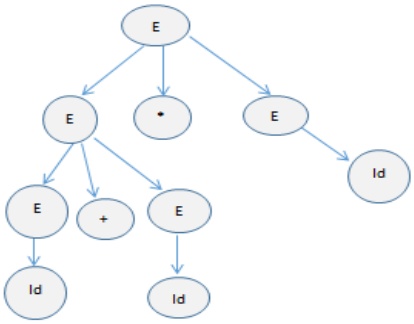
S'→eS|ϵ
E→b.

4.4 Ambiguity: The ambiguous grammar is nnot desirable in top down parsing . Hence we need to remove the ambiguity from the grammar if its present

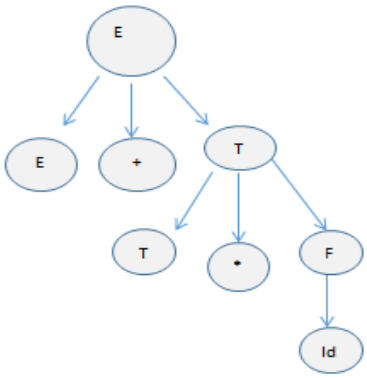
Example : E→E+E|E*E|id is an ambiguous grammar . We will design the parse tree for id +id*id follows



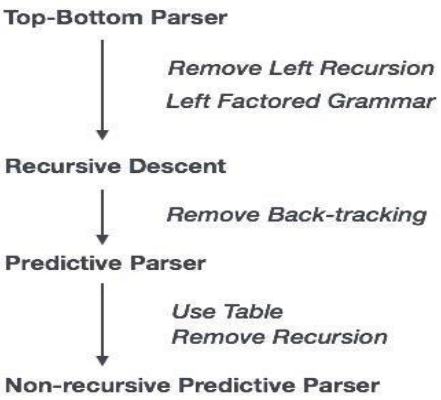
Parse Tree 2.



To remove the Ambiguity we use one rule : if grammar has left associativity operator (such as +,- ,*, \)then induce the left recursion and if grammar has right associative operator (exponential operator) then include the recursion.
The unambiguous grammar is
E→ E + T | T
T→ T *F |F
F→id



Summary for method evaluation :-



V. Cconclusion

We observed that top down parser is suffer with backtracking , left recursion , left factorization and ambiguity with my greatest bottleneck to generate the intermediate code with good efficiency . This Paper give us simple approach and idea to how to deal with those problem. Top Down parser take the token and process through elimination of left recursion and left factored grammar give us recursive decent parser where we remove backtracking yield predictive parser. Next we remove recursion and stored information in symbol table and finally get a non-recursive predictive parser

References

[1]. Alfred_V._Aho,_Monica_S._Lam,_Ravi_Sethi,_Jeffrey" Compilers Principal , techniques , tools second edition
[2]. E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In CGO '06.
[3]. Earley, J., "An efficient context-free parsing algorithm," Comm. A CM
[4]. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. POPL '89
[5]. J. Guo, F. HÃijffner, E. Kenar, R. Niedermeier, and J. Uhlmann. Complexity and exact algorithms for vertex multicut in interval and bounded treewidth graphs. European Journal of Operational Research, 186(2):542 – 553, 2008.
[6]. Principal of Compiler design by A.A. puntabekar
[7]. Backus, J.W, "The syntax and semantics of the proposed international algebraic language of the Zurich-ACM-GAMM Conference," Proc. Intl.
[8]. Conf. Information Processing, UNESCO, Paris R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.
[9]. Birman, A. and J. D. Ullman, "Parsing algorithms with backtrack," Information and Control 23:l (1973)
[10]. Hopcroft, J. E., R. Motwani, and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Boston MA, 2001
[11]. Sale, Arthur H. J. [1979]. A note on scope, one-pass compilers, and Pascal. Pascal News,15:6263.
[12]. object programs. IBM Journal of Research and Development, 24(6):660676.
[13]. In Popplewell, C.M., editor, Information processing 1962, pages 524,525. North-Holland,Amsterdam, NL.