ALTERNATIVE METHOD TO CHECK AMBIGUITY IN CONTEXT FREE GRAMMARS

L. A. Alamirew*

Information Technology Department, College of Technology, Debre Markos University, Debre Markos, P.O.Box 269, Ethiopia

Article History: Received 13.01.2017; Revised 11.12.2017; Accepted 26.12.2017

ABSTRACT

Compilation of source code from different papers and books is a tedious and lengthy process. The result of the first phase of compilation, which is tokens, is given to the next phase called parsing which outputs a parse tree (also called syntax tree) to a grammar for a given input string. If there is more than one parse tree which are different with one another, then it calls the grammar as ambiguous grammar, or in short there exists an ambiguity in the grammar This paper tries to present an alternative method for checking the so called ambiguity problem in context free grammars by having an infix tree structure for a given input and comparing it with the corresponding parse trees and obtained from the given grammar in a top-down parsing approach.

KEYWORDS*: Ambiguous grammar; Top-down parsing; Parse tree; hierarchical; grammar*

1.0 INTRODUCTION

Different papers and books define and describe the process of compilation done by a compiler clearly. As it is stated in previous studies, source code is converted and changed to a target machine code, there are a number of steps (phases) under go arrangement by a compiler (Dube & Feeley, 2000; Aho et al., 2007; Costa et al., 2005; Luca et al., 2013; Richard & Hafiz, 2006). Of these phases, the second is the syntax analysis phase, done by syntax analyzer, which is next to lexical analysis one. As it is described there, this phase has actually different, but the same meaning, naming like parsing and hierarchical

* Corresponding Email: lame2002@gmail.com

analysis. The output obtained from the first phase of compilation (also called lexical/ linear analysis or scanning) which is, in fact, the sequence of characters called tokens are grouped hierarchically into nested collections with collective meaning. These tokens resulted from lexical analysis are given to the syntax analysis phase of compilation, and checks whether that string can be generated by the grammar for the source language or not and produce phrases which are usually expressed by a parse/syntax tree.

As far as hierarchical analysis is concerned, two main standard approaches exist for parsing, namely top-down and bottom-up parsing. For the first type of parsing, i.e. top down parsing, for a given input string, parse tree is constructed beginning from the starting root non-terminal and then it goes to the leaves. In this strategy, by the help of reworking production rules of a grammar, the highest level of the parse tree (root) is first parsed and then it proceeds to the leaves of the tree. In an easiest way, as its name implies, one can say that, in a top down parsing, a parse tree is built or constructed from top to down direction of the tree in which for deriving the given input string the root nonterminal will be expanded. The language accepts the string when the string can be derived, and not otherwise. The benefit of this type of parsing is that, using top down mechanism can easily create the parser manually.

On the other side, the second category of parsing, namely bottom up parsing, does in a reverse direction with the top down strategy. For a given input string, a bottom up parser built the parse tree starting from the leaves of the tree and proceeds to the starting non-terminal root. Unlike top down parsing in which the input string is expanded, here in bottom-up parsing, the input string will be reduced to the starting non-terminal symbol, and parsing succeeds when the whole input has been replaced by the start symbol. The profit for bottom up parsing is that, the large class of grammars and translation schemes can be handled by it.

The rest of the document is organized as follows. In section 2, some discussion about ambiguity and context free grammar is presented. Section 3 presents all about the proposed method of this article. In section 4, discussion about a comparison of the proposed method with an already existed methods is presented. At the end, the conclusion of the paper is stated in section 5.

2.0 CONTEXT FREE GRAMMAR AND AMBIGUITY

According to previous researchers, the so called context free grammar (shortly named as grammar) describes sets of strings (i.e. languages) and defines structure on these strings. In the language it defines, the syntax elements and rules for composing them from other syntactic elements are specifically allowed. A set of rules to identify non-terminals to obtain from other terminals and non-terminals, is useful in specifying the syntactical structures of a programming language (Dube & Feeley, 2000; Aho et al., 2007; Costa et al., 2005; Luca et al., 2013; Richard & Hafiz, 2006).

According to these literatures, there exist four components for a context free grammar, namely terminal, non-terminal, start and production rule described as follows:-

1. Terminals: these components are the basic & elementary symbols that can't be replaced by anything of a grammar from which strings are obtained. They are also named as tokens, and it is assumed that they are the first component of tokens resulted from the early phase of compilation done by the lexical analyzer. Usually lowercase letters are in use to represent terminals in production.

2. Non-Terminals: these components, also called syntactic variables, represent a set of strings of terminals, and they must be replaced by other things in a given grammar. Non-terminals are used to specify strings and mostly they are represented by uppercase letters in production.

3. Production rules: these are the grammatical rules that are used for specifying the manner of forming strings from combining terminals and non-terminals. The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production rule consists of non-terminal followed by a string of non-terminals and terminals.

   In production, there exists an arrow (→), the left side of this arrow consists of non-terminals, also called head of production, and the right hand side of the arrow contains a sequence of terminals and/or non-terminals, also called body of production.

4. Start symbol: from non-terminals of a grammar has, one is designated as start symbol that specify the language, and the rest is used for specifying the string.

There might be a problem of ambiguity in grammars. There are different occasions to state whether this ambiguity problem exist for a given grammar or not. One is, when a grammar has more than one parse/syntax tree for a given input. The other option is that, when this grammar has more than one rightmost derivation for the given input. At the end when a grammar has more than one left most derivation. Whenever one of these options comes to true, then the grammar is ambiguous. For compilation purpose, unambiguous grammars have to be designed due to the fact that a string with more than one syntax tree frequently has more than one meaning. In fact, this ambiguity problem for a grammar can be resolved by changing and rewriting the grammar itself.

Let us have a look for the above idea using a simple example. For our example, take the following grammar and input string.

E → E – E | E * E | a | b | c ; and input string given is " **a – b * c** ".

First of all, the production rule for the above grammar will be the following

E → E – E
E → E * E
E → a
E → b
E → c

In this grammar, the start symbol is E; terminals are -, *, a, b, c; and non-terminal is E. Left side of the arrow is head of the production, and right side of the arrow is the body of the production. Second, let us now try to draw the parse tree for the grammar and given input string. Before drawing the parse tree, let us describe what parse tree and derivation are.

As it is described in Torben (2010), there exists two kinds of derivations namely left most and right most derivations. If choosing the leftmost non-terminal in

each derivation step, this derivation is leftmost derivation. If choosing the rightmost non-terminal in each derivation step, this derivation is rightmost derivation. Accordingly, left most derivation always rewrites the leftmost nonterminal and that of rightmost derivation always rewrites the rightmost nonterminal.

It has been stated, a parse tree can be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order (Aho et al., 2007). Thus, a parse tree pictorially shows how the start symbol of a grammar derives a string in the language. Moreover, each interior node of a parse tree is labeled by some non-terminal A, and that the children of the node are labeled, from left to right, with the symbols on the right side of the production by which this A was replaced in the derivation. In addition, the root of the parse tree is labeled by the start symbol and the leaves by non-terminals or terminals and, read from left to right, they constitute a sentential form of the tree. When come to drawing the parse tree, it can be done using left most derivation and/or right most derivation.

i. Using left most derivation (LMD)

$E \rightarrow E * E$
$E \rightarrow E - E * E$
$E \rightarrow a - E * E$
$E \rightarrow a - b * E$
$E \rightarrow a - b * c$

Using the rightmost derivation, the parse tree is shown in the Figure 1(a).

ii. Using right most derivation (RMD)

$E \rightarrow E - E$
$E \rightarrow E - E * E$
$E \rightarrow E - E * c$
$E \rightarrow E - b * c$
$E \rightarrow a - b * c$

Using the rightmost derivation, the parse tree is shown in Figure 1(b).

A grammar is said too ambiguous when it permits several different parse trees for some strings (Aho et al., 2007). Or in another way, ambiguous grammar is the one which produces more than one parse tree for some sentence. Alternatively, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

As seen in Figure 1, these two different parse trees, which show as a graphical representation of a derivation for a grammar of a given input, are different with one another. Hence, according to the above descriptions, it can conclude that this given grammar is ambiguous since it has got two different parse trees for the same input string.

Figure 1: Two possible parse trees for string a – b * c

3.0 PROPOSED ALTERNATIVE METHOD FOR CHECKING AMBIGUITY

The headache of checking ambiguity of a grammar for the current top down parsing is that, one has to draw and find all the possible parse trees and compares and contrasts with one another. Finally, if there is a difference and variation between or among the possible parse trees, there exists an ambiguity in a given grammar for a given input string. But, this paper aims to have a better mechanism of checking this problem of ambiguity in the way that, first of all, the input string has to be described in the form of tree data structure (expression tree) and later on, this tree structure is helpful for comparing with the parse tree.

An expression tree presents a tree whose leaf nodes are operands and internal nodes are operators of a postfix expression kind. In so doing, it can present the parse

tree in the form of expression tree structure in the way that values and variables of terminals are kept on leaf nodes and operators of terminal symbols on internal nodes of the tree. Hence, if the expression tree structure of a parse tree differs with the expression tree structure of an input string, then the grammar is ambiguous for a given input string. Unlike the current method of checking ambiguity, for this new approach, it may not be required to have all possible parse trees of a grammar for a given input string. If there is a parse tree structure which differs with the expression tree structure of an input string, then it can conclude that the given grammar is ambiguous over the given input string.

The algorithm for doing this is the following: Convert the input string to tree data structure. Then, take the equivalent tree structure of a parse tree and check with the tree structure of an input string. If there is a difference, then the grammar is ambiguous and end up here. Otherwise, take another parse tree and check. Finally, do this until a difference is encountered or all possible parse trees are checked.

If all possible parse trees are checked and no difference is obtained with the expression tree structure of an input string, then the grammar is free from ambiguity over the given input string.

4.0 DISCUSSION OF PROPOSED METHOD

To have a better understanding of the proposed alternative method for checking ambiguity in a grammar, let us take the above example of grammar & input string, and discuss it.

Take the following grammar $E \rightarrow E - E \mid E * E \mid a \mid b \mid c$; and
Input string "a – b * c "

Accordingly, the production rule will be

$E \rightarrow E - E$
$E \rightarrow E * E$
$E \rightarrow a$
$E \rightarrow b$
$E \rightarrow c$

In order traversal the above tree depicted in Figure 2 (in the order of left, parent & right) gives us the infix notation; i.e. **a – b * c** which is the given input string.

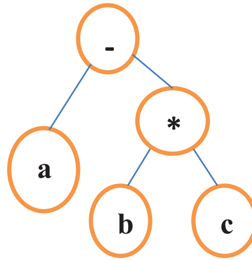

Figure 2: Tree structure for input string **a – b * c**

Now take the second parse tree of the grammar for the given input string, and draw the corresponding tree structure for it as shown in the following figure.

As seen from the diagrams, the equivalent tree structure depicted under (b) for the parse tree depicted under (a) of the above figure differs from the tree structure of the input string shown under same Figure 3. Therefore, using this new method of checking ambiguity, the grammar has now over the given input string is ambiguous. The same is true for the next example.

Consider the following grammar $E \rightarrow E / E \mid E + E \mid a \mid b \mid c$; and Input string "a / b + c".

Accordingly, the production rule will be

$E \rightarrow E / E$

$E \rightarrow E + E$

$E \rightarrow a$

$E \rightarrow b$

$E \rightarrow c$

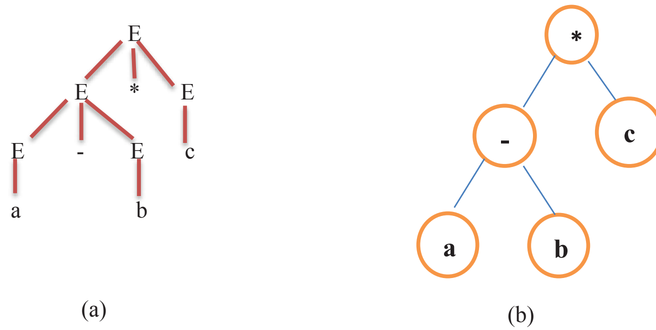

Figure 3: Parse tree and its corresponding tree structure

In order, traversal the tree (in the order of left, parent & right) gives us the infix notation; i.e. a / b + c which is the given input string.

There is a variation between input string tree structure shown in Figure 4 and the corresponding tree structure (b) for parse tree (a) in Figure 5. Therefore, according to our proposed approach, since there exists a difference in the tree structures, the grammar is ambiguous for a given input string.

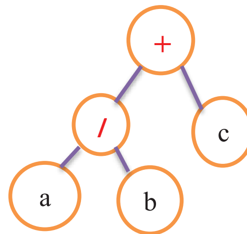

Figure 4: Tree structure for input string **a / b + c**

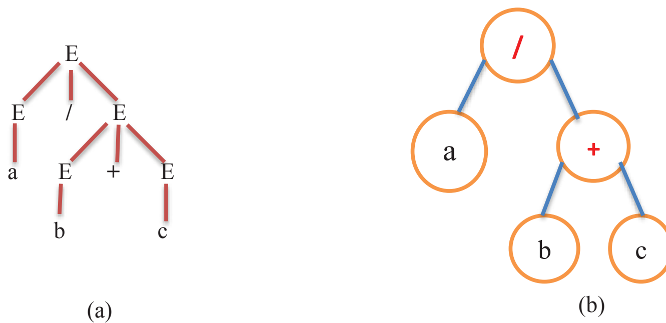

Figure 5: Parse tree and its equivalent tree structure

5.0 CONCLUSION

The grammar is ambiguous whenever it results two or more different parse trees for a given input string. The current method checks this problem by drawing all the possible parse trees of a grammar for an input string. In this paper, it is shown a possible alternative mechanism to check whether the ambiguity problem presents in a grammar to a given input string. Having an expression tree structure of the input string and equivalent tree structure for parse tree of a grammar can also be used as a method to check the so-called ambiguity problem syntax analysis during the compilation process. As mentioned earlier, in the best case, the existing method of checking ambiguity requires two parse trees, whereas the proposed new method requires one parse tree for a grammar in order to determine whether an ambiguity exists or not. In the worst case, both the current and proposed methods of checking an ambiguity require n possible parsed tree for a grammar it has.

ACKNOWLEDGEMENT

The authors are grateful to Debre Markos University for the technical support provided for this research work.

REFERENCES

Dube, D., & Feeley, M. (2000). *Efficiently building a parse tree from a regular expression*, Acta Informatica, 37(2): 121-144.

Aho, A.V., Lam M.S, Sethi, R., Ullman J.D., (2007). Compilers Principles, Techniques, & Tools + Gradiance.

Costa, F., Frasconi, P., Lombardo, V., Sturt, P., & Soda, G. (2005). *Ambiguity Resolution Analysis in Incremental Parsing of Natural Language*, IEEE Transactions On Neural Networks, 16(4): 959-971.

Luca B., Stefano C. R. and Angelo M. (2013). *Parsing methods streamlined*, Politecnico di Milano, Department di Elettronica, Informazione e Bioingegneria.

Richard A. Frost and Rahmatullah Hafiz. (2006). *A New Top-Down Parsing Algorithm to Accommodate Ambiguity and Left Recursion in Polynomial Time, ACM SIGPALN*, 41: 46-54.