COLLEGE OF ENGINEERING AND MANAGEMENT , KOLAGHAT

NAME: SANTU JANA

UNIVERSITY ROLL: 10700121127

COLLEGE ROLL:CSE/21/L-146

TITLE OF THE REPORT: ELIMINATING AMBIGUTITY

-: CONTENTS:-

Abstract

This paper details the study of an existing ambiguity detection algorithm for grammars. This algorithm is sound as well as complete to detect ambiguous and unambiguous grammars. In this paper we are exploring ambiguity, existing approaches to deal with ambiguity, their comparisons, challenges, recent trends, etc., and cite the important literature on these elements. The author tested one of the existing algorithms using a set of eighty four grammars includes ambiguous, unambiguous, and grammars derive empty strings. The observation has been made available to show the variations in execution time in finding ambiguous grammars and unambiguous grammar by the studied algorithms.

INTRODUCTION

A grammar is said to be ambiguous if there exists more than one leftmost derivation or more than one rightmost derivation or more than one parse tree for the given input string. If the grammar is not ambiguous, then it is called unambiguous.

If the grammar has ambiguity, then it is not good for compiler construction. No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.

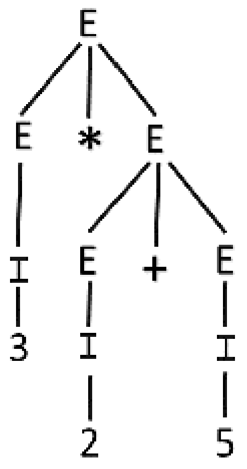
Example 1:

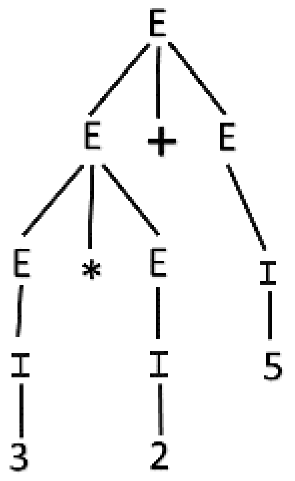
Let us consider a grammar G with the production rule

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow \varepsilon \mid 0 \mid 1 \mid 2 \mid ... \mid 9$

Solution:

For the string "3 * 2 + 5", the above grammar can generate two parse trees by leftmost derivation:





Since there are two parse trees for a single string "3 * 2 + 5", the grammar G is

ambiguous.

Example 2:

Check whether the given grammar G is ambiguous or not.

1. E → E + E
2. E → E - E
3. E → id

Solution:

From the above grammar String "id + id - id" can be derived in 2 ways:

First Leftmost derivation

1. E → E + E
2. → id + E
3. → id + E - E
4. → id + id - E

5. → id + id- id

Second Leftmost derivation

1. E → E - E
2. → E + E - E
3. → id + E - E
4. → id + id - E
5. → id + id - id

Since there are two leftmost derivation for a single string "id + id - id", the grammar G is ambiguous.

DISCUSSION WITH DIAGRAMS/TABLES

Ambiguity Elimination Compiler Design

Ambiguity Elimination

Ambiguity elimination makes the sentence clear and readable. A sentence is grammatically ambiguous if it can produce more than one parse tree for a particular grammar. In this article, we will learn how to remove ambiguity to make the grammar ambiguous.

Associativity

If an **operand** has an **operator** with the same precedence on both sides, the associativity of the operator will decide on which direction (side) the operand takes these operators. The operand will be taken by the left operator when the **operation** is left-associative and will be taken by the right operator when the operation is right-associative.

The operation such as multiplication, addition, subtraction, and division are left-associative.

Example

Suppose we have an expression as below:

```
id op id op id
```

Here, **id** is an operand, and **op** indicates the operator. This expression will be expressed as:

```
(id op id) op id
```

For example, (id + id) + id

If the operator is **exponentiation**, it is **right-associative**. It means if we evaluate the same expression, the order of evaluation will be:

```
id op (id op id)
```

For example, id ^ (id ^ id)

Precedence

If there is a situation arise when two different operators take a common operand, then the operand will assign to which operator will be decided by the precedence of operators. The 3 + 4 * 2 can have two distinct parse trees, one corresponds to (3 + 4) * 2 and another corresponds to 3 + (4 * 2). With the help of precedence among operators, this situation will be easily removed. We know that multiplication (*) has higher precedence over addition (+), so the expression 3 + 4 * 2 will be implemented as 3 + (4 * 2).

With the use of associativity of the operator and precedence of the operator, the ambiguity in grammar or its language can be eliminated.

Left Recursion

A grammar is said to be left recursive if it has any non–terminal, say A, and there is a derivation starting from that non-terminal such that A => Aa for some string a. We should know that the top-down parser cannot handle left–recursive grammar. To eliminate left–recursion from any given left-recursive string, we need to make changes in the given string.

Example

A => A? | ?

S => A? | ?

A => Sd

First is the example of immediate left recursion.

Second is the example of indirect left recursion.

Removal of left recursion

The production:

```
                  A => A? | ?
```

Can be transformed into the following production

```
                 A => ?A′
                 A′ => ?A′ | ?
```

without changing the strings derivable from A.

Left Factoring

The grammatical transformation is useful for the production of grammar. This transformation is suitable for predictive or top-down parser. If more than one grammar production has the same starting symbol in the string, the top-down parser cannot choose which of the production it should take to parse the string.

Example

Let us take the following grammar.

```
A => ?? | ?? |
```

In the given grammar, both the string have the same starting symbol. So we cannot immediately tell which production to choose to expand A. To eliminate this confusion, we use a technique called left factoring.

In this method, we combined the string with the same starting symbol into the single string, and the remaining derivation is added by new production.

So the above grammar can be written as:

```
A => ?A'
```

A'=> ? | ? |

Now there is only one production responsible for beginning from each starting symbol. This will be very useful for the parser to make a decision.

Limitations of Syntax Analyzers

The tokens from the lexical analyzer are the input for syntax analysis. A lexical analyzer checks the validity of the token. Syntax analyzer has the following drawback:

- The syntax analyzer cannot determine the validation of tokens.
- The syntax analyzer cannot detect that the token used in the lexical analyzer has been declared before being used or not.
- It cannot determine whether the token is initialized before using or not.

- Any operation performed on the token type is valid or not can be determined by syntax analysis.

CONCLUSION

Ambiguity is a type of meaning in which a phrase, statement or resolution is not explicitly defined, making several interpretations plausible. A common aspect of ambiguity is uncertainty. It is thus an attribute of any idea or statement whose intended meaning cannot be definitively resolved according to a rule or process with a finite number of steps. (The *ambi*- part of the term reflects an idea of "two", as in "two meanings".) The concept of ambiguity is generally contrasted with vagueness. In ambiguity, specific and distinct interpretations are permitted (although some may not be immediately obvious), whereas with information that is vague, it is difficult to form any interpretation at the desired level of specificity.

References

- *Willem J. M. Levelt (2008). An Introduction to the Theory of Formal Languages and Automata. John Benjamins Publishing. ISBN 978-90-272-3250-2.*
- • *Scott, Elizabeth (April 1, 2008). "SPPF-Style Parsing From Earley Recognizers". Electronic Notes in Theoretical Computer Science.* ***203*** *(2): 53–67. doi:10.1016/j.entcs.2008.03.044.*
- • Tomita, Masaru. "An efficient augmented-context-free parsing algorithm." Computational linguistics 13.1-2 (1987): 31-46.
- • *Hopcroft, John; Motwani, Rajeev; Ullman, Jeffrey (2001). Introduction to automata theory, languages, and computation (2nd ed.). Addison-Wesley. Theorem 9.20, pp. 405–406. ISBN 0-201-44124-1.*
- • *Axelsson, Roland; Heljanko, Keijo; Lange, Martin (2008). "Analyzing Context-Free Grammars Using an Incremental SAT Solver" (PDF). Proceedings of the 35th International Colloquium on Automata, Languages and*

Programming (ICALP'08), Reykjavik, Iceland. Lecture Notes in Computer Science. Vol. 5126. Springer-Verlag. pp. 410–422. doi:10.1007/978-3-540-70583-3_34. ISBN 978-3-540-70582-6.

- • *Knuth, D. E. (July 1965). "On the translation of languages from left to right". Information and Control.* ***8*** *(6): 607–639. doi:10.1016/S0019-9958(65)90426-2.*

- • *Hopcroft, John; Motwani, Rajeev; Ullman, Jeffrey (2001). Introduction to automata theory, languages, and computation (2nd ed.). Addison-Wesley. pp. 249–253. ISBN* 0-201-44124-

1. Which Parser is used for the implementation of recursive descent parsing? Draw the model diagram for that parser. Construct the parsing table for the grammar

E →E+T|T

*T →T*F|F*

F →(E)|id

Also construct the LR (0) Parsing table of the above grammar.

ASSIGNMENT

Answer

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation

A=>A α for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → A α | β it can be replaced with a sequence of two productions

A → β A' A' → α A' | ε

Without changing the set of strings derivable from A.
Consider the following grammar for arithmetic expressions:

E → E+T | T

T → T*F | F

F → (E) | id

This is not LR(0) Grammar.

First eliminate the left recursion for E as

E → TE'

E' → +TE' | ε

Then eliminate for T as

T → FT'

T'→ *FT' | ε

Thus, the obtained grammar after eliminating left recursion is

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

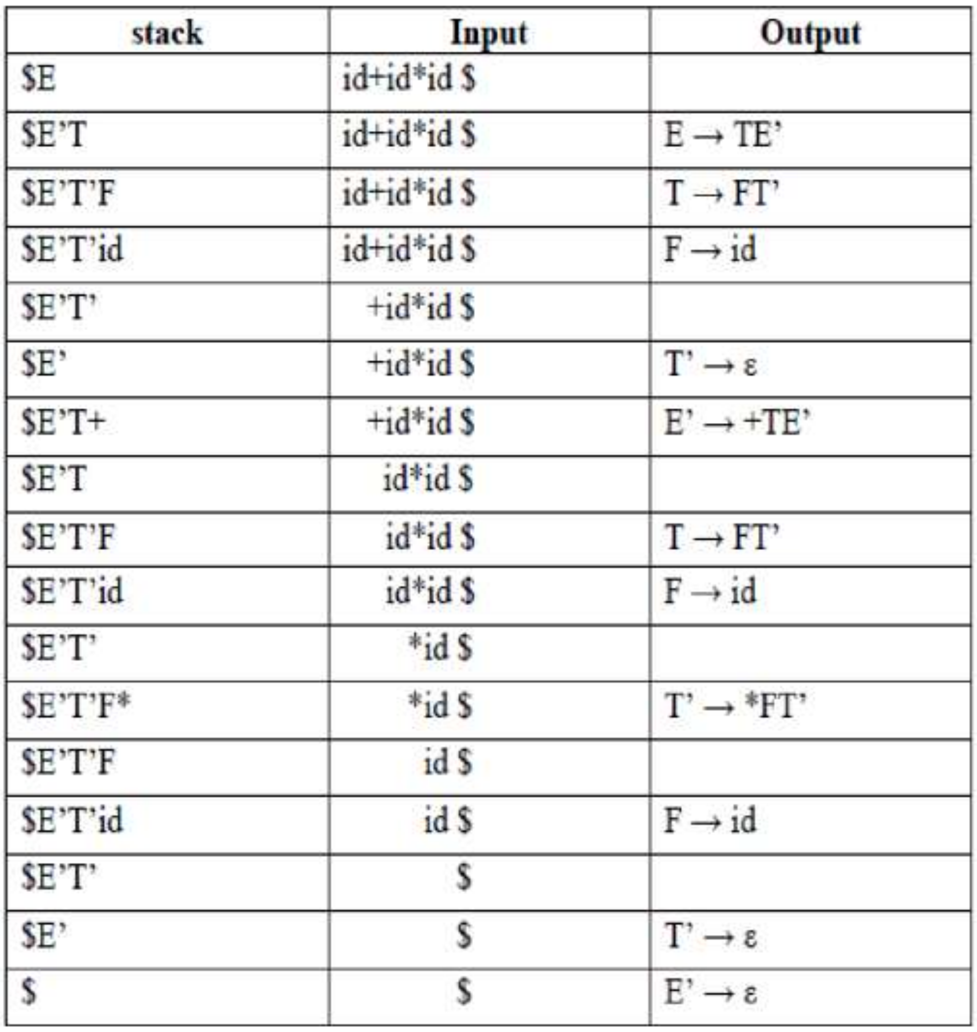
F → (E) | id

	FIRST	FOLLOW
E	{(, id}	{ $,) }
E'	{+, ε }	{ $,) }
T	{(, id}	{ +, $,) }
T'	{*, ε }	{ +, $,) }
F	{(, id}	{ *, +, $,) }

Answer CONT…

	INPUT SYMBOLS					
	+	*	(	)	id	$
E			E →TE’		E →TE’	
E’	E’→+TE’			E’→ ε		E’→ ε
T			T → FT’		T → FT’	
T’	T’ → ε	T’ → *FT’		T’ → ε		T’ → ε
F			F → (E)		F → id	

Stack implementation:



stack	Input	Output
$E	id+id*id $	
$E’T	id+id*id $	E → TE’
$E’T’F	id+id*id $	T → FT’
$E’T’id	id+id*id $	F → id
$E’T’	+id*id $	
$E’	+id*id $	T’ → ε
$E’T+	+id*id $	E’ → +TE’
$E’T	id*id $	
$E’T’F	id*id $	T → FT’
$E’T’id	id*id $	F → id
$E’T’	*id $	
$E’T’F*	*id $	T’ → *FT’
$E’T’F	id $	
$E’T’id	id $	F → id
$E’T’	$	
$E’	$	T’ → ε
$	$	E’ → ε

2. *$S \rightarrow iEtSS'|a$*

$S' \rightarrow Es|\varepsilon$

$E \rightarrow b$

Is this grammar LL (1). Describe it.

	FIRST	FOLLOW
S	{i ,a}	[$,b}
S’	{b, ε }	{$,b}
E	{b}	{t}

Parsing Table:

Non-terminal	i	t	a	b	s	$
S	*$S \rightarrow iEtSS'$*		S->a			
S’				S’-> ε S’->Es		S’-> ε
E				E->b		

M[S’ ,E]=2

So this grammar is not an LL(1) grammar. BeauseM[S’, E] has two entries.