CD: COMPILER DESIGN
CD: UNIT-2 09/2022

SEPTEMBER 2022 / IT-3rd year, Vth semester
FALL SEMESTER, YEAR (Vth, 3rd)
FALL SESSION (2022-23)
(CD)
MS. SHWETA TIWARI
Published: SEPTEMBER, 2022

PREPARED FOR
Engineering Students
All Engineering College

Shweta Tiwari

INSTRUCTOR: Ms. SHWETA TIWARI
shwetatiwari08@recabn.ac.in
shwetatiwari08aug@gmail.com

CD: COMPILER DESIGN

TOPIC On : Removing Ambiguity Ambiguous to Unambiguous

By SHWETA TIWARI

Under On: Unit-2

TOPIC On : Removing Ambiguity Ambiguous to Unambiguous

Grammar Ambiguity

We will discuss how to convert ambiguous grammar to unambiguous grammar.

Converting Ambiguous Grammar To Unambiguous Grammar-

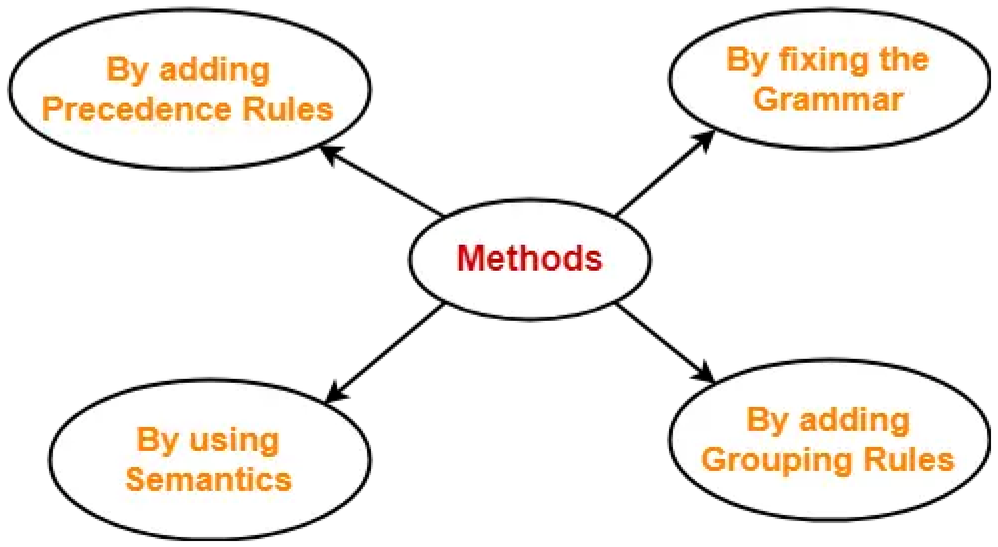
- Causes such as left recursion, common prefixes etc makes the grammar ambiguous.
- The removal of these causes may convert the grammar into unambiguous grammar.
- However, it is not always compulsory.

NOTE

It is not always possible to convert an ambiguous grammar into an unambiguous grammar.

Methods To Remove Ambiguity-

The ambiguity from the grammar may be removed using the following methods-



- By fixing the grammar
- By adding grouping rules
- By using semantics and choosing the parse that makes the most sense
- By adding the precedence rules or other context sensitive parsing rules

Removing Ambiguity By Precedence & Associativity Rules-

An ambiguous grammar may be converted into an unambiguous grammar by implementing-

- Precedence Constraints
- Associativity Constraints

These constraints are implemented using the following rules-

Rule-01:

The precedence constraint is implemented using the following rules-

- The level at which the production is present defines the priority of the operator contained in it.

- The higher the level of the production, the lower the priority of the operator.
- The lower the level of the production, the higher the priority of the operator.

Rule-02:

The associativity constraint is implemented using the following rules-

- If the operator is left associative, induce left recursion in its production.
- If the operator is right associative, induce right recursion in its production.

PROBLEMS BASED ON CONVERSION INTO UNAMBIGUOUS GRAMMAR-

Problem-01:

Convert the following ambiguous grammar to unambiguous grammar-

R → R + R / R . R / R* / a / b

where * is kleene closure and . is concatenation.

Solution-

To convert the given grammar into its corresponding unambiguous grammar, we implement the precedence and associativity constraints.

We have-

- Given grammar consists of the following operators-

+ , . , *

- Given grammar consists of the following operands-

a , b

The priority order is-

(**a** , **b**) > * > **.** > +

where-

- **.** operator is left associative
- + operator is left associative

Using the precedence and associativity rules, we write the corresponding unambiguous grammar as-

E → E + T / T

T → T . F / F

F → F* / G

G → a / b

Unambiguous Grammar

OR

E → E + T / T

T → T . F / F

F → F* / a / b

Unambiguous Grammar

Problem-02:

Convert the following ambiguous grammar to unambiguous grammar-

bexp → bexp or bexp / bexp and bexp / not bexp / T / F

where bexp represents Boolean expression, T represents True and F represents False.

<u>Solution-</u>

To convert the given grammar into its corresponding unambiguous grammar, we implement the precedence and associativity constraints.

We have-

- Given grammar consists of the following operators-

or , **and** , **not**

- Given grammar consists of the following operands-

T , **F**

The priority order is-

(**T** , **F**) > **not** > **and** > **or**

where-

- **and** operator is left associative
- **or** operator is left associative

Using the precedence and associativity rules, we write the corresponding unambiguous grammar as-

bexp → bexp or M / M

M → M and N / N

N → not N / G

G → T / F

Unambiguous Grammar

OR

bexp → bexp or M / M

M → M and N / N

N → not N / T / F

Unambiguous Grammar

Example 1 – Consider the ambiguous grammar

E -> E-E | id

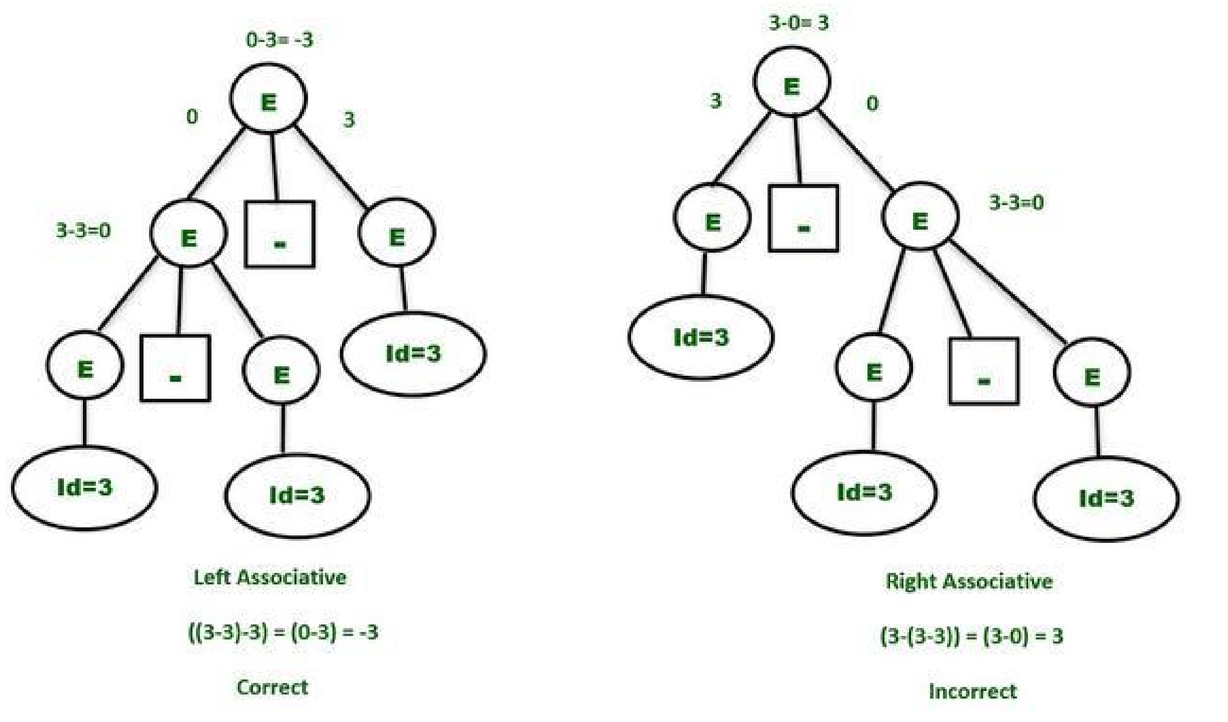
The language in the grammar will contain { id, id-id, id-id-id, ….}

Say, we want to derive the string **id-id-id.** Let's consider a single value of id=3 to get more insights. The result should be :

3-3-3 =-3

Since the same priority operators, we need to consider associativity which is left to right.

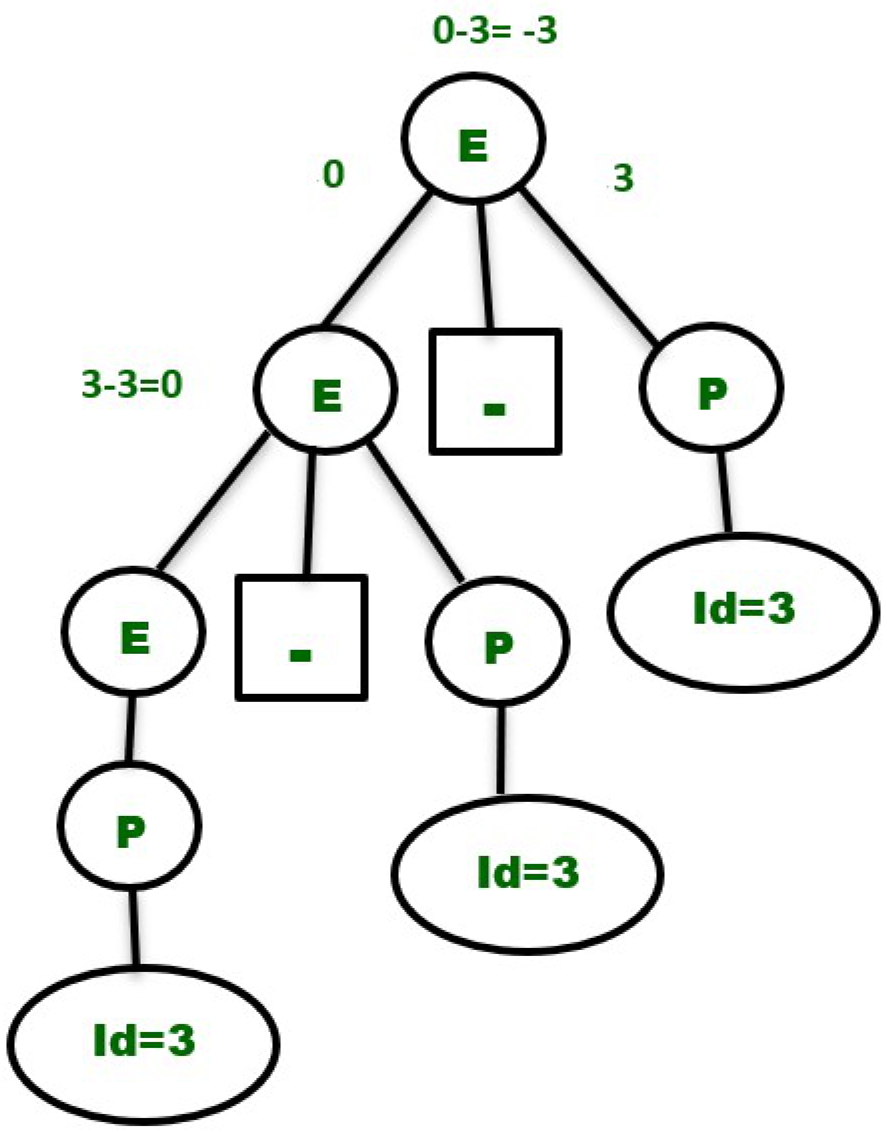
Parse Tree – The parse tree which grows on the left side of the root will be the correct parse tree in order to make the grammar unambiguous.



So, to make the above grammar unambiguous, simply make the grammar **Left Recursive** by replacing the left most non-terminal E in the right side of the production with another random variable, say **P.** The grammar becomes :

E -> E – P | P
P -> id

The above grammar is now unambiguous and will contain only one Parse Tree for the above expression as shown below –



Similarly, the unambiguous grammar for the expression : **2^3^2** will be –

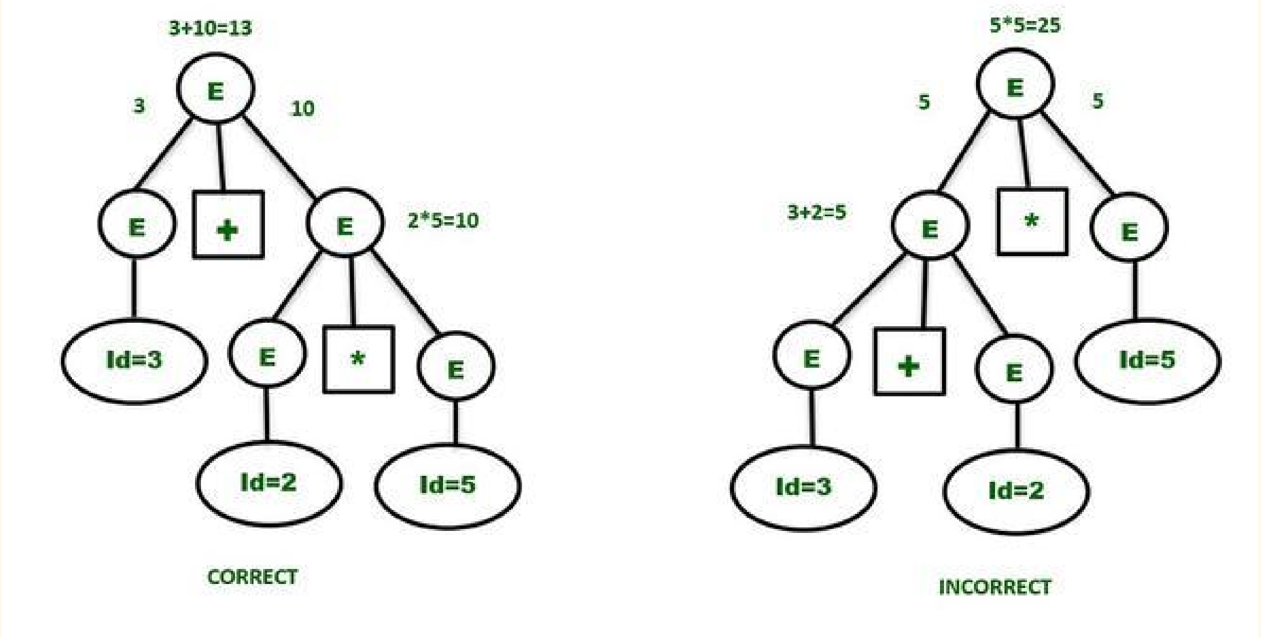
E -> P ^ E | P   // Right Recursive as ^ is right associative.
P -> id

Example 2 – Consider the grammar shown below, which has two different operators :

E -> E + E | E * E | id

Clearly, the above grammar is ambiguous as we can draw two parse trees for the string **"id+id*id"** as shown below. Consider the expression :

3 + 2 * 5   // "*" has more priority than "+"
The correct answer is : (3+(2*5))=13



The "+" having the least priority has to be at the upper level and has to wait for the result produced by the "*" operator which is at the lower level. So, the first parse tree is the correct one and gives the same result as expected.

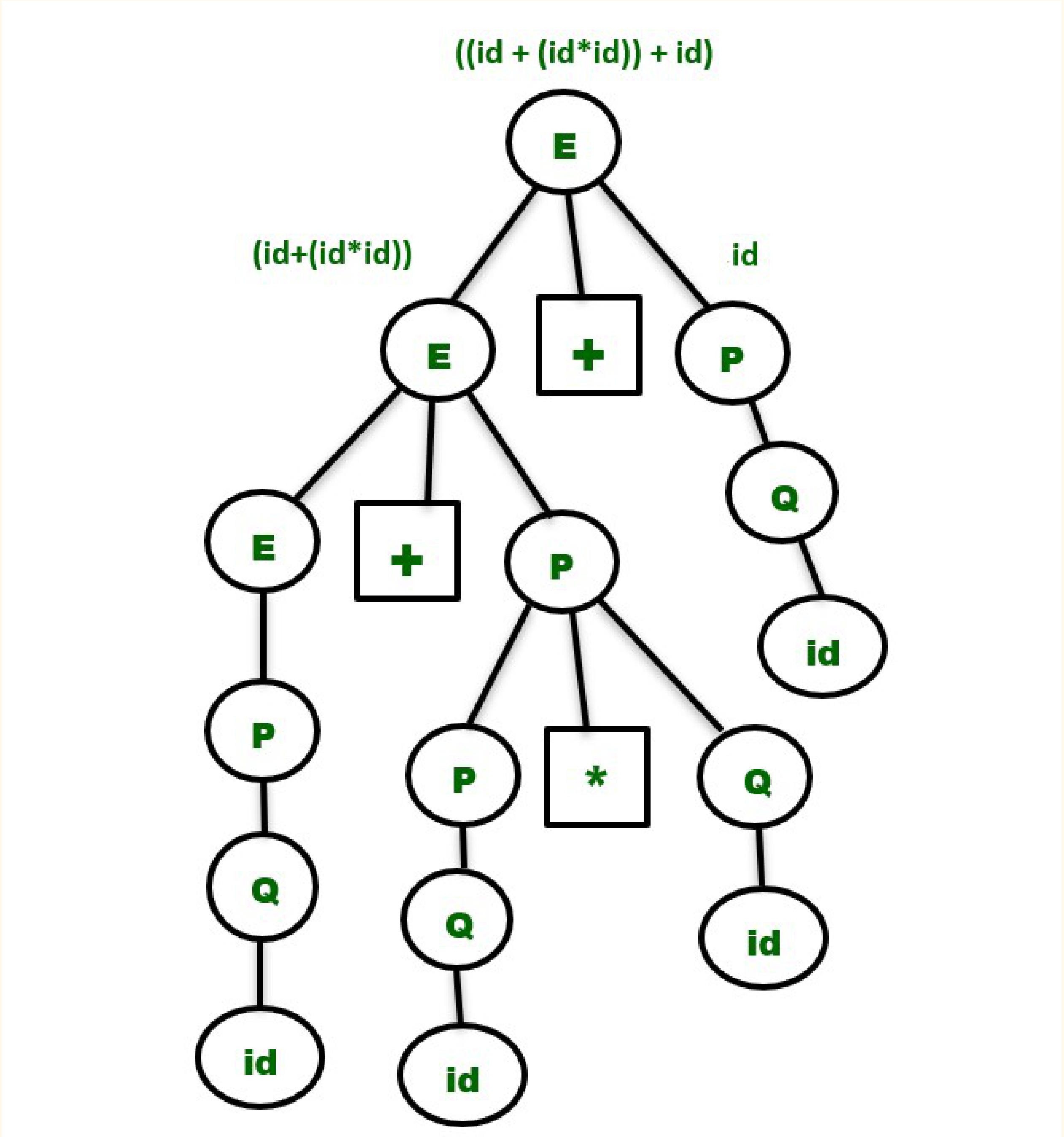
The unambiguous grammar will contain the productions having the highest priority operator **("*" in the example)** at the lower level and vice versa. The associativity of both the operators are **Left to Right**. So, the unambiguous grammar has to be **left recursive.** The grammar will be :

E -> E + P *// + is at higher level and left associative*
E -> P
P -> P * Q *// * is at lower level and left associative*
P -> Q
Q -> id

(or)

E -> E + P | P
P -> P * Q | Q
Q -> id

E is used for doing addition operations and **P** is used to perform multiplication operations. They are independent and will maintain the precedence order in the parse tree.
The parse tree for the string **” id+id*id+id ”** will be –



Note : It is very important to note that while converting an ambiguous grammar to an unambiguous grammar, we shouldn't change the original language provided by the ambiguous grammar. So, the non-terminals in the ambiguous grammar have to be replaced with other variables in such a way that we get the same language as it was derived before and also maintain the precedence and associativity rule simultaneously.

This is the reason we wrote the production **E -> P** and **P -> Q** and **Q -> id** after replacing them in the above example, because the language contains the strings **{ id, id+id }** as well.
Similarly, the unambiguous grammar for an expression having the operators **-,*,^** is :

E -> E – P | P *// Minus operator is at higher level due to least priority and left associative.*
P -> P * Q | Q *// Multiplication operator has more priority than – and lesser than ^ and left* ***associative.***
Q -> R ^ Q | R *// Exponent operator is at lower level due to highest priority and right associative.*
R -> id

Also, there are some ambiguous grammars which can't be converted into unambiguous grammars.