Syntax Analyzer

Dr. Meera Thapar Khanna

CSE Department

Pandit Deendayal Energy University, Gandhinagar

Phase 2: Syntax Analysis

- Syntax analysis discovers structure in code.
- It determines whether or not a text follows the expected format i.e. to make sure that the source is correct or not.
- Syntax analysis is based on the rules based on the specific programing language by constructing the parse tree with the help of tokens.

Following tasks are performed in this phase:

- Obtain tokens from the lexical analyzer
- Checks if the expression is syntactically correct or not
- Report all syntax errors
- Construct a hierarchical structure which is known as a parse tree

Grammar

It is a finite set of formal rules for generating syntactically correct sentences or meaningful correct sentences.

Grammar is basically composed of two basic elements –

✓**Terminal Symbols :**

Terminal symbols are those which are the components of the sentences generated using a grammar and are represented using small case letter like a, b, c etc.

✓**Non-Terminal Symbols :**

Non-Terminal Symbols are those symbols which take part in the generation of the sentence but are not the component of the sentence. Non-Terminal Symbols are also called Auxiliary Symbols and Variables. These symbols are represented using a capital letter like A, B, C, etc.

Grammar

Grammar can be represented by 4 tuples –

<N, T, P, S>

N – Finite Non-Empty Set of Non-Terminal Symbols.

T – Finite Set of Terminal Symbols.

P – Finite Non-Empty Set of Production Rules.

S – Start Symbol (Symbol from where we start producing strings).

Production Rules

A production or production rule is a rewrite rule specifying a symbol substitution that can be recursively performed to generate new symbol sequences.

It is of the form $\alpha \rightarrow \beta$ where α is a Non-Terminal Symbol which can be replaced by β which is a string of Terminal Symbols or Non-Terminal Symbols.

Example

- Expression-> expression + term
- expression-> expression – term
- expression-> term
- term-> term * factor
- term-> factor
- factor-> (expression)
- factor-> id

The Non terminal symbols are expression, term, factor and Expression is the starting symbol.

Conventions Used for Writing Grammars

Terminals:

(a) Lowercase letters early in the alphabet, such as a, b, e.

(b) Operator symbols such as +, *, and so on.

(c) Punctuation symbols such as parentheses, comma, and so on.

(d) The digits 0, 1. . . 9.

(e) Boldface strings such as id or if, each of which represents a single terminal symbol.

Non terminals:

(a) Uppercase letters early in the alphabet, such as A, B, C.

(b) The letter S, which, when it appears, is usually the start symbol.

(c) Lowercase, italic names such as expr or stmt.

For example, non terminal for expressions, terms, and factors are often represented by E, T, and F, respectively. The grammar for the arithmetic expressions:

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid id$$

Example

Grammar G1 = <N, T, P, S>

- T = {a,b}   (terminal symbols)
- P = {A->Aa, A->Ab, A->a, A->b, A-> ε}   (production rules)
- S = {A}   (Start Symbol)

With the start symbol S, we can produce Aa, Ab, a, b, ε which can further produce strings by replacing A by the Strings mentioned in the production rules.

So this grammar can be used to produce strings like (a+b)*.

Derivation

String 1:

A->a #using production rule 3

String 2:

A->Aa #using production rule 1

Aa->ba #using production rule 4

String 3:

A->Aa #using production rule 1

Aa->Aba #using production rule 2

Aba->ba #using production rule 5

Example

Grammar G2 = <N, T, P, S>

- N = {A} non-terminals Symbols
- T = {a} terminal symbols
- P = {A->Aa, A->AAa, A->a, A->ε} production rules
- S = {A} Start Symbol

With the start symbol is S, we can produce Aa, AAa, a, ε which can further produce strings where A can be replaced by the Strings mentioned in the production rules.

So this grammar can be used to produce strings of form (a)*.

Derivation

- A derivation is basically a sequence of production rules, in order to get the input string.
- During parsing, 2 decisions to be taken:
  - Deciding the non-terminal which is to be replaced.
  - Deciding the production rule, by which, the non-terminal will be replaced.
- To decide which non-terminal to be replaced with production rule:
  - **Left-most Derivation:** If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.
  - **Right-most Derivation:** If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Left Most Derivation

Grammar

E-> E+E | E*E | -E| (E) | id

Input is: id+id*id

Order of Productions (LMD):

E-> E + E

-> id + E

-> id + E * E

-> id + id * E

-> id + id * id

Right Most Derivation

Grammar

E-> E+E | E*E | -E| (E) | id

Input is: id+id*id

Order of Productions (RMD):

E-> E + <u>E</u>

-> E + E * <u>E</u>

-> E + <u>E</u> * id

-> <u>E</u> + id * id

-> id + id * id

Parse Tree

- A parse tree is a graphical representation of a derivation that specifies the order in which productions are applied to replace non terminals.
- Each interior node of a parse tree represents the application of a production.
- All the interior nodes are Non terminals.
- All the leaf nodes are terminals.
- All the leaf nodes reading from the left to right will be the output of the parse tree.
- If a node n is labeled X and has children $n_1, n_2, n_3, \dots n_k$ with labels $X_1, X_2, \dots X_k$ respectively, then there must be a production $A \rightarrow X_1 X_2 \dots X_k$ in the grammar.

Parse Tree Example

Grammar

E-> E+E | E*E | -E| (E) | id

Input is: –(id+id)

Order of Productions:

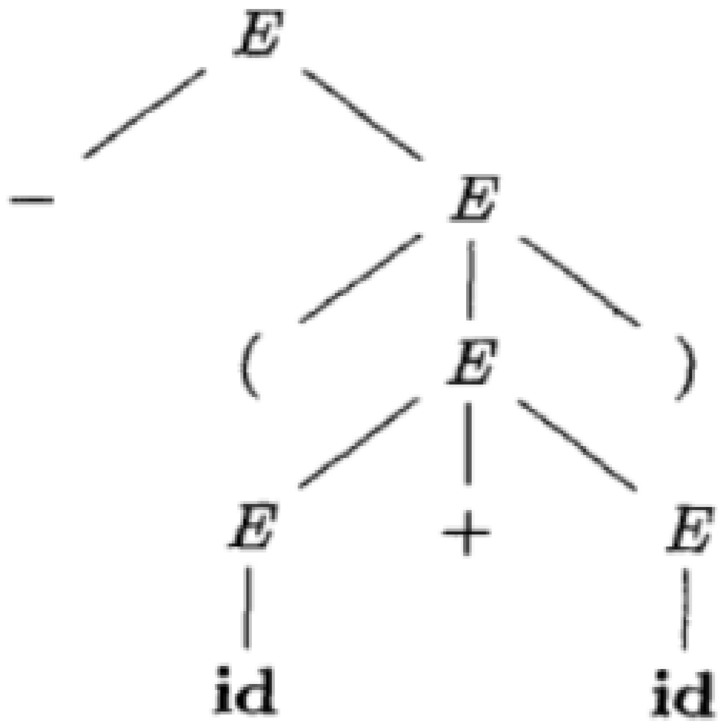
E-> -E

E-> (E)

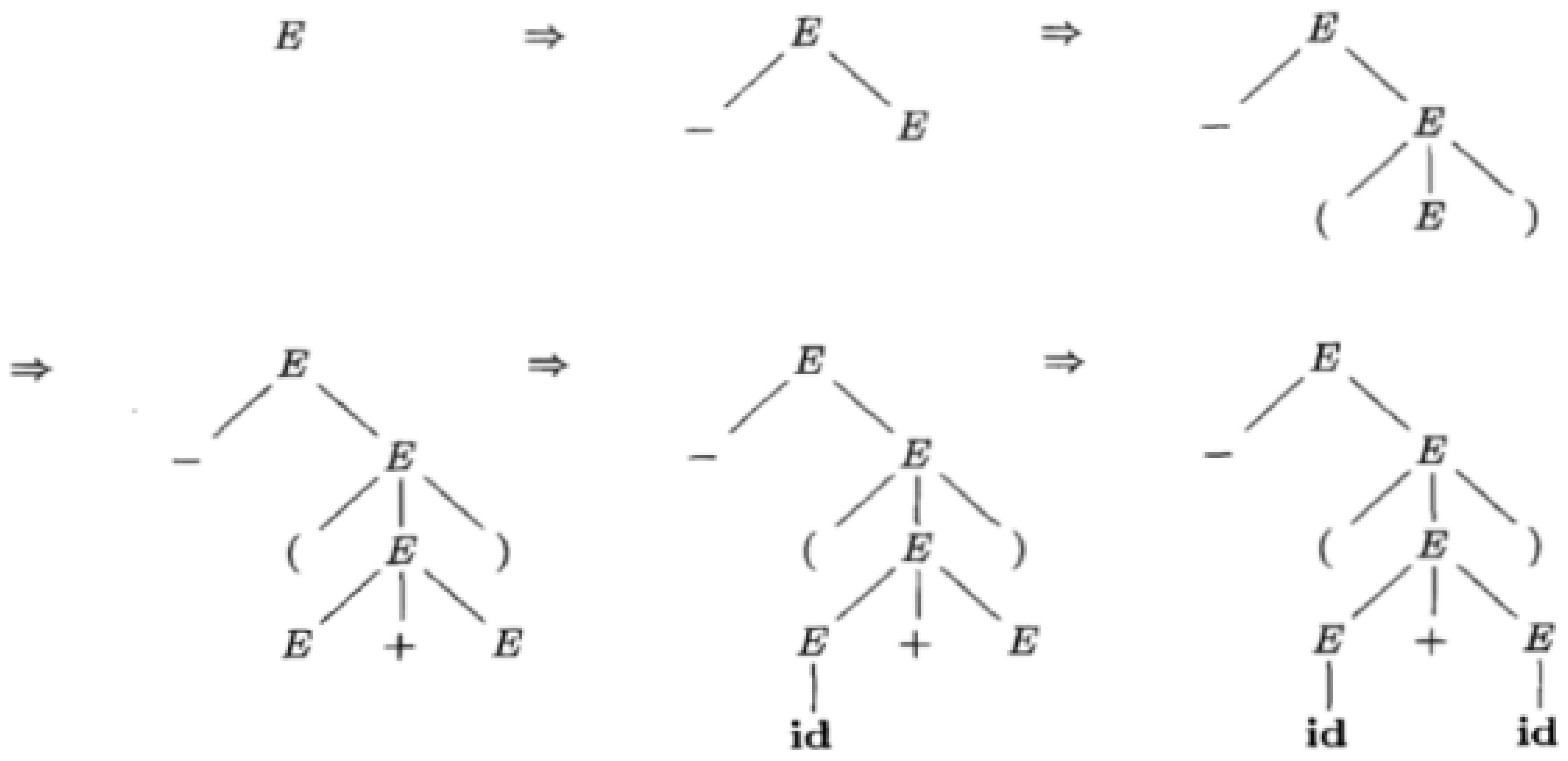
E-> E+E

E-> id

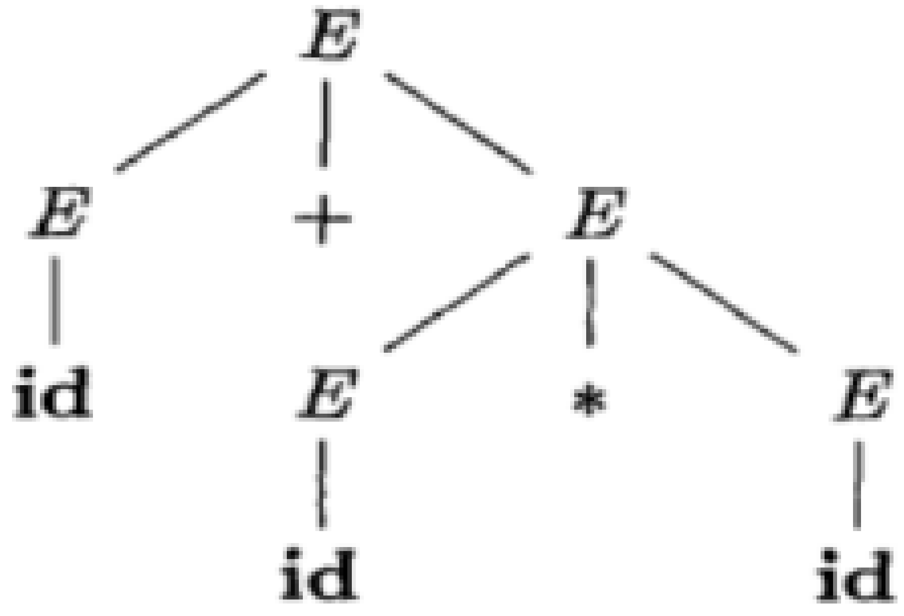
E-> id



Step by Step Parse Tree



Generate Parse Tree for Input -> id+id*id



Two Left Most Derivations (id+id*id)

E -> E + E

-> id + E

-> id + E * E

-> id + id * E

-> id + id * id

E -> E * E

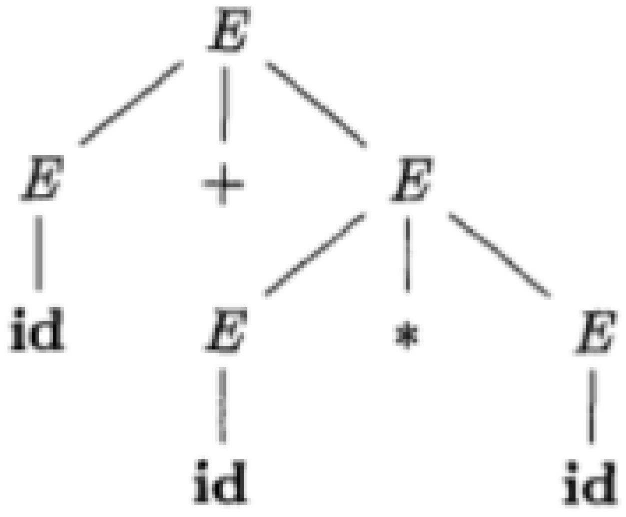
-> E + E * E

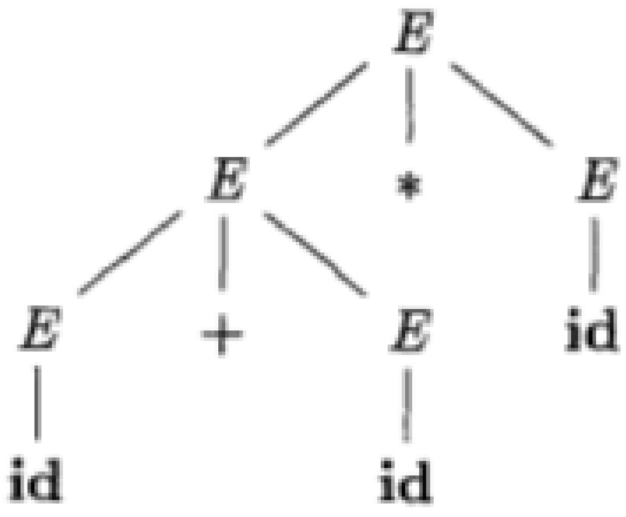
-> id + E * E

-> id + id * E

-> id + id * id

Parse Tree (LMD) for input (id+id*id)





Ambiguity

- A grammar G is said to be ambiguous if it has more than one parse tree for any of the left or right derivation and for at least one string.
- Example:

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

V = {E}

T = {+, *, id}

S = {E}

Generate string (id+id*id) using above production rules.

Example

Grammar:

stmt -> if *expr* then *stmt*

stmt -> if *expr* then *stmt* else *stmt*

stmt -> other

string if E1 then if E2 then S1 else S2

Ambiguous Grammar

Associativity

If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators.

If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

Example

Operations such as Addition, Multiplication, Subtraction, and Division are left associative. If the expression contains:

Operand operator operand operator operand

it will be evaluated as:

(Operand operator operand) operator operand

For example:

(id + id) + id

Operations like Exponentiation are right associative, i.e., the order of evaluation in the same expression will be:

Operand operator (operand operator operand)

For example:

id ^ (id ^ id)

Precedence

If two different operators share a common operand, the precedence of operators decides which will take the operand.

For instance, expression 2+3*4 can have two different parse trees,

(2+3)*4 and

2+(3*4).

By setting precedence among operators, this problem can be easily removed. Mathematically * (multiplication) has precedence over + (addition), so the expression 2+3*4 will always be interpreted as:

2 + (3 * 4)

Removal of Ambiguity

Ambiguity can be removed on the basis of the following two properties –

Precedence –

If different operators are used, we will consider the precedence of the operators. The three important characteristics are :

- The level at which the production is present denotes the priority of the operator used.
- The production at higher levels will have operators with less priority. In the parse tree, the nodes which are at top levels or close to the root node will contain the lower priority operators.
- The production at lower levels will have operators with higher priority. In the parse tree, the nodes which are at lower levels or close to the leaf nodes will contain the higher priority operators.

Associativity –

If the same precedence operators are in production, then we will have to consider the associativity.

- If the associativity is left to right, then we have to prompt a left recursion in the production. The parse tree will also be left recursive and grow on the left side.
- +, -, *, / are left associative operators.
- If the associativity is right to left, then we have to prompt the right recursion in the productions. The parse tree will also be right recursive and grow on the right side.
- ^ is a right associative operator.

Example

Grammar

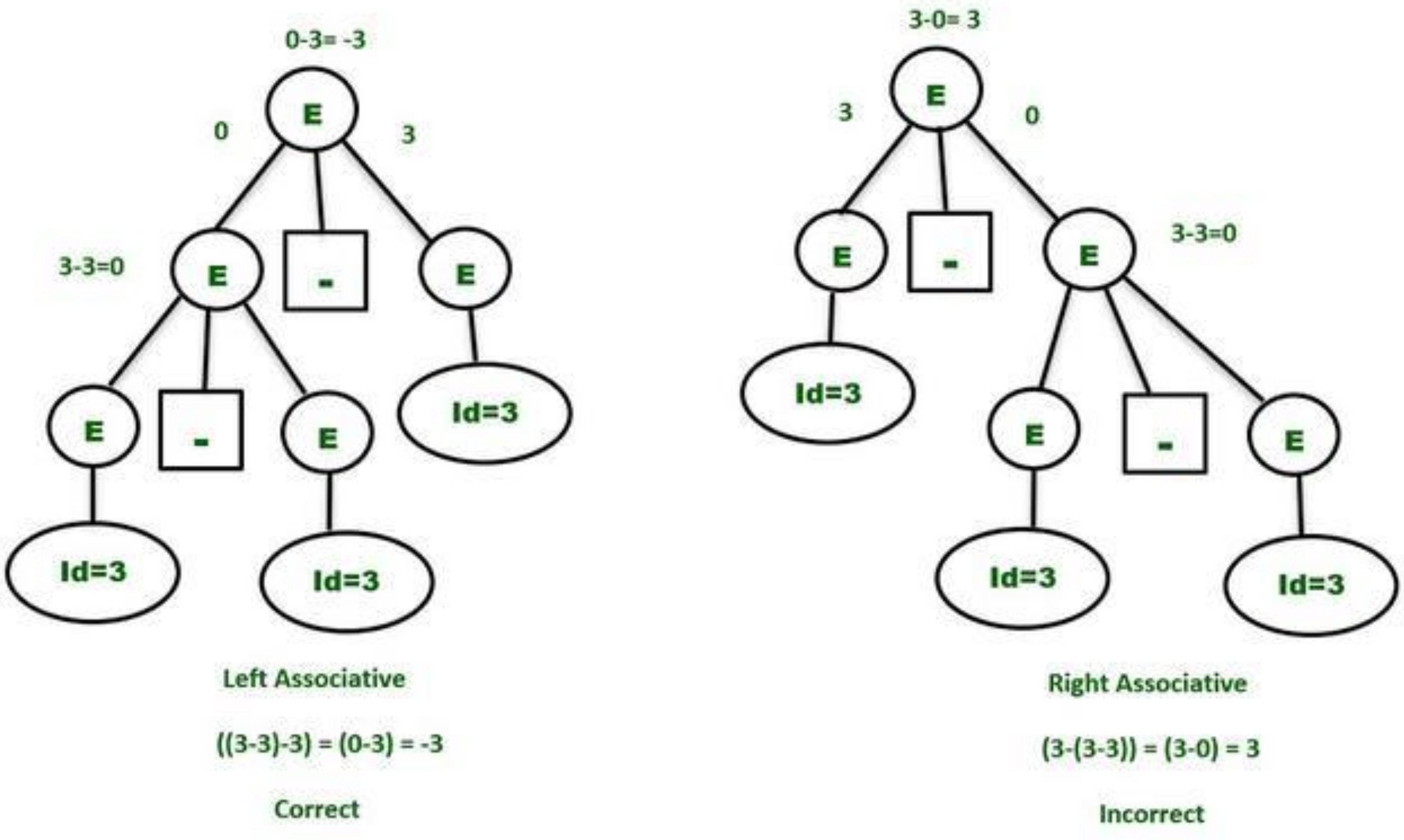
E -> E-E | id

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

For instance, we want to derive the string id-id-id. Let's take value of id=3. The result should be 3-3-3 =-3

Here the same priority operators are in the production, so consider associativity which is left to right.

Parse Tree


0-3= -3
0
3
E
3-3=0
E
-
E
E
-
E
Id=3
Id=3
Id=3
Left Associative
((3-3)-3) = (0-3) = -3
Correct
3-0= 3
3
0
E
E
-
E
3-3=0
Id=3
E
-
E
Id=3
Id=3
Right Associative
(3-(3-3)) = (3-0) = 3
Incorrect

Ambiguous grammar

$A \rightarrow A\alpha \mid \beta$

To make the above grammar unambiguous, simply make the grammar Left Recursive by replacing the left most non-terminal A in the right side of the production with another random variable, say A'.

Unambiguous grammar

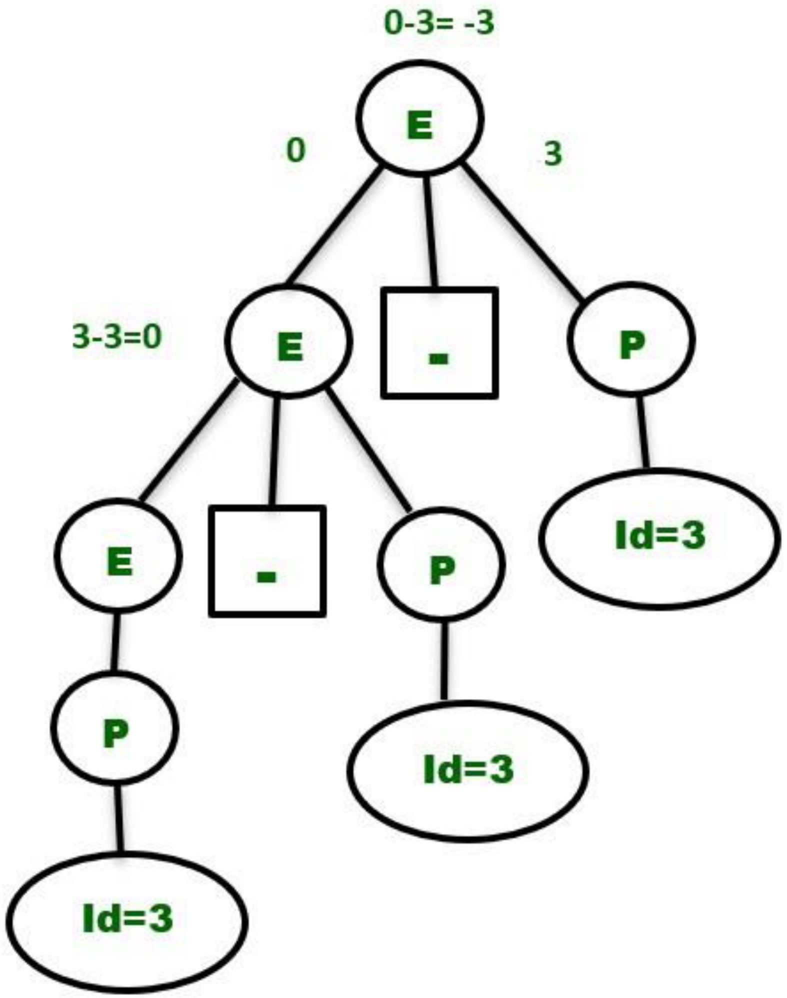
$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

- E -> E – P | P
- P -> id

unambiguous grammar

having only one Parse Tree



unambiguous grammar for the expression : 2^3^2

E -> E^E|id

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

Example

Grammar

E-> E+E | E*E | -E| (E) | id

Unambiguous Grammar

E -> E + T | T

T -> T * F | F

F -> -E | (E) | id

Left Recursion

- Another feature of the CFGs which is not desirable to be used in top down parsers is left recursion.
- A grammar is left recursive if it has a non terminal A such that there is a derivation A=>Aα for some string α in (TUV)*.
- LL(1) or Top Down Parsers can not handle the Left Recursive grammars, so we need to remove the left recursion from the grammars before being used in Top Down Parsing.
- A grammar becomes left-recursive if it has any non-terminal ‘A’ whose derivation contains ‘A’ itself as the left-most symbol.
- Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

(1) $A \Rightarrow A\alpha \mid \beta$

an example of immediate left recursion, where A is any non-terminal symbol and α represents a string of non-terminals.

(2) $S \Rightarrow A\alpha \mid \beta$

$A \Rightarrow Sd$

an example of indirect-left recursion. A top-down parser will first parse the A, which in-turn will yield a string consisting of A itself and the parser may go into a loop forever.

Removal of Left Recursion

The production

- $A \Rightarrow A\alpha \mid \beta$

is converted into following productions

- $A \Rightarrow \beta A'$
- $A' \Rightarrow \alpha A' \mid \varepsilon$

no impact on the strings derived from the grammar, but removes immediate left recursion.

Unambiguous Grammar

E -> E + T | T

T -> T * F | F

F -> -E | (E) | id

First eliminate the left recursion for E as

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

Then eliminate for T as

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

New Grammar

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Example

Grammar

S -> Aa | b | ε

A -> Ac | Sd

If left recursion is two or more level deep or indirect recursion like:

S -> Aa -> Sda

Algorithm to eliminate left recursion

1. Arrange the non-terminals in some order A1, A2 . . . An.
2. for i := 1 to n do begin

    for j := 1 to i-1 do begin

        replace each production of the form Ai → Aj γ

        by the productions Ai → δ1 γ | δ2γ | . . . | δk γ

        where Aj→δ1|δ2| . . .|δk are all the current Aj-productions;

    end

    eliminate the immediate left recursion among the Ai-productions

end

$A1 \rightarrow A2a \mid b \mid \varepsilon$

$A2 \rightarrow A2c \mid A1d$

i=1

For A1 no left recursion

i=2

For j=1 to 1 do

A2 -> A1 γ and replace with A2 -> δ1 γ | δ2γ | . . . | δk γ where A1 -> δ1|δ2| . . .|δk are A1 productions

here, A2 -> A1d becomes A2 -> A2ad|bd|d

Now we have

A1 -> A2a | b | ε

A2 -> A2c|A2ad|bd|d

A2-> bdA’|dA’

A’ -> cA’|adA’| ε

- A1 -> A2a | b | ε

- A2-> bdA’|dA’
- A’ -> cA’|adA’| ε

Example

- $S \to a|\wedge|(T)$

- $T \to T, S|S$

Example

A-> Aα

A-> Aα|β|….|γ

New Grammar

A -> NA’

N -> β | … | γ

A’ -> αA’| ε

The production set

- $S \Rightarrow A\alpha \mid \beta$
- $A \Rightarrow Sd$

after converting productions become

- $S \Rightarrow A\alpha \mid \beta$
- $A \Rightarrow A\alpha d \mid \beta d$

and then, remove immediate left recursion using the first technique.

- $A \Rightarrow \beta dA'$
- $A' \Rightarrow \alpha dA' \mid \varepsilon$

Now none of the production has either direct or indirect left recursion.

- **Left factoring** is useful for producing a grammar suitable for predictive or top-down parsing.
- A grammar in which more than one production has common prefix is to be rewritten by factoring out the prefixes.
- For example, in the following grammar there are n A productions have the common prefix α, which should be removed or factored out without changing the language defined for A
- A -> αA1 | αA2 | αA3 |αA4 |… | αAn
- Then it cannot determine which production to follow to parse the string as both productions are starting from the same terminal (or non-terminal). To remove this confusion, we use a technique called left factoring.
- Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.
- A -> αA'
- A' -> A1 | A2 | A3 |A4 |… | An

Example

Grammar :

S -> iEtS | iEtSeS | a

E → b

New Grammar:

S -> iEtSS’ | a

S’ -> eS | ε

E -> b