\

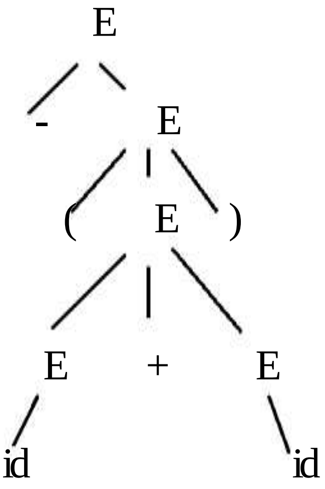
## Syntactic Analyzer ( Parser )

Every programming language has rules that prescribe the syntactic structure of well formed programs. The syntax of programming language constructs can be described by context free grammars. In our compiler model, the parser obtains a string of tokens from the lexical analyzer, and verifies that the string can be generated by the grammar for the source program. We expect the parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.

In syntax analysis we are concerned with groping *tokens* into larger syntactic classes such as *expression* , *statements* , and *procedure*. The syntax analyzer (parser) outputs a *syntax tree*, in which its leaves are the *tokens* and every non-
leaf node represents a syntactic *class* type. For example:-
Consider the following grammars:-

$$E \longrightarrow E+E \mid E*E \mid (E) \; - \mid E \; id \mid$$

Then the parse tree for **-(id+id)** is:-



## Syntax Error Handling :-

Often much of the error detection and recovery in a compiler is central around the *parser*. One reason of this is that many errors are syntactic in nature. Errors where the token stream violates the structure of the language are determined by parser, such as an arithmetic expression with unbalanced parentheses.

## Derivations :-

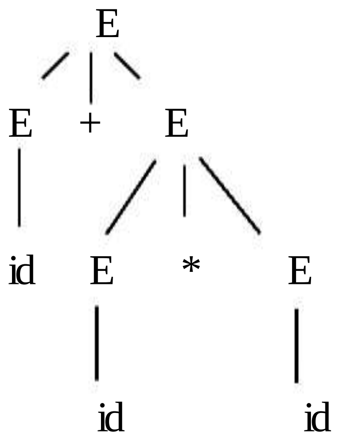
This derivational of view gives a precise description of the *top-down* construction of *parse tree*. The central idea here is that a *production* is treated as rewriting rule in which the *nonterminal* on the left is replaced by the string on the right side of the *production*. For example, consider the following grammar:

E → E+E
E → E*E
E → (E)
E → -E
E → id

The *derivation* of the input string **id + id* id** is:

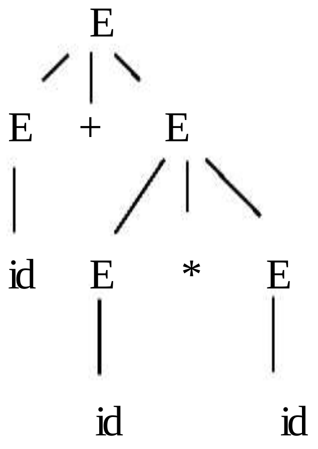
| ***Left-most derivation*** | ***Right-most derivation*** |
|---|---|
| $\underline{E}$ | $\underline{E}$ |
| $\underline{E}+E$ | $E+\underline{E}$ |
| $id+\underline{E}$ | $E+E*\underline{E}$ |
| $id+\underline{E}*E$ | $E+\underline{E}*id$ |
| $id+id*\underline{E}$ | $\underline{E+id*id}$ |
| id+id*id | id+id*id |

Note:- *parse tree* may by viewed as a graphical representation for a derivation :

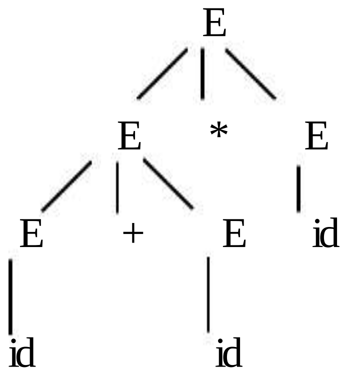


**Ambiguity :-**

A grammar that produce more that one parse tree for same sentence is said to be ***Ambiguous***. In the another way, by produced more that one *left-most derivation* or more that one *Right-most derivation* for the same sentence.



**(1)**



**(2)**

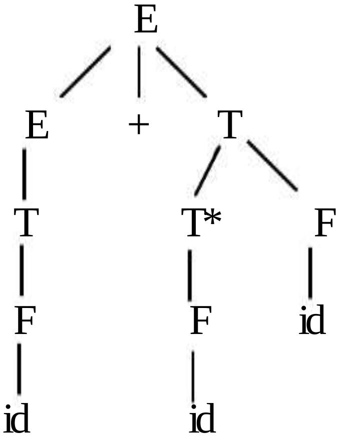
two parse tree for **id+id*id**

| | |
|---|---|
| E | E |
| E+E | E*E |
| id+E | E+E*E |
| id+E*E | id+E*E |
| id*id*E | id+id*E |
| id+id*id | id+id*id |
| **(1)** | **(2)** |

Two *left-most derivation*s for **id+id*id**

Sometimes am ambiguous grammar can be rewritten to eliminate the ambiguity. Such as:

$$\begin{array}{lcl} E \rightarrow E+E & & E \rightarrow E+T \mid T \\ E \rightarrow E*E & & E \rightarrow (E) \\ E \rightarrow (E) & \Rightarrow & E \rightarrow -E \\ E \rightarrow -E & & T \rightarrow T*F \mid F \\ E \rightarrow id & & F \rightarrow id \end{array}$$



parse tree for **id+id*id**

**Left-Recursion :-**

A grammar is left-recursion if has a *nonterminal* **A** such that there is a derivation A ⟶ Aα for some string α . Top-down parser cannot handle *left-recursion* grammars, so a transformation that eliminates left-recursion is needed:

$$A \rightarrow A\alpha \mid \text{ß}$$

$$A \rightarrow \text{ß}\, A'$$

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

**OR:**

$A \rightarrow A\alpha1 \mid A\alpha2 \mid \ldots A\alpha m \,.\mid.\, ß1 \mid ß2 \mid \ldots \mid \mid ßn$

$A \rightarrow ß1A' \mid ß2\, A' \,.\mid..\, ßn \mid A'$

$A' \rightarrow \alpha1\, A' \mid \alpha2\, A' \ldots \mid \alpha m \mid A' \mid \varepsilon\ .$

**Example:**

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

---

$E \rightarrow T\, E'$

$E' \rightarrow +T\, E' \mid \in$

$T \rightarrow FT'$

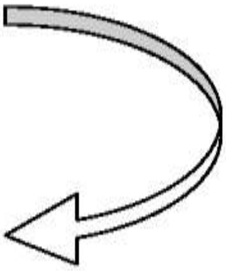
$T' \rightarrow *F\, T' \mid \in$

$F \rightarrow (E) \mid id$

**Left-Factoring :-**

The basic idea is that when is not clear which of two alternative production to use to expand a *nonterminal* A . We may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

$A \rightarrow \alpha\, ß1 \mid \alpha\, ß2$ where $\alpha \neq \in$



**$A \rightarrow \alpha\, A'$**

**$A' \rightarrow ß1 \mid ß2$**

**OR :-**

**A→ α ß1 | α ß2 .| α ßn | γ** **where α ≠ ∈**

**A→ α A' | γ**

**A'→ ß1 | ß2 .. | | ßn**

**Example:**

S→ iEtS | iEtSeS a |
E→b

S → iEtSS' a |

S' → eS | ∈

E→ b

***Easy Come, Easy Go***