Syntax Analyzer

When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation.

Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a Parse Tree or Syntax Tree.

The parse tree is constructed by using the pre-defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. If not, error is reported by syntax analyzer.

Example (1):-

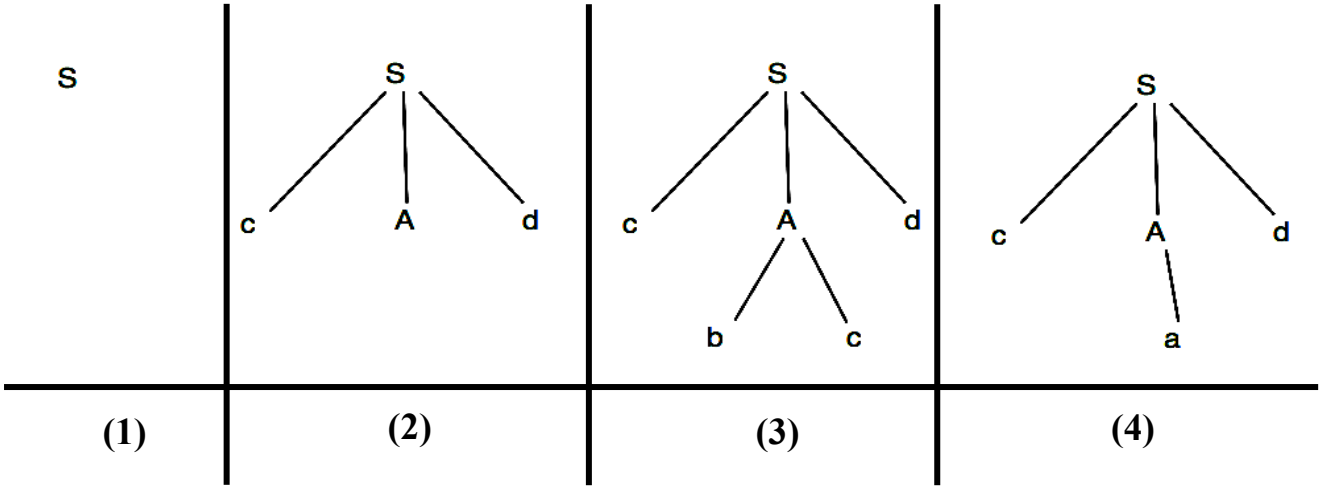
Suppose Production rules for the Grammar of a language are:

S → cAd

A → bc|a

And the input string is "cad".

Now the parser attempts to construct syntax tree from this grammar for the given input string. It uses the given production rules and applies those as needed to generate the string. To generate string "cad" it uses the rules as shown in the given diagram:-



In the step (3) above, the production rule A⟶bc was not a suitable one to apply (because the string produced is "cbcd" not "cad"), here the parser needs to backtrack, and apply the next production rule available with A which is shown in the step (4), and the string "cad" is produced.

Thus, the given input can be produced by the given grammar; therefore the input is correct in syntax. But backtrack was needed to get the correct syntax tree, which is really a complex process to implement.

Example (2):-

G= ({<exp>, <operand>, <id>},{a , b , c , + , - , (,) },<exp>, P)

T= {a , b , c , + , - , (,) }

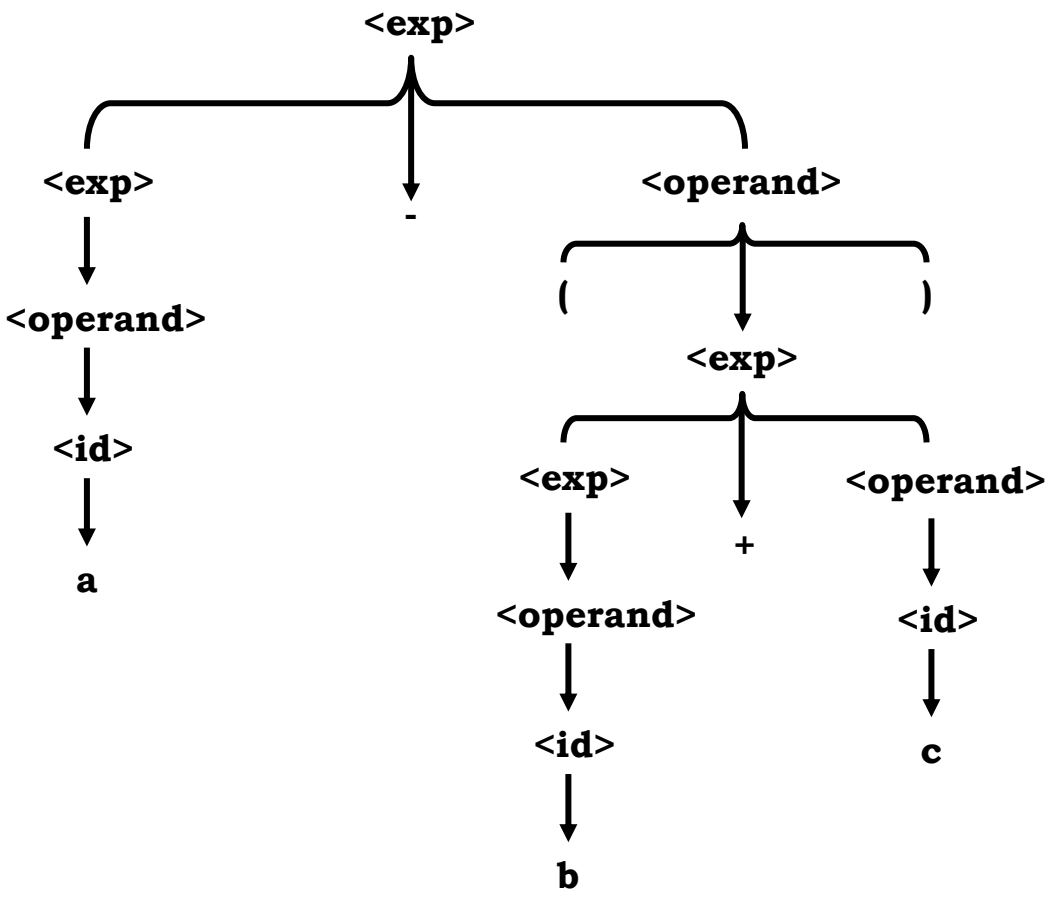
P=

<exp> ::= <operand> | <exp> + <operand> | <exp> - <operand>

<operand> ::= <id> | (<exp>)

<id> ::= a | b |c

Syntax analyzer utilizes syntax trees to determine whether a statement is accepted or not. Check if a-(b+c) accepted?



We can use another method to determine whether a statement is accepted or not, this method is called (*Derivation Method*).

There are two types of derivation:-

1. ***Leftmost derivation***
2. ***Rightmost derivation***

***Example* (3):-**

Let G be a grammar with this components ({S , E , F , P , R , L},{a , b , (,) , + , - , × , ^ , /}, S ,P)

P=

S→ E	**S→ +E**	**S→ -E**	**E→ T**
T→ F	**F→ P**	**P→ b**	**R→ a(L)**
E→ E+T	**E→T×F**	**F→ F^P**	**L→ S**
S→ E-T	**E→T/F**	**P→ a**	**P→ (S)**

Is a×(b+a) accepted or not?

***Leftmost derivation* :-**

S→ E→ T×F → F×F → P×F → a×F → a×P → a× (S) → a×(E)→ a×(E+T) → a×(T+T) → a×(F+T) → a×(P+T) → a×(b+T) → a×(b+F) → a×(b+P) →a×(b+a) ∴ a×(b+a) is accepted

***Rightmost derivation* :-**

S → E → T×F → T×P → T×(S) → T×(E) → T×(E+T) → T×(E+F) →T×(E+P) → T×(E+a) → T×(T+a) → T×(F+a) → T×(P+a) → T×(b+a) →F×(b+a) → P×(b+a) → a×(b+a) ∴ a× (b+a) is accepted

Context-Free Grammars:

The syntax of a programming language is described by context-free grammar (CFG). CFG consists of a set of terminals, a set of non-terminals, a start symbol, and a set of productions.

Ambiguity

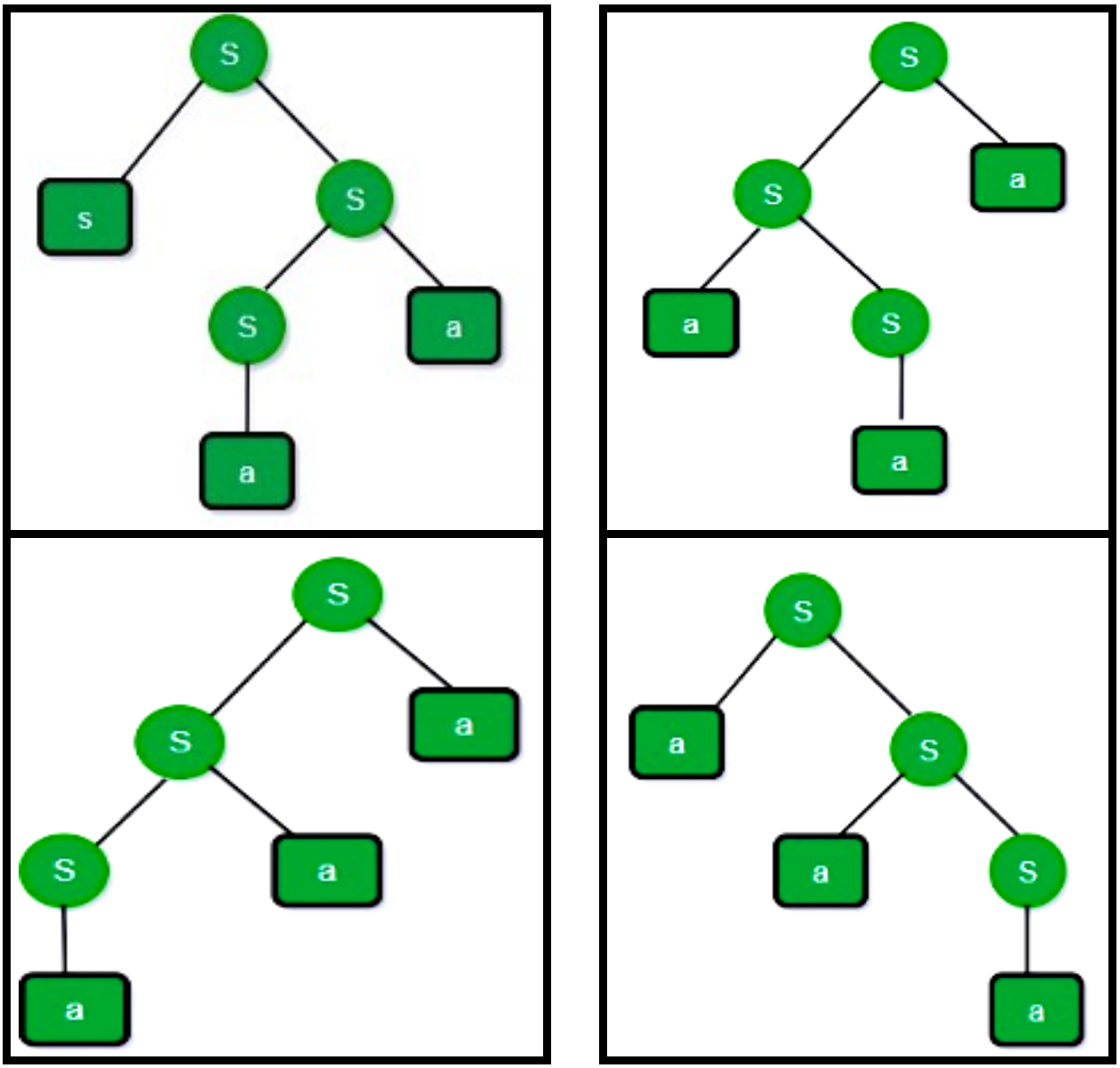
A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

Example:-

consider a grammar

S ⟶ aS | Sa | a

Now for string aaa, we will have 4 parse trees, hence ambiguous



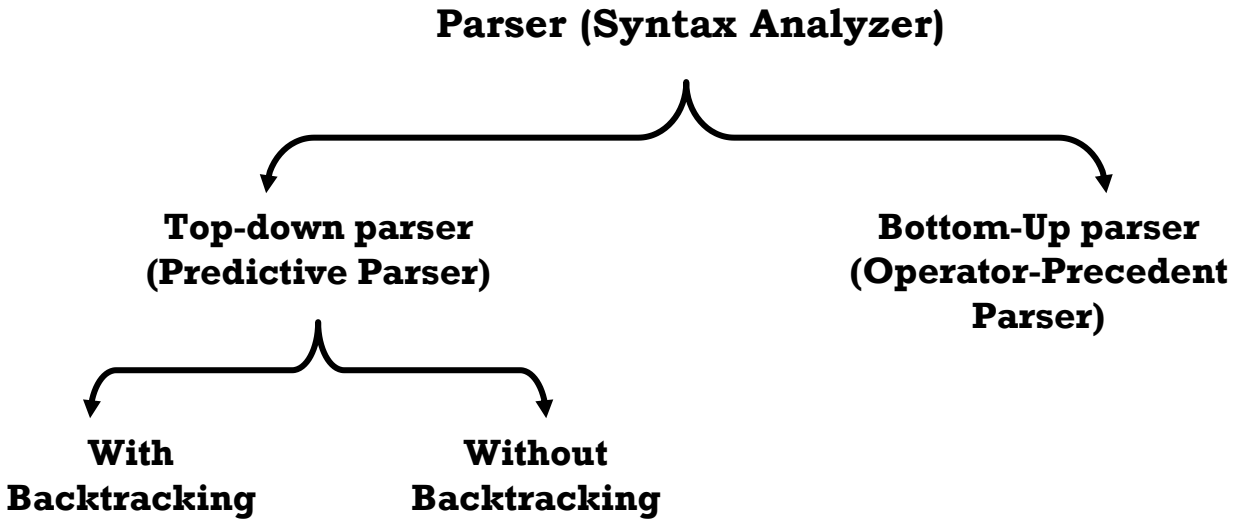
Parser Techniques

Types of Parsers in Compiler Design:-

The parser is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation. The parser is also known as Syntax Analyzer.

Types of Parser:

The parser is mainly classified into two categories, i.e. *Top-down Parser*, and *Bottom-up Parser*. These are explained below:-



1- Top-Down Parser:

The top-down parser is the parser that generates parse for the given input string with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.

Further Top-down parser is classified into two types: Recursive parser, and Non-recursive parser.

1. *Recursive parser* is also known as the *backtracking parser*. It basically generates the parse tree by using backtracking.

2. *Non-recursive parser* is also known as LL(1) parser or predictive parser or without backtracking parser. It uses a parsing table to generate the parse tree instead of backtracking.

2-Bottom-up Parser:

Bottom-up Parser is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the start symbol. It uses the reverse of the rightmost derivation. Further Bottom-up parser is classified into two types: *LR parser*, and *Operator precedence parser*.

LR parser is the bottom-up parser that generates the parse tree for the given string by using unambiguous grammar. It follows the reverse of the rightmost derivation.

LR parser is of four types:-

(a) LR(0)

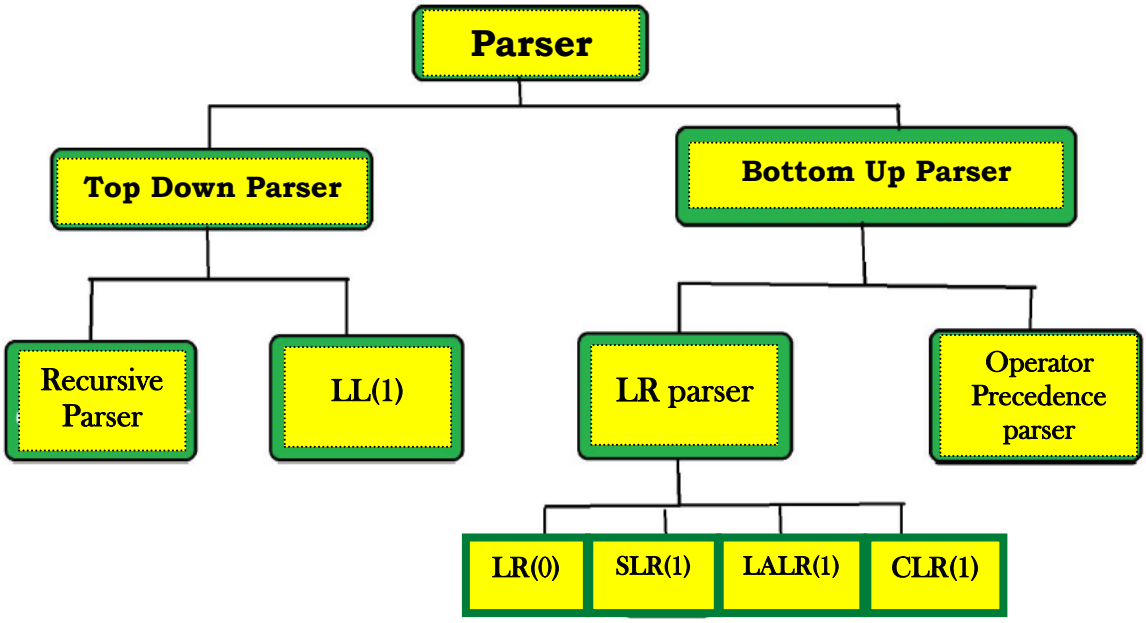
(b) SLR(1)

(c) LALR(1)

(d) CLR(1)

Operator precedence parser generates the parse tree form given grammar and string but the only condition is two consecutive

non-terminals and epsilon never appear on the right-hand side of any production.



Steps of parsing in LL(1) parser or predictive parser with or without backtracking:-

1- **Remove left recursion, because ambiguous not allowed in LL(1).**
2- **Compute FIRST and FOLLOW sets.**
3- **Construct the predictive parsing table using algorithm.**
4- **Parse string or statement using parser.**

Backtracking manipulating (Removing Left Recursion)

Left-Recursion Elimination إلغاء تكرار العنصر في أقصى يسار الطرف الأيمن

E → E+A

Left Recursion Elimination :-

Left Recursion Elimination is of two types:-

1. **Immediate Left-Recursion Elimination.**
2. **Not-Immediate Left-Recursion Elimination.**

Immediate Left-Recursion Elimination

A grammar is left recursive if it has a nonterminal (variable) S such that there is a derivation

$S \rightarrow S\alpha \mid \beta$

Where α and β (sequence of terminals and non-terminals that do not start with S)

Due to the presence of left recursion some top-down parsers enter into an infinite loop so we have to eliminate left recursion.

If we have a production of the form:-

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Where no βi begins with an A. The main rule for removing the immediate backtracking is by generating two rules as follows:-

A → β_1Á | β_2Á |...| β_nÁ ***(the first one depends on the part of the previous rule exactly on the part that not begins with A)***

Á → α_1Á |α_2Á |α_3Á |...|α_mÁ |ε ***(the second one depends on the part of the initial rule exactly on the part that begins with A)***

Example **(1):-**

S ⟶ S a b| S c d | S e f | g |h

Sol.

S ⟶ g S′| h S′

S′ ⟶ a b S′ | c d S′ | e f S′ | ε

Example **(2):-**

E → a b c | d e f | E r x

Sol.

É → a b c | d e f É

É → r x É | ε

Example **(3):-**

S ⟶ (L) |a (No left recursion)

L ⟶ L c S | S (left recursion)

Sol.

L ⟶ S L′

L′ ⟶ c S L′ | ε

Example **(4):-**

exp → exp or term | term

term → term and factor | factor

factor → not factor | (exp) | true | false

Sol.

exp → term exp´

exp´ → or term exp´ | ε

term → factor term´

term´ → and factor term´ | ε

factor → not factor | (exp) | true | false

Not Immediate Left-Recursion Elimination

Algorithm:-

Arrange NT in any order;

For I :=2 to n do

For J := 1 to i-1 do

Begin

Replace each production of the form $A_i \rightarrow A_J \alpha$ by the production

$A_i \rightarrow \partial_1 \alpha / \partial_2 \alpha / \partial_3 \alpha / \ldots / \partial_k \alpha$;

Where

$A_J \rightarrow \partial_1 / \partial_2 / \partial_3 / \ldots / \partial_k$ are the current A_J productions;

End;

Eliminate the immediate left recursion among the A_i productions;

End;{of algorithm}

Example (1):-

B → A c/d

A → Br/x

Solution:-

A_1=B A_2=A

$A_1 \rightarrow A_2 c/d$

$A_2 \rightarrow A_1 r/ x$

Replace:-	**$A_i \rightarrow A_J \alpha$**
By:-	**$A_i \rightarrow \partial_1 \alpha / \partial_2 \alpha / \partial_3 \alpha / \ldots / \partial_k \alpha$**
Using:-	**$A_J \rightarrow \partial_1 / \partial_2 / \partial_3 / \ldots / \partial_k$**

$A_2 \rightarrow A_1 r$ $\therefore \alpha = r$

.....

$A_2 \rightarrow \partial_1 \alpha / \partial_2 \alpha$

$A_1 \rightarrow \partial_1 / \partial_2$

$\because A_1 \rightarrow A_2c/d \qquad \therefore \partial_1 = A_2c$ and $\partial_1 = d$

I = 2	J = 1	α = r	$\partial_1 = A_2c$	∂_2 = d

$A_2 \rightarrow \partial_1 \alpha / \partial_2 \alpha$

$\therefore A_2 \rightarrow A_2c\ r / d\ r / x$

.....

$A_1 \rightarrow A_2c/d$

$A_2 \rightarrow A_2c\ r / d\ r / x$

These two rules are converted to immediate backtracking which can be eliminated by the following rules:-

$A \rightarrow A\alpha_1 / A\alpha_2 / A\alpha_3 / \ldots / A\alpha_m / \mathcal{B}_1 / \mathcal{B}_2 / \ldots / \mathcal{B}_n$

$A \rightarrow \mathcal{B}_1 Á / \mathcal{B}_2 Á / \ldots / \mathcal{B}_n Á$

$Á \rightarrow \alpha_1 Á / \alpha_2 Á / \alpha_3 Á / \ldots / \alpha_m Á / \varepsilon$

$B \rightarrow Ac/d$

$A \rightarrow Ac\ r / d\ r / x$

The result will be:-

$B \rightarrow Ac/d$

$A \rightarrow d\ r\ Á / x\ Á$

$Á \rightarrow c\ r\ Á / \varepsilon$

Example **(2):-**

$S \rightarrow A\ b / b$

$A \rightarrow Ac / Sd / e$

Another method to convert not immediate left recursion to immediate left recursion is by using substitution, as shown in the following example:-

S → A b / b

A → Ac / Sd/ e

The values of prameters $i, j, \alpha, \partial 1, \partial 2, \partial 3, \ldots$

- **Usually, (i) refers to the rule that contains the not immediate left recursion (rule no. 2), while (j) refers to the first rule (rule no.1).**
- **(α) represent the element next to the non terminal that causes the not immediate left recursion.**
- **($\partial 1, \partial 2, \partial 3, \ldots$) these values can get them from rule no.1 (the first rule), through taking the right hand side of the rule.**

Now, depending on the notes above,

Rule no. 1 S → A b / b (j=1) from this rule we can get the values of ($\partial 1, \partial 2, \partial 3, \ldots$), so $\partial 1$ = Ab and $\partial 2$ = b

Rule no. 2 A → Ac / Sd/ e (i=2), from this rule we can get the value of α = d

i=2 j=1 $\partial 1$ = Ab $\partial 2$ = b α = d

S → A b | b

A → Ac | Sd | e

……

S → A b | b

A → Ac | (A b | b) d | e

S → A b | b

A → Ac | A b d | b d | e

Now in this step, the not immediate left recursion is converted to immediate left recursion

……

S → A b | b

A → b d A′ | e A′

A′ → c A′ | b d A′ | ε

Now in this step, eliminate the immediate left recursion

……

***Example* (2):-**

B → A c | d rule no.1

A → B r | x rule no. 2

i=2 j=1 $\partial 1$ = **A c** $\partial 2$ = **d** α = **r**

B → A c | d

A → (A c | d) r | x

……

B → A c | d

A → A c r | d r | x

Now in this step, the not immediate left recursion is converted to immediate left recursion

……

B → A c | d

A → d r A′ | x A′

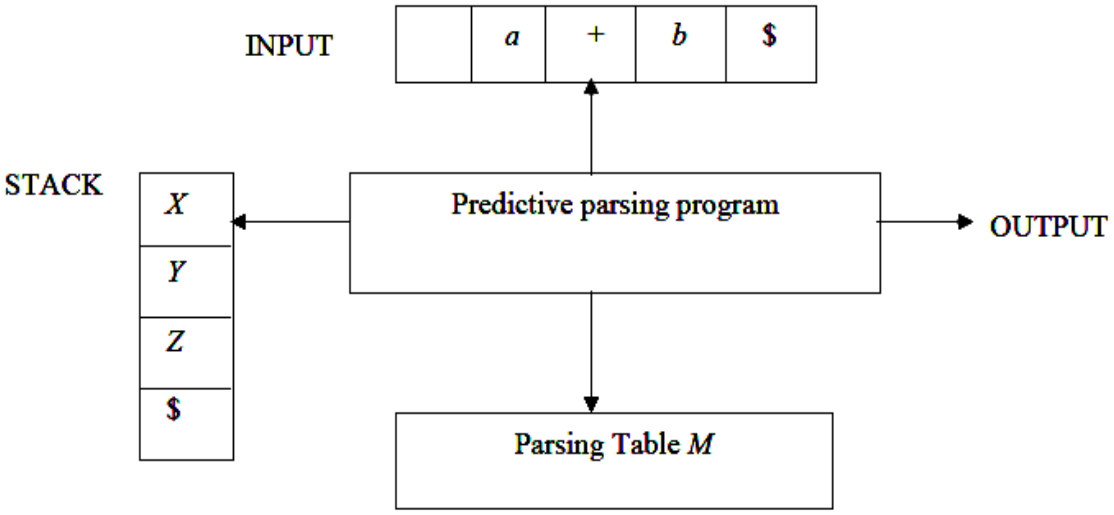
A′ → c r A′ | ε

Now in this step, eliminate the immediate left recursion

Predicative Parsing (Top Down Parser)

- **Predictive parsing is a special case of recursive descent parsing where no backtracking is required.**
- **The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives**

Non-recursive predictive parser architecture:-



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:- **It consists of strings to be parsed, followed by $ to indicate the end of the input string.**

Stack:- **It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. *Initially, the stack contains the start symbol on top of $*.**

Parsing table:- **It is a two-dimensional array M[A, a], where 'A' is a non-terminal and 'a' is a terminal.**

Previously, we talk about the steps of top-down parser with or without backtracking, as shown below:-

1- **Remove left recursion, because ambiguous not allowed in LL(1). (*note that, this step is previously explained*)**
2- **Compute FIRST and FOLLOW sets.**
3- **Construct the predictive parsing table using algorithm.**
4- **Parse string or statement using parser.**

Predictive parsing table construction

The construction of a predictive parser is aided by two functions associated with a grammar:-

1. FIRST

2. FOLLOW

FIRST Set in Syntax Analysis

FIRST(X) for a grammar symbol X is the set of terminals that begin the strings derivable from X.

Rules to compute FIRST set:-

1. If x is a terminal, then FIRST(x) = { 'x' }

2. If x ⟶ Є, is a production rule, then add Є to FIRST(x).

3. If X is non-terminal and X → a α is a production then add (a) to FIRST(X).

4. If X ⟶ Y1 Y2 Y3….Yn is a production,

- **a. FIRST(X) = FIRST(Y1)**
- **b. If FIRST(Y1) contains Є then FIRST(X) = { FIRST(Y1) – Є } U { FIRST(Y2) }**

c. If FIRST (Yi) contains Є for all i = 1 to n, then add Є to FIRST(X).

Example **(1):-**

Consider the following grammar:-

E → E+T | T

T → T*F | F

F → (E) | id

Sol.:-

Before calculating the first and follow functions, eliminate Left Recursion from the grammar, if present.

E → TE′

E′ → +TE' | ε

T → FT'

T′ → *FT′ | ε

F → (E) | id

Production Rules of Grammar		**FIRST sets**
E → TE′	**⟹**	**FIRST(E) = FIRST(T) = { (, id }**
E′ → +T E′|Є	**⟹**	**FIRST(E′) = { +, Є }**
T → F T′	**⟹**	**FIRST(T) = FIRST(F) = { (, id }**
T′→ *F T′ | Є	**⟹**	**FIRST(T′) = { *, Є }**
F → (E) | id	**⟹**	**FIRST(F) = { (, id }**

Example **(2):- Consider the following grammar:-**

S → A

A → aB / Ad

B → b

C → g

Sol.:-

Before calculating the first and follow functions, eliminate Left Recursion from the grammar, if present.

S → A

A → aBA'

A' → dA' / ∈

B → b

C → g

FIRST sets

Production Rules of Grammar			
S → A	⟹	First(S) = First(A) = { a }
A → aBA'	⟹	First(A) = { a }
A' → dA' / ∈	⟹	First(A') = { d , ∈ }
B → b	⟹	First(B) = { b }
C → g	⟹	First(C) = { g }

***Example* (3):- Consider the following grammar:-**

S → aBDh

B → cC

C → bC / ∈

D → EF

E → g / ∈

F → f / ∈

Sol.:-

FIRST sets

Production Rules of Grammar		
S → aBDh	⟹	First(S) = { a }
B → cC	⟹	First(B) = { c }
C → bC / ∈	⟹	First(C) = { b , ∈ }
D → EF	⟹	First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }
E → g / ∈	⟹	First(E) = { g , ∈ }
F → f / ∈	⟹	First(F) = { f , ∈ }

***Example* (4):- Consider the following grammar:-**

E → E + T / T

T → T x F / F

F → (E) / id

Sol.:-

Before calculating the first and follow functions, eliminate Left Recursion from the grammar, if present.

E → TE'

E' → + TE' / ∈

T → FT'

T' → x FT' / ∈

F → (E) / id

FIRST sets

Grammar Production Rules of

E → TE'	⟹	**First(E) = First(T) = First(F) = { (, id }**
E' → + TE' /∈	⟹	**First(E') = { + , ∈ }**
T → FT'	⟹	**First(T) = First(F) = { (, id }**
T' → x FT' / ∈	⟹	**First(T') = { x , ∈ }**
F → (E) / id	⟹	**First(F) = { (, id }**

FOLLOW Set in Syntax Analysis

Follow (X) to be the set of terminals that can appear immediately to the right of Non- Terminal X in some sentential form. That is mean; we calculate the follow function of a non-terminal by looking where it is present on the Right Hand Side (RHS) of a production rule.

ملاحظات مهمة:-

1- مجموعة (Follow) تعتمد على الجزء الايمن من كل (rule).

2- قيمة (Follow) للعنصر (start) دائما يساوي ($).

3- من غير الممكن ان تحتوي مجموعة (Follow) على (∈).

4- دائما يتم البحث عن العنصر المجاور الايمن للعنصر المطلوب ايجاد قيمة (Follow) له:-

أ- اذا كان العنصر من نوع (terminal) فان قيمة (Follow) ستكون نفس هذا العنصر (terminal T).

ب- اذا لم يكن هنالك عنصر مجاور ايمن فسوف تكون قيمة (Follow) لهذا العنصر هي مساوية لقيمة (Follow) للعنصر الموجود في الجزء الايسر من (rule).

ت- اذا كان العنصر المجاور الايمن من نوع (non terminal NT) فان قيمة (Follow) لهذا العنصر ستكون عبارة عن اتحاد كل من مجموعة (First) للعنصر المجاور الايمن مع حذف قيمة (∈) بالاضافة الى مجموعة (Follow) للعنصر الموجود في الجزء الايسر من (rule)

Rules For Calculating Follow Function:-

1- **If S is a start symbol, then FOLLOW(S) contains $, means, for the start symbol S, place $ in Follow(S). {Means *put $ (the end of input marker) in Follow(S) (S is the start symbol)*}**

2- **If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in Follow (B), means Follow(B) = First(β)**

3- If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B), means Follow(B) = Follow(A)

4- ∈ will never appear in the follow function of a nonterminal.

***Example* (1):- Consider the following grammar:-**

S → aBDh

B → cC

C → bC / ∈

D → EF

E → g / ∈

F → f / ∈

Sol.:-

Rule	First Set	Follow Set
S → aBDh	First(S) = { a }	Follow(S) = { $ }
B → cC	First(B) = { c }	Follow(B) = { First(D) – ∈ } ∪ First(h) = { g , f , h }
C → bC / ∈	First(C) = { b , ∈ }	Follow(C) = Follow(B) = { g , f , h }
D → EF	First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }	Follow(D) = First(h) = { h }
E → g / ∈	First(E) = { g , ∈ }	Follow(E) = { First(F) – ∈ } ∪ Follow(D) = { f , h }
F → f / ∈	First(F) = { f , ∈ }	Follow(F) = Follow(D) = { h }

***Example* (2):- Consider the following grammar:-**

E → E + T / T

T → T × F / F

F → (E) / id

Sol.:-

The given grammar is left recursive. So, we first remove left recursion from the given grammar. After eliminating left recursion, we get the following grammar-

E → TE′

E′ → + TE′ / ∈

T → FT′

T′ → ×FT′ / ∈

F → (E) / id

Rule	First Set	Follow Set
E → TE′	First(E) = First(T) = First(F) = { (, id }	Follow(E) = { $,) }
E′→ +TE′/ ∈	First(E′) = { + , ∈ }	Follow(E′) = Follow(E) = { $,) }
T → FT′	First(T) = First(F) = { (, id }	FOLLOW(T)={First(E′) – ∈}∪ Follow(E′) = { +, $,) }
T′→ ×FT′/∈	First(T′) = { × , ∈ }	Follow(T′) = Follow(T) = { + , $,) }
F → (E) / id	First(F) = { (, id }	Follow(F) = {First(T′) – ∈} ∪ Follow(T) = { ×, +, $,) }

Example **(3):- Consider the following grammar:-**

S → A

A → aB / Ad

B → b

C → g

Sol.:-

The given grammar is left recursive. So, we first remove left recursion from the given grammar. After eliminating left recursion, we get the following grammar-

S → A

A → aBA′

A′ → dA′ / ∈

B → b

C → g

Rule	First Set	Follow Set
S → A	First(S) = First(A) = { a }	Follow(S) = { $ }
A → aBA′	First(A) = { a }	Follow(A) = Follow(S) = { $ }
A′ →dA′/∈	First(A′) = { d , ∈ }	Follow(A′) = Follow(A) = { $ }
B → b	First(B) = { b }	Follow(B) = {First(A′) –∈ } ∪ Follow(A) = { d , $ }
C → g	First(C) = { g }	Follow(C) = empty set

Algorithm for construction of predictive parsing table

Input : Grammar G

Output : Parsing table M

Method :

1- **For each production A → α of the grammar, do steps 2 and 3.**

2- **For each terminal a in FIRST(α), add A → α to M[A, a].**

3- **If ε is in FIRST(α), add A → α to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) , add A →α to M[A, $].**

4- **Make each undefined entry of M be error.**

Predictive parsing program

Algorithm:-

Set IP (Input Pointer) point to the first symbol of the input string W$

Repeat

Let X be the top stack symbol and (a) be the symbol pointed by IP;

If X is a terminal or $ then

If X = a then

Pop X from the stack and advance IP

Else error()

Else

if M[X,a] = $X \rightarrow Y_1 Y_2 \dots Y_k$ then

Begin

Pop X from the stack

push Y_1 Y_2 ... Y_k on to stack with Y_1 on top

Output the production $X \rightarrow Y_1 Y_2 ... Y_k$

End

Else error();

Until X=$;

الشرط الأساسي للإعراب بطريقة (Top-Down) هو خلو القواعد من الرجوع الخلفي (Backtracking). فإذا كانت القواعد تحتوي على الرجوع الخلفي فلابد من التأكد من نوع الرجوع الخلفي فيما إذا كان من النوع المباشر (Immediate Backtracking) أو غير المباشر (Not-Immediate Backtracking) لكي يتم معالجته وفق الطرق التي تم شرحها مسبقاً.

نحتاج في هذه الخوارزمية إلى وجود Stack والعمليات الخاصة بها والتي تمثل Push & Pop .

يتم حساب قيمة (First and follow) من اجل تكوين جدول (Parse table) بعدد من الأسطر والأعمدة حيث أن عناصر الأسطر تمثل عناصر Non-Terminal أما قيم أو عناصر الأعمدة فتمثل عناصر Terminal، حسب ما تم التطرق اليه مسبقاً.

خطوات ما قبل الإعراب بهذه الطريقة :-

- تكوين جدول بخمسة أعمدة
  1. العمود الأول يمثل الرمز X والذي يمثل Top of Stack .
  2. العمود الثاني يمثل الرمز a والذي يمثل مؤشر يشير إلى الكلمة المطلوب إعرابها .
  3. العمود الثالث يمثل Stack .
  4. العمود الرابع يمثل عناصر الجملة المطلوب أعربها بالكامل.
  5. العمود الخامس والأخير يمثل Output والذي يحتوي على العلاقات ما بين العناصر terminal والعناصر Non-terminal .
- القيمة الابتدائية للعمود الثالث (Stack) تحتوي على Start Symbol $.
- القيمة الابتدائية للعمود الرابع (Input) هي الجملة المطلوب إعرابها.
- القيمة الابتدائية للعمود الخامس والأخير تكون فارغة.

- القيمة الابتدائية للعمود الأول تعتمد على ما موجود في العمود الثالث وتمثل Top of Stack .
- القيمة الابتدائية للعمود الثاني تعتمد على ما موجود في العمود الرابع وتمثل لعنصر الموجود في أقصى يسار الجملة المطلوب إعرابها.

طريقة الإعراب:-

1. عندما يكون X من نوع Terminal لابد من ملاحظة إذا كان X=a

⇐إذا تحقق الشرط نقوم بعملية سحب قيمة X والذي يمثل Top of Stack ونأخذ العنصر التالي في الجملة المطلوب إعرابها (أي إن قيمة العمود a تتغير وكذلك تتغير قيمة العمود الرابع Input وأيضا قيمة العمود الثالث والذي يمثل Stack).

⇐إذا لم يتحقق الشرط أعلاه أي إن (X ≠ a) معناه أن الجملة المطلوب إعرابها تكون غير مقبولة (Not accepted).

2. عندما يكون X من نوع Not-Terminal فنبحث عن علاقة X مع a في الجدول (Parse table) أي تقاطع السطر X مع العمود a وان تلك العلاقة سوف يتم إضافتها في العمود الخامس وسحب من Stack العنصر الموجود في القمة وعمل Push للطرف الأيمن من العلاقة ولكن بالمقلوب ويبقى حقل a بدون تغيير وكذلك حقل Input.
3. نستمر بتكرار الخطوات الأولى والثانية طالما قيمة Stack ≠ $.

Example ①:-

Having the following grammar:-

E→E+T / T

T→T×F / F

F→(E) / id

Show the moves made by the Top-Down Parser on the input=id+id×id$

1- We must solve the left recursion and left factoring if it founded in the grammar

تحتوي هذه القواعد على رجوع خلفي من نوع المباشر فلابد من معالجة الرجوع الخلفي قبل البدء بعملية الإعراب.

E → T E´

E´→ +T E´ / ε

T → F T´

T´→ ×F T´ / ε

F → (E) / id

2- We must find the first and follow to the grammar:

Rule	First Set	Follow Set
E → TE′	First(E) = { (, id }	Follow(E) = { $,) }
E′→ +TE′/ ε	First(E′) = { + , ε }	Follow(E′) = { $,) }
T → FT′	First(T) = { (, id }	Follow(T) = { +, $,) }
T′→ ×FT ε	First(T′) = { × , ε }	Follow(T′) = { + , $,) }
F → (E) / id	First(F) = { (, id }	Follow(F) = { ×, +, $,) }

3- We must find or construct now the predictive parsing table

	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 → id			F → (E)		

4- Parse string or statement using parser.

X	a	Stack	Input	Output
E	id	$E	id+id×id$	----------
T	id	$E´T	id+id×id$	E → TE´
F	id	$E´ T´ F	id+id×id$	T →FT´
id	id	$E´ T´ id	id+id×id$	F → id
T´	+	$E´ T´	+id×id$	Pop id
E´	+	$E´	+id×id$	T´ → ε
+	+	$E´ T+	+id×id$	E´ →+TE´
T	id	$E´ T	id×id$	Pop +
F	id	$E´ T´ F	id×id$	T →FT´
id	id	$E´ T´ id	id×id$	F → id
T´	×	$E´ T´	×id$	Pop id
×	×	$E´ T´ F×	×id$	T´ →×FT´
F	id	$E´ T´ F	id$	Pop ×
id	id	$E´ T´ id	Id$	F → id
T´	$	$E´ T´	$	Pop id
E´	$	$E´	$	T´→ε
$	$	$	$	E´→ε
Stop				

Example **②:- Having the following grammar, parse the following statement:-** ***not (true or false) $***

exp → exp or term | term

term → term and factor | factor

factor → not factor |(exp) |true | false

1- We must solve the left recursion and left factoring if it founded in the grammar

تحتوي هذه القواعد على رجوع خلفي من نوع المباشر فلابد من معالجة الرجوع الخلفي قبل البدء بعملية الإعراب.

exp → term exp′

exp′ → or term exp′ | ε

term → factor term′

term′ → and factor term′| ε

factor → not factor |(exp) |true | false

2- We must find the first and follow to the grammar:

3- We must find or construct now the predictive parsing table, the resultant table will be as follows:-

	not	or	and	(	)	true	false	$
exp	exp→ term exp′			exp→ term exp′		exp→ term exp′	exp→ term exp	
exp′		exp′→ or term exp′			exp′→ϵ			exp′→ϵ
term	term→ factor term′			term→ factor term′		term→ factor term′	term→ factor term′	
term′		term′ → or factor term′	term′ → and factor term′		term′→ϵ			term′→ϵ
factor	factor→ not factor			factor → (exp)		factor→ true	factor → false	

Example ②:-

Having the following grammar:-

exp → exp or term | term

term → term and factor | factor

factor → not factor | (exp) | true | false

Parse the following statement:- ***not (true or false) $***

Sol.

1- We must solve the left recursion and left factoring if it founded in the grammar

exp → term exp'

exp' → or term exp | ∈

term → factor term'

term' → and factor term'| ∈

factor → not factor | (exp) | true | false

2- We must find the first and follow to the grammar:

Rule	First Set	Follow Set
exp → term exp'	First (exp)={not,(,true,false}	Follow (exp) = { $,) }
exp' → or term exp' | ϵ	First(exp′) = {or,∈ }	Follow (exp′) = { $,) }
term → factor term'	First(term)={not,(,true,false}	Follow (term) = first ((exp')-∈) ∪ follow (exp)= { or , $,) }
term' → and factor term'|ϵ	First(term′) = {and , ∈}	Follow(term′) = follow (term)= { or , $,) }
factor → not factor | (exp) | true | false	First(factor)={not,(,true,false}	Follow(factor) = first ((term')-∈) ∪ follow (term)= {and, or , $,)}

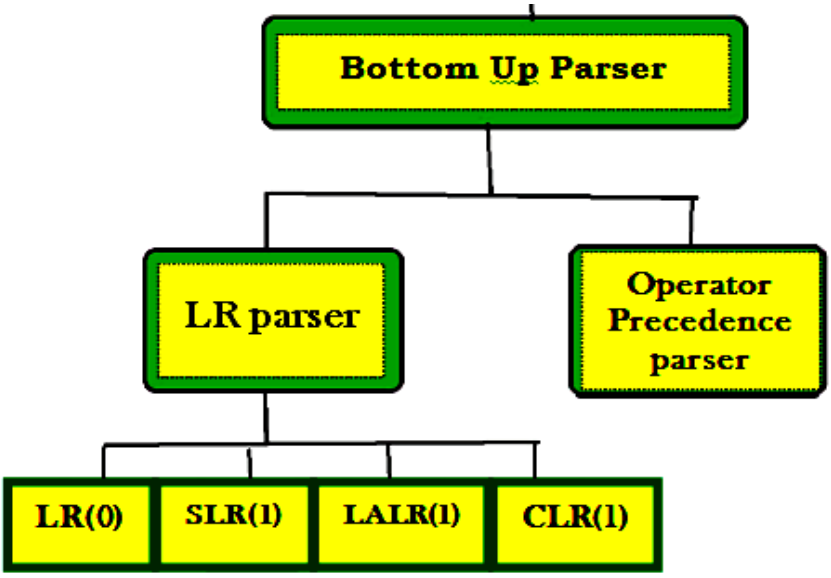
3- We must find or construct now the predictive parsing table

	not	or	and	(	)	true	false	$
exp	exp→ term exp′			exp→ term exp′		exp→ term exp′	exp→ term exp	
exp′		exp′→ or term exp′			exp′→ϵ			exp′→ϵ
term	term→ factor term′			term→ factor term′		term→ factor term′	term→ factor term′	
term′			term′ → and factor term′		term′→ϵ			term′→ϵ
factor	factor→ not factor			factor → (exp)		factor→ true	factor → false	

4- Apply parsing algorithm to parse the statement *not (true or false) $*

X	a	Stack	Input	Output
exp	not	$exp	not (true or false) $	----------
term	not	$ exp' term	not (true or false) $	exp→ term exp'
factor	not	$ exp' term' factor	not (true or false) $	term→ factor term'
not	not	$ exp' term' factor not	not (true or false) $	factor→ not factor
factor	(	$ exp' term' factor	(true or false) $	pop not
(	(	$ exp' term') exp (	(true or false) $	factor→ (exp)
exp	true	$ exp' term') exp	true or false) $	pop (
term	true	$ exp' term') exp' term	true or false) $	exp→ term exp'
and so on until we reach to to stop condition when stack=$ only				

Bottom Up Parser (Shift-Reduce Parser)



Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.

There is a general style of bottom-up syntax analysis, known as shift reduces parsing.

Is a right most derivation for a sentential form in reverse order.

Conditions for Bottom-Up Parser:-

1. **No ε-rules (i.e., A → ε).**
2. **It must be operator grammar (i.e., no adjacent non-terminal).**

Example **①:- E → E A E / (E) / -E / id**

Since of this production rule, the grammar is not operator grammar (E=NT, A=NT, E=NT).

Example **②:- E → E + E / E-E**

This grammar is an operator grammar (E is NT, + is T, E is NT).

SHIFT-REDUCE PARSING (Operator Precedence Parser)

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example: **Consider the grammar:**

S → aABe

A → Abc | b

B → d

The sentence to be recognized is abbcde.

REDUCTION (LEFTMOST)		RIGHTMOST DERIVATION
abbcde	(A → b)	S → aABe
aAbcde	(A → Abc)	→ aAde
aAde	(B → d)	→ aAbcde
aABe	(S → aABe)	→ abbcde
S		

We need to do a table with three fields (Stack, Input, action {which will be either shift or reduce}).

Actions in SHIFT-REDUCE PARSING

- **Shift - The next input symbol is shifted onto the top of the stack**
- **Reduce – the parser replaces the handle within a stack with a non-terminal.**
- **Accept – the parser announces successful completion of parsing.**
- **Error – the parser discovers that a syntax error has occurred and calls an error recovery routine.**

Initial value for stack=$.

Initial value for input=the sentence which we want to parse.

Initial value for action = Shift.

We need to know the meaning of the handle.

Definition: **a handle is a substring that:-**

1- Matches a right hand side of a production rule in the grammar

2- Whose reduction to the non-terminal on the left hand side of that grammar rule is a step along the reverse of a rightmost derivation.

- الشرط الأساسي للإعراب بطريقة (Bottom-Up) هو خلو القواعد من (ε) Empty word وان تكون من نوع (Operator grammar) أي عدم وجود عناصر متجاورة من نوع Non-Terminal.
- لا تهتم هذه الطريقة بوجود أو عدم وجود رجوع خلفي في القواعد المطلوب التعامل معها.
- نحتاج في هذه الخوارزمية إلى وجود Stack والعمليات الخاصة بها والتي تمثل Push & Pop.

خطوات الخوارزمية :-

- تكوين جدول بثلاثة أعمدة:-
  1. العمود الأول يمثل Stack .
  2. العمود الثاني يمثل عناصر الجملة المطلوب أعربها بالكامل (Input).
  3. العمود الثالث والأخير يمثل Action والذي يمثل عمليتين أساسيتين هما Shift & Reduce.
- القيمة الابتدائية للعمود الأول (Stack) تحتوي فقط على $.
- القيمة الابتدائية للعمود الثاني (Input) هي الجملة المطلوب إعرابها.
- القيمة الابتدائية للعمود الثالث والأخير تكون Shift وتمثل عملية Push للعنصر الموجود في أقصى يسار العمود الثاني ودفع العنصر في Stack.
- لابد من تطبيق Right Most Derivation على القواعد المعطاة.
- بعد الخطوة السابقة مباشرة وبالاعتماد عليها يتم تحديد ما يسمى بـ (Handle) والتي سوف يعتمد عليها قيم العمود الثالث (Action).
- اشتقاق القواعد باستخدام (Tree).
- أول مرحلة تمثل حالة إضافة العنصر الموجود في أقصى يسار الجملة المطلوب إعرابها وإضافته إلى (Top of Stack).
- ملاحظة إذا كان العنصر الذي تم إضافته إلى(Top of Stack) في الخطوة السابقة هل هو (Handle) أم لا، إذا كان (Handle) فيتم إرجاع العنصر إلى أصله وإذا لم يكن (Handle) فيتم إضافته إلى (Top of Stack).
- نستمر بالخطوات السابقة الى ان تكون قيمة الحقل الأول (Stack=$Start Symbol).

***Example* ①:-**

S → S×S / S+S / id **Input = id×id+id$**

Sol.

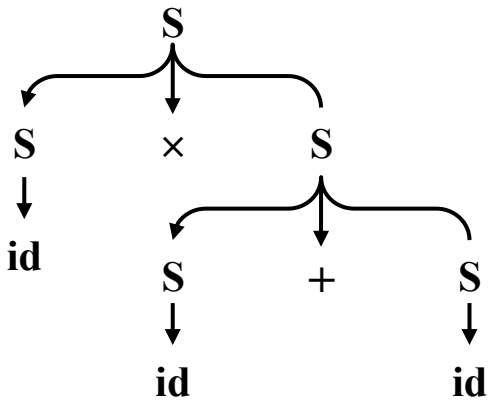
① ***Derive this grammar using right most derivation***

S → S×S → S×S+S → S×S+id → S×id+id → id×id+id

②*Specify the handles (using the above derivation)*

S → S×S → S× S+S → S×S+ id → S× id +id → id ×id+id

③*Doing Syntax tree (parse tree)*



④*Doing Parse table*

Stack	Input	Action
$	id×id+id$	Shift
$ id	×id+id$	Reduce S →id
$ S	×id+id$	Shift
$ S×	id+id$	Shift
$ S×id	+id$	Reduce S →id
$ S×S	+id$	Shift
$ S×S+	id$	Shift
$ S×S+id	$	Reduce S →id
$ S×S+S	$	Reduce S →S+S
$ S×S	$	Reduce S →S×S
$ S	$	Accept

Example **②:-**

E → T / E+T / E-T / -T

T → F / T×F/ T/F

F → (E) / id

Input = -(id×(id-id) / id)$

Solution **:-**

E → -T

→ -F

→ -(E)

→ -(T)

→-(T/F)

→-(T/ id)

→-(T×F / id)

→-(T× (E) /id)

→ -(T×(E-T) /id)

→ -(T×(E - F) /id)

→ -(T×(E - id) /id)

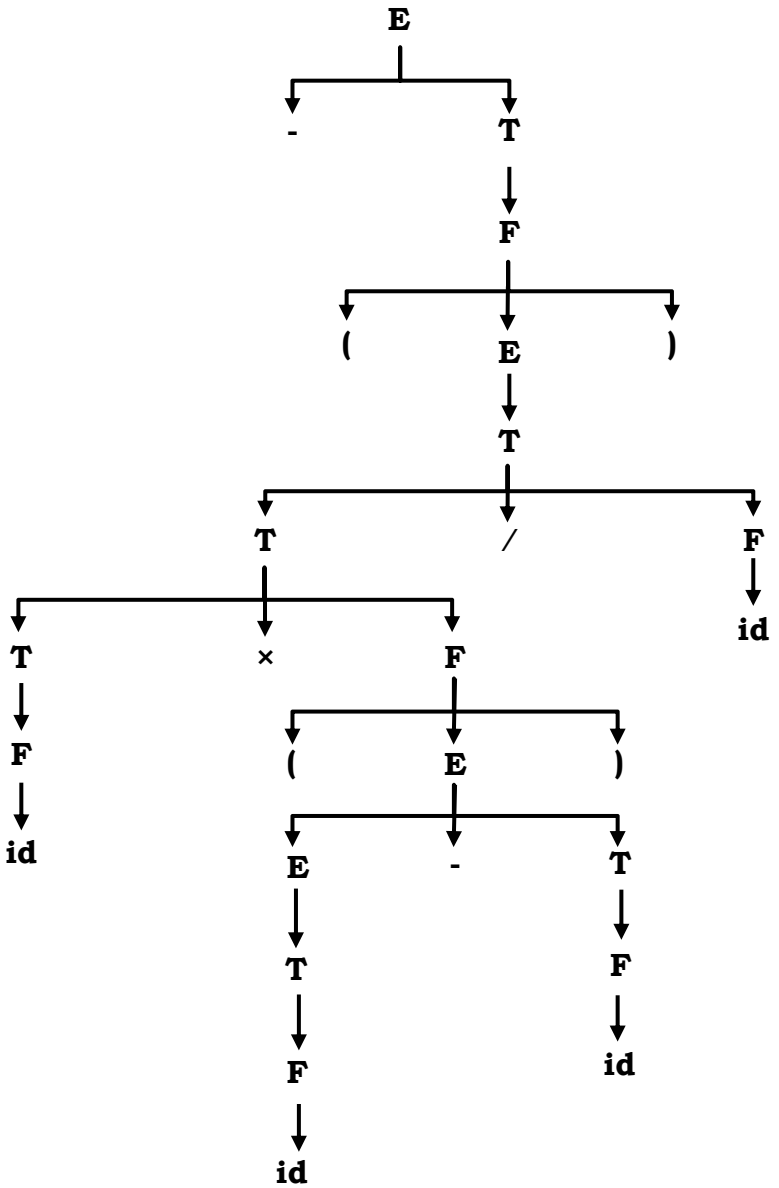
→ -(T×(T - id) /id)

→ -(T×(F - id) /id)

→ -(T×(id - id) /id)

→ -(F × (id - id) /id)

→ -(id × (id - id) /id)



Stack	Input	Action
$	-(id×(id-id)/id)$	Shift
$-	(id×(id-id)/id)$	Shift
$-(	id×(id-id)/id)$	Shift
$-(id	×(id-id)/id)$	Reduce F →id
$-(F	×(id-id)/id)$	Reduce T →F
$ -(T	×(id-id)/id)$	Shift
$ -(T×	(id-id)/id)$	Shift
$ -(T×(	id-id)/id)$	Shift
$ -(T×(id	-id)/id)$	Reduce F →id
$ -(T×(F	-id)/id)$	Reduce T →F
$ -(T×(T	-id)/id)$	Reduce E →T
$ -(T×(E	-id)/id)$	Shift
$ -(T×(E-	id)/id)$	Shift
$ -(T×(E-id	)/id)$	Reduce F →id
$ -(T×(E-F	)/id)$	Reduce T →F
$ -(T×(E-T	)/id)$	Reduce E →E-T
$-(T×(E	)/id)$	Shift
$-(T×(E)	/id)$	Reduce F → (E)
$-(T×F	/id)$	Reduce T →T×F
$-(T	/id)$	Shift
$-(T/	id)$	Shift

$-(T/id	)$	Reduce F →id
$-(T/F	)$	Reduce T →T/F
$-(T	)$	Reduce E → T
$-(E	)$	Shift
$-(E)	$	Reduce F → (E)
$-F	$	Reduce T →F
$-T	$	Reduce E → -T
$E	$	Accept

LR Parser

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(*k*) parsing. The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse.

Advantages of LR Parser:-

- ✓ It is an efficient non-backtracking shift-reduce parsing method.
- ✓ A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- ✓ It detects a syntactic error as soon as possible.

Types of LR Parsing method:-

1. SLR- Simple LR
   - Easiest to implement, least powerful.
2. CLR- Canonical LR
   - Most powerful, most expensive.
3. LALR- Look-Ahead LR
   - Intermediate in size and cost between the other two methods.

Let us see the comparison between SLR, CLR, and LALR Parser.

SLR Parser	LALR Parser	CLR Parser
It is very easy and cheap to implement.	It is also easy and cheap to implement.	It is expensive and difficult to implement.
SLR Parser is the smallest in size.	LALR and SLR have the same size. As they have less number of states.	CLR Parser is the largest. As the number of states is very large.
Error detection is not immediate in SLR.	Error detection is not immediate in LALR.	Error detection can be done immediately in CLR Parser.
SLR fails to produce a parsing table for a certain class of grammars.	It is intermediate in power between SLR and CLR i.e., SLR ≤ LALR ≤ CLR.	It is very powerful and works on a large class of grammar.
It requires less time and space complexity.	It requires more time and space complexity.	It also requires more time and space complexity.