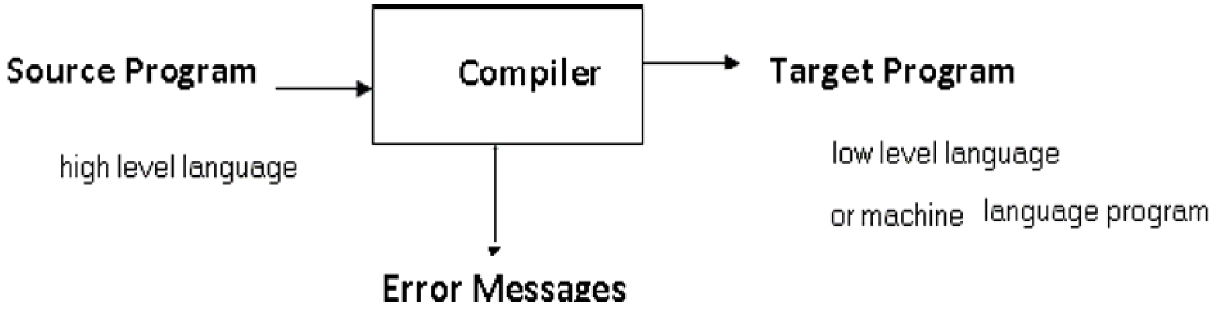
A Compiler

Is a program that reads a program written in one language -the Source Language- and translates it into an equivalent program in another language - the Target Language -. A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.

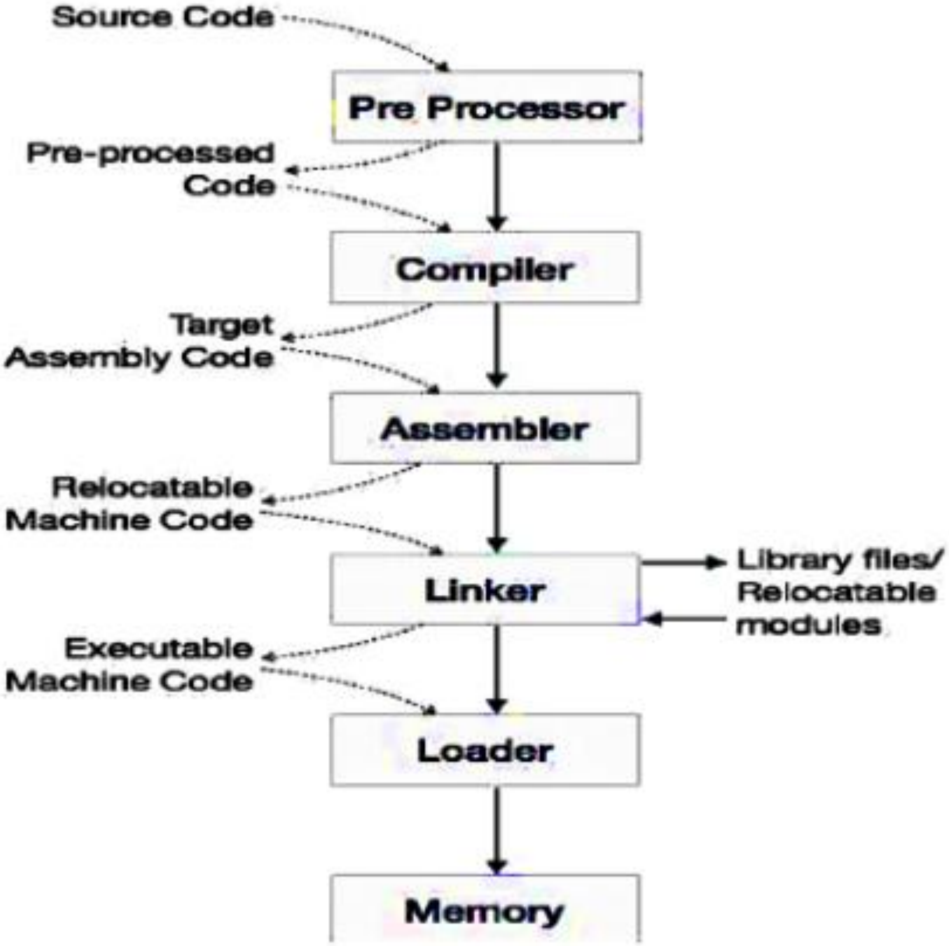


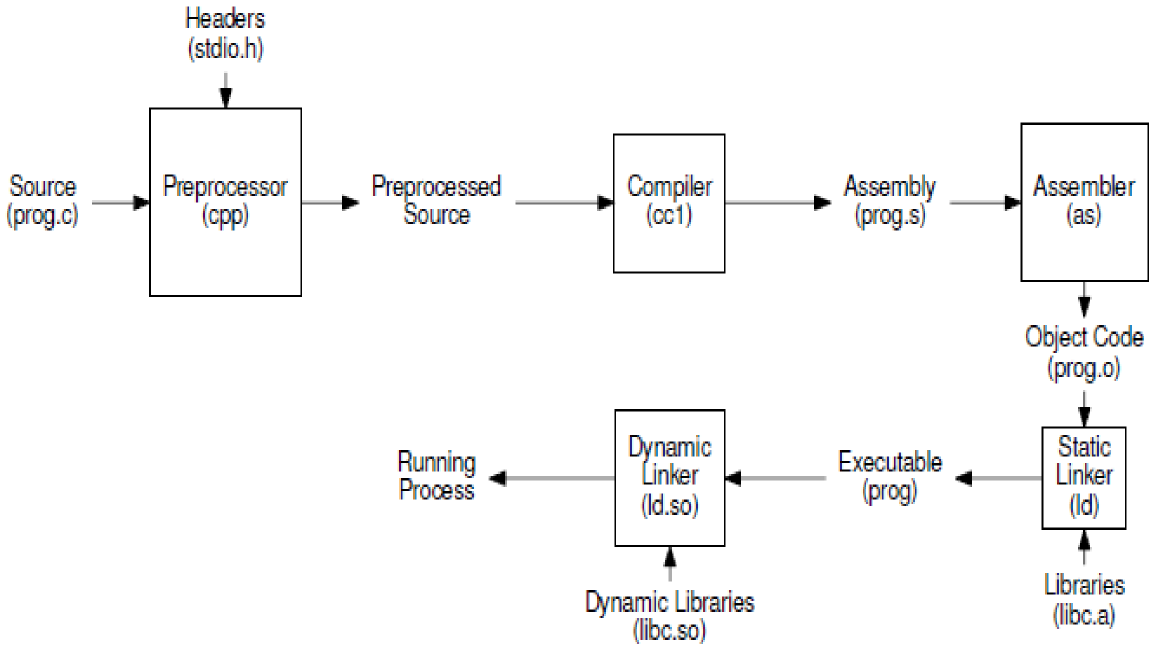
Compiler Design

Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by compatible software. Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming. Binary language has only two alphabets, 0 and 1. To instruct, the hardware codes must be written in binary format, which is simply a series of 1s and 0s. It would be a difficult task for computer programmers to write such codes, which is why we have compilers to write such codes.

Language Processing System

Any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.





The high-level language is converted into binary language in various phases. A compiler is a program that converts high-level language to assembly language. Similarly, an assembler is a program that converts the assembly language to machine-level language. Let us first understand how a program, using C compiler, is executed on a host machine.

1. User writes a program in C language (High-Level Language).

2. The C *compiler* compiles the program and translates it to assembly program (Low-Level Language).

3. *An Assembler* then translates the assembly program into machine code (object).

4. *A Linker* tool is used to link all the parts of the program together for execution (Executable Machine Code).

5. *A Loader* loads all of them into memory and then the program is executed.

Preprocessor A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers.

Interpreter An *interpreter*, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A *compiler* reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an *interpreter* reads a statement from the input converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. Whereas a compiler reads the whole program even if it encounters several errors.

Assembler An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker Linker is a computer program that links and merges various object files together in order to make an executable file. The major task of a linker is to determine the memory location where these files will be loaded.

Loader Loader is a part of operating system and is responsible for loading executable files into memory and executes them. It calculates the size of a program (instructions and data) and creates memory space for it.

***Compiler Architecture*:-**

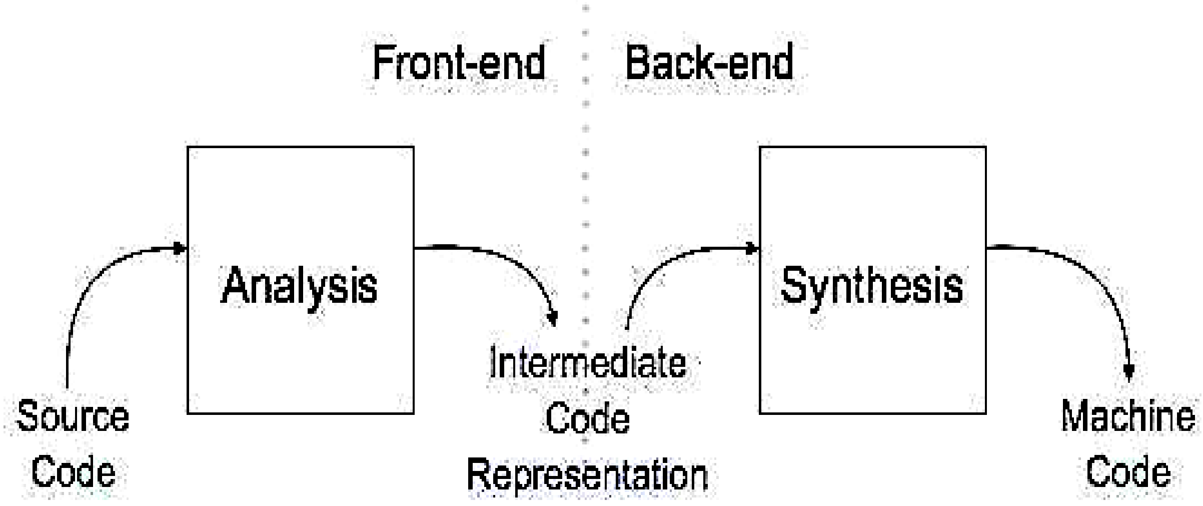
A compiler can broadly be divided into two phases based on the way they compile.

1. *Analysis Phase*

Known as the Front-End of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

2. *Synthesis Phase*

Known as the Back-End of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table.

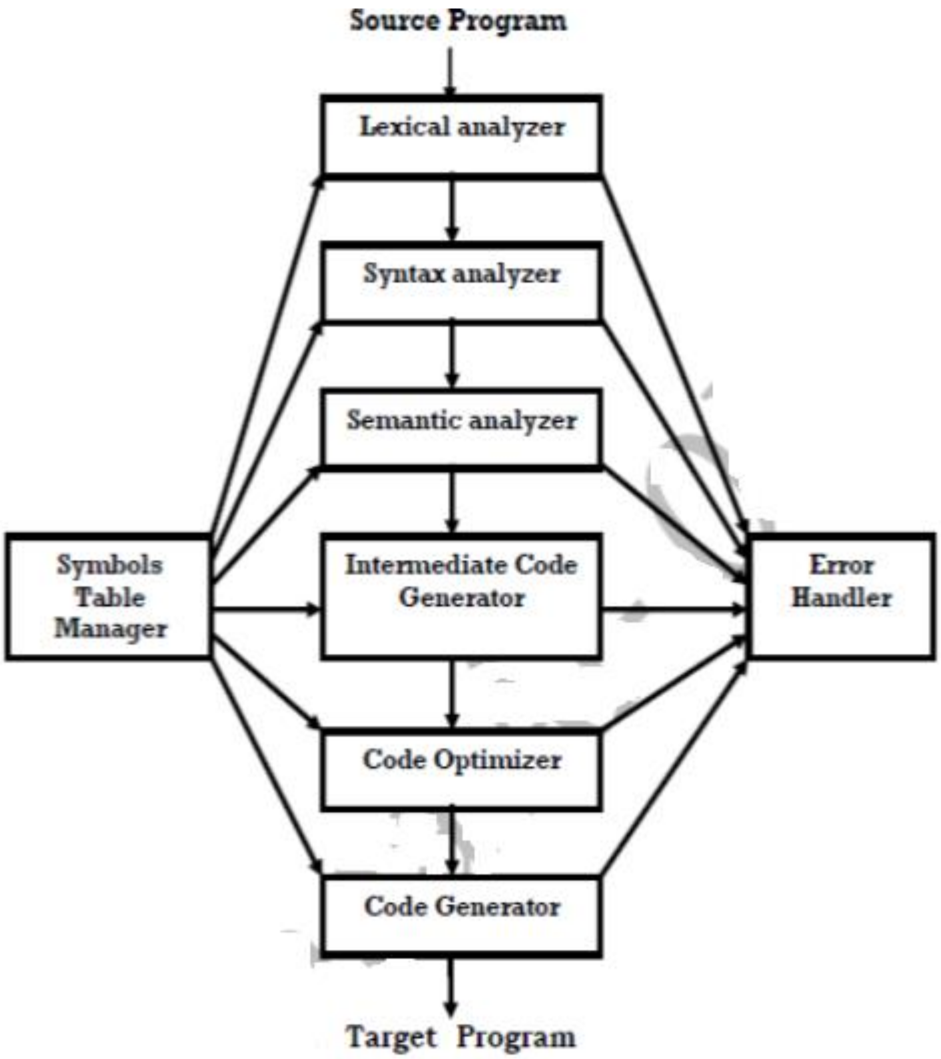


***The Phases of a Compiler* :-**

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

1. Lexical Analyzer. اللفظي التحليل مرحلة
2. Syntax Analyzer. القواعدي التحليل مرحلة
3. Semantic Analyzer. المعنوي التحليل مرحلة
4. Intermediate Code Generator. الوسطية الشفرات توليد مرحلة
5. Code Optimizer. الشفرات تحسين مرحلة
6. Code Generator. الشفرات مولد

In each phase we need variables that can be obtained from a table called *Symbol Table manager*, and in each phase some errors may be generated so we must have a program used to handle these errors , this program called *Error Handler*.



 Lexical Analyzer :- Is the initial part of reading and analyzing the program text (source program); The text is read (character by character) and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

 Syntax analyzer :- The next phase is called the syntax analysis or parsing. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree) that reflects the structure of the program. This phase is often called parsing.

 Semantic Analysis*:-* Semantic analysis checks whether the parse tree constructed follows the rules of language. Also is known as Type checking which main function is to analyze the syntax tree to determine if the program violates certain consistency requirements, such as, if a variable is used but not declared, assignment of values is between compatible data types, and adding string to an integer.

- ***Intermediate Code Generator*** :- After syntax and semantic analysis, It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code. This phase bridges the analysis and synthesis phases of translation.
- ***Code Optimization phase*** :- The code optimization phase attempts to improve the intermediate code which results that the output runs faster and takes less space. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).
- ***Code Generator*** :- The final phase of complier is the generation of target code, which represents the output of the code generator in the machine language.

- ***Symbol Table*** :- It is a data-structure maintained throughout all the phases of a compiler. All the identifiers' names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it.
- ***Error Handler*** :- Each phase can produce errors. However, after detecting an error, a phase must deal with that error, so that the compilation can proceed. So dealing with that error is done by a program known as Error Handler which is software used to handle any error that may be produced from any phase and it is needed in all phases of the compliers.

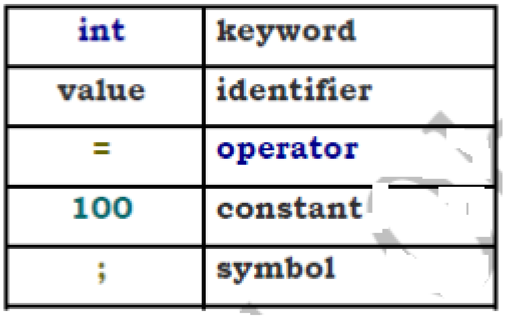
Note :- Each phase of the complier has two inputs and two outputs; for example:- for the first phase (*Lexical Analyzer*) the first input to it is the source program while the second input is some variables that may be needed in that phase; while the first output is the errors that may be generated in it and will be manipulated by the Error Handler program, and the second output from it will represent the input for the next compiler phase (Syntax).

Lexical Analysis:- A Review

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

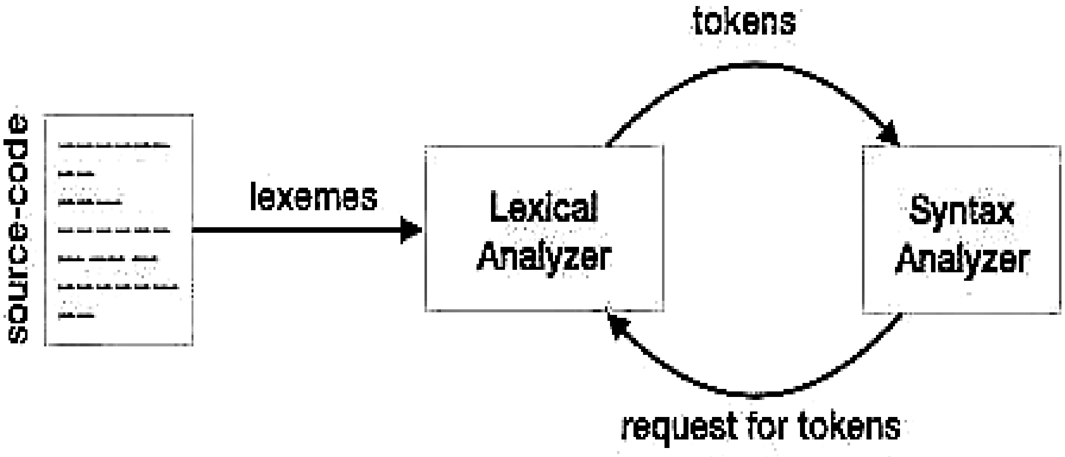
strings, numbers, operators and punctuations symbols can be considered as tokens.
For example, in C language, the variable declaration line

int value = 100;
Contains the tokens:-



int	keyword
value	identifier
=	operator
100	constant
;	symbol

The lexical analyzer also follows rule priority where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.
If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Specifications of Tokens
Let us understand how the language theory undertakes the following terms:

Alphabets
Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string tutorials point is 14 and is denoted by |tutorialspoint| = 14. A string having no alphabets, i.e. a string of zero length is known as an empty string and is denoted by ε (epsilon).

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions. The various operations on languages are:

□ **Union** of two languages L and M is written as

L U M = {s | s is in L or s is in M}

□ **Concatenation** of two languages L and M is written as

LM = {st | s is in L and t is in M}

□ The **Kleene** Closure of a language L is written as

L* = Zero or more occurrence of language L.

Grammars

A grammar is a set of formal rules for constructing correct sentences in any language; such sentences are called *Grammatical Sentences*.

Concatenation

We define the *Concatenation* of two symbols U and V by:-

UV= { X | X= uv, u is in U and v is in V }

Note that:- UV ≠ VU

U (VW) = (UV) W

Example □:-

Let Σ = {0,1} and U= {000,111} and V= {101,010}

UV= {000101, 000010, 111101, 111010}

VU= {101000, 101111, 010000, 010111}

UV ≠ VU

Example □:-

Let Σ = {a,b,c,d} ; U= {abd , bcd} ; V= {bcd , cab} and W= {da , bd}

To prove the following:- U (VW) = (UV) W

First, take the left side;

U (VW) ={abd , bcd} {bcdda, bcdbd, cabda, cabbd}
= { abdbcdda, abdbcdbd, abdcabda, abdcabbd, bcdbcdda, bcdbcdbd, bcdcabda, bcdcabbd }

Second, take the right side;

(UV) W = { abdbcd, abdcab, bcdbcd, bcdcab} {da , bd}
= { abdbcdda, abdcabda, bcdbcdda, bcdcabda, abdbcdbd, abdcabbd, bcdbcdbd, bcdcabbd }

□ U (VW) = (UV) W

Closure or Star Operation **:-**

This operation defines on a set S, a derived set S*, having as members the empty word and all words formed by concatenating a finite number of words in S, as shown below:-

$$S^* = S^0 \cup S^1 \cup S^2 \cup \ldots\ldots$$

Where :-

$$S^0 = \varepsilon \quad \text{and} \quad S^i = S^{i-1}S \quad \text{for} \quad i > 0$$

Example **:-**

Let S = {01, 11}, then

$$S^* = \{\underset{S^0}{\underset{\uparrow}{\varepsilon}}, \underset{S^1}{\underset{\uparrow}{01, 11}}, \underbrace{0101, 0111, 1101, 1111}_{S^2}, \underbrace{010101, 010111, \ldots}_{S^3}\}$$

Formalization**:-**

A phrase structure grammar is of the form G= (N, T, S, P); where:-

N = A finite set of non-terminal symbols denoted by A, B, C,...

T = A finite set of terminal symbols denoted by a, b, c,...

With N **U** T = V and N ∩ T= φ (null set).

P = A finite set of ordered pairs ((α, β)) called the Production Rules, α

and β being the string over V^* and α involving at least one symbol from N.
S = is a special symbol called the Starting Symbol.

Example :-
Let G= (N, T, S, P); N= {S, B, C}, T= {a, b}

P= {(S → aba), (SB →b), (b→bB), (b→λ)}

This grammar is not a structure grammar because of the production rule **b →bB** because the left side of this rule containing only a terminal symbol (b) and in any production rule the left side must involve at least one non-terminal symbol.

Example :-

Let G= (N, T, S, P) where N= {S, A}, T= {a, b}

P= {(S→aAa), (A→bAb), (A→a)}

S → aAa → abAba → abbAbba → abbabba

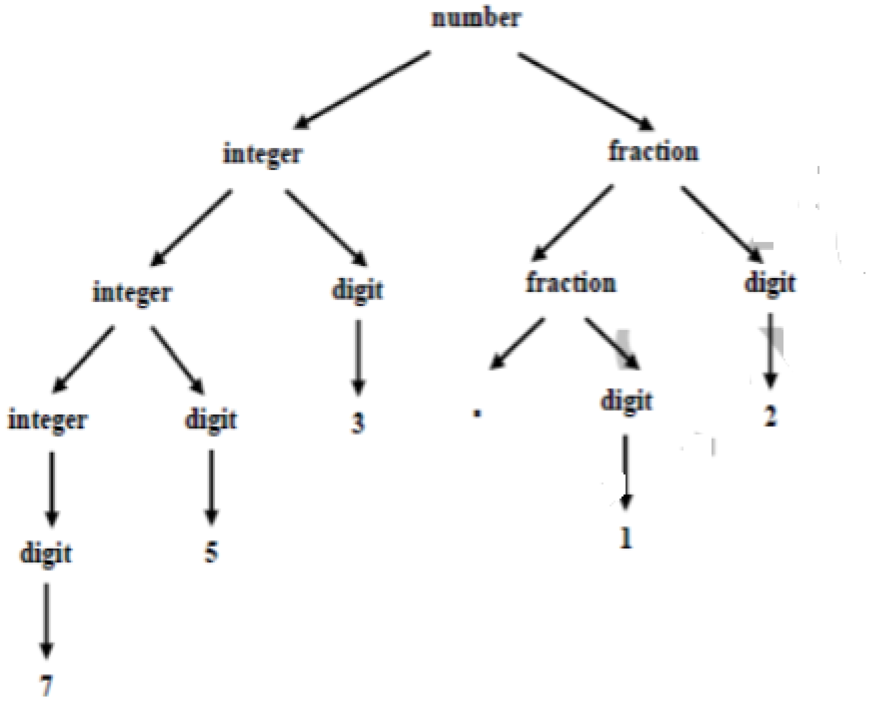
Note :-
1. The production rules can be written in another form, for the above example, the production rule is written as follows:-
P= {(S, aAa), (A, bAb), (A, a)}
2. Some times it may be that two different grammars G and Ğ generated the same language **L (G)=L(Ğ)** ∴ the grammars are said to be *equivalent*.

Example :-
G= (N,T,S,P)
N= {number, integer, fraction, digit}
T= {., 0, 1, 2, 3, ..., 9}
S=number

P={(number→integer fraction), (integer→digit), (integer→ integer digit), (fraction→.digit), (fraction→fraction digit),(digit→0), (digit→1),

(digit→2), (digit→3), (digit→4), (digit→5), (digit→6), (digit→7), (digit→8), (digit→9)}

Now we want to prove if the following number is accepted or not 753□12?



Kinds of Grammar Description :-

1. Transition Diagram.
2. BNF (Backus_ Naur form).
3. EBNF.
4. Cobol_Meta Language.
5. Syntax Equations.
6. Regular Expression (R.E.).

By using BNF the grammar can be represented as follows:-
(For the previous example)
G= (N, T, S, P)
N= {<number>, <integer> , <fraction> , <digit>}
T= {., 0, 1, 2, 3, ..., 9}

S= <number>
Production rules P will be represented as follows:
<number> ::= <integer> <fraction>
<integer> ::= <digit>|<integer> <digit>
<fraction> ::= .<digit>|<fraction> <digit>
<digit> ::= 0|1|2|3|4|5|6|7|8|9

Regular Expression (R.E.)

The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as regular grammar. The language defined by regular grammar is known as regular language. The various operations on languages are:

□ Union of two languages L and M is written as
L U M = {s | s is in L or s is in M}
□ Concatenation of two languages L and M is written as
LM = {st | s is in L and t is in M}
□ The Kleene Closure of a language L is written as
L* = Zero or more occurrence of language L.

For example, R* is R.E. denoting

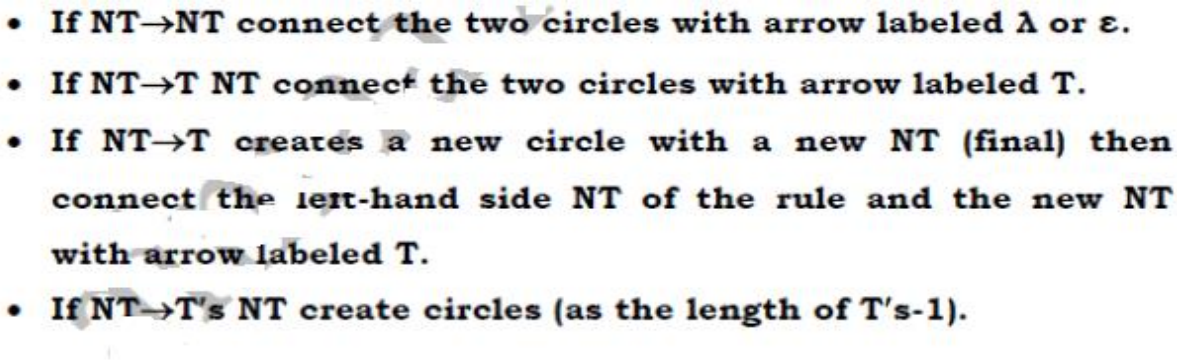
$$\{\varepsilon\} \cup L_R \cup L_R^2 \cup \ldots \cup L_R^n$$

The main components of RE are
1. ε or λ is R.E. denoting by $L^0=\{\varepsilon\}=L$
2. Any terminal symbol like a is R.E. denoting L={a}
3. [a-z] is all lower-case alphabets of English language.
4. [A-Z] is all upper-case alphabets of English language.
5. [0-9] is all natural digits used in mathematics.

Transformation of R.E. to Transition Diagram (Formal Method)

1. For each non terminal NT draw a circle.

2. Connect with arrows between any two circles with respect to the following rules:-



- **If NT→NT connect the two circles with arrow labeled Λ or ε.**
- **If NT→T NT connect the two circles with arrow labeled T.**
- **If NT→T creates a new circle with a new NT (final) then connect the left-hand side NT of the rule and the new NT with arrow labeled T.**
- **If NT→T's NT create circles (as the length of T's-1).**

Example :-

Let G= {{S, R, U},{a, b}, S, P}

P=

S → a

R → abaU

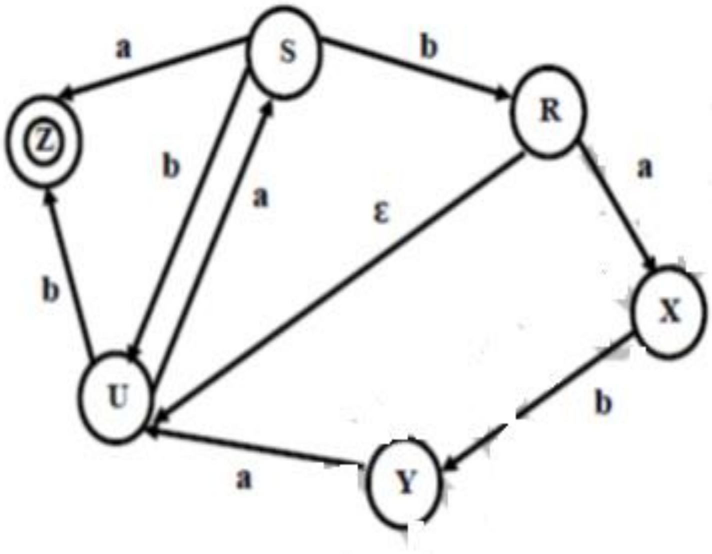
U → b

S → bU

R → U

U → aS

S → bR



Transformation of BNF to Transition Diagram (Informal Method)

1. Draw a separate transition diagram for each production rule.

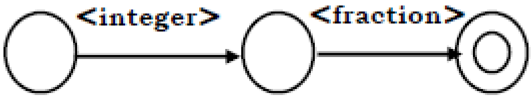
2. Substitute each non-terminal symbol by its corresponding transition diagrams.

Example :-

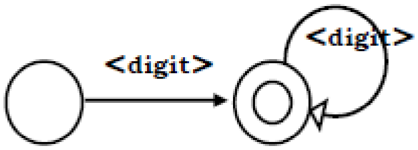
G= (N, T, S, P)

N= {<number>, <integer>, <fraction>, <digit>}
T= {., 0, 1, 2, 3, ..., 9}
S= <number>
P=
<number> ::= <integer> <fraction>
<integer> ::= <digit>|<integer> <digit>
<fraction> ::= .<digit>|<fraction> <digit>
<digit> ::= 0|1|2|3|4|5|6|7|8|9

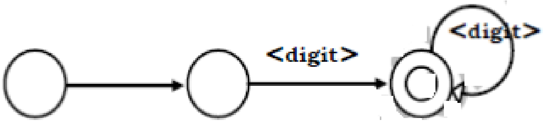
Now we take each production rule and draw to it a separate transition diagram:-
<number> ::= <integer> <fraction>



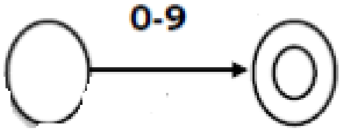
<integer> ::= <digit>|<integer> <digit>



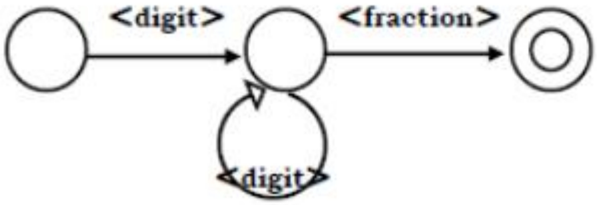
<fraction> ::= .<digit>|<fraction> <digit>

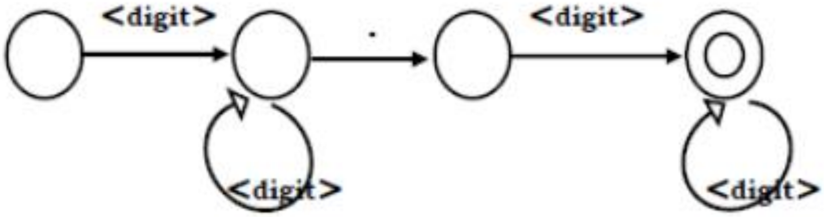


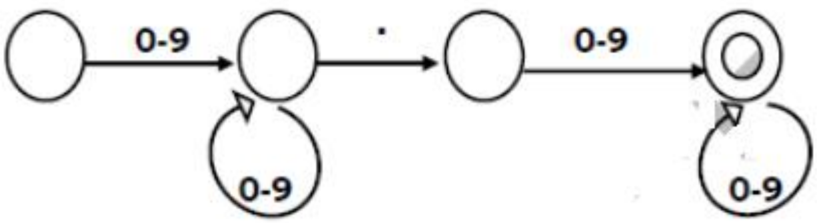
<digit> ::= 0|1|2|3|4|5|6|7|8|9



Now we must substitute each non-terminal symbol by its corresponding transition diagram.

- 
<digit>
<fraction>
<digit>

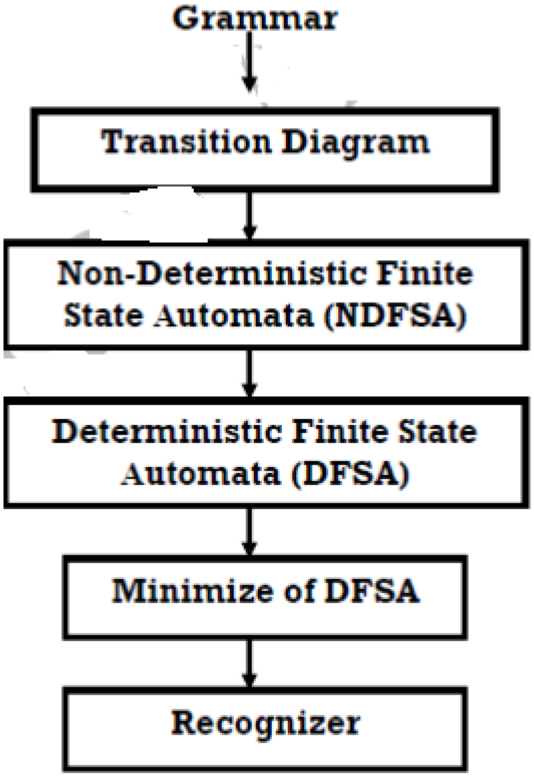
- 
<digit>
.
<digit>
<digit>
<digit>

- 
0-9
.
0-9
0-9
0-9

Lexical Analyzer Design

Lexical analysis is the first phase of a compiler. It takes modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

The main sub-phases of the Lexical analyzer phase are shown below in the following figure:-


Grammar
Transition Diagram
Non-Deterministic Finite State Automata (NDFSA)
Deterministic Finite State Automata (DFSA)
Minimize of DFSA
Recognizer

- The grammar will converted to a Transition Diagram using special algorithm.
- The converted Transition Diagram must be checked whether if it is in NDFSA form or not; if so, the grammar must converted to DFSA using algorithm which will be described in this chapter.

- The resulted grammar will be in DFSA form which must be minimized to reduce the number of nodes depending on algorithm designed for this purpose (fast searching and minimum memory storage).
- The final sub-phase in lexical analyzer phase is to recognize if the input string or statement is accepted or not depending on a specific grammar.

Finite State Automata **(FSA):-**

Is a mathematical model consists of:-

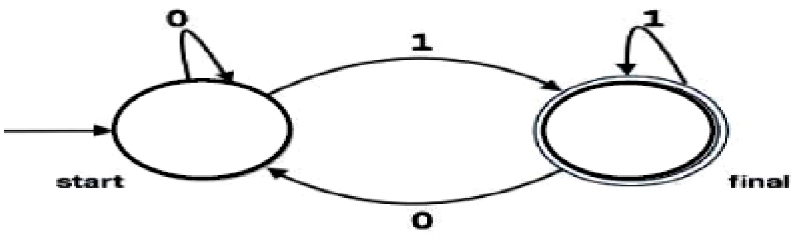
1. A set of terminal symbols
2. Transition functions
3. One-Initial state (Start state)
4. One or Set of Final states
5. Finite set of elements called states

States : States of FSA are represented by circles. State names or numbers are written inside circles.

Start state : The state from where the FSA starts, is known as the start state. Start state has an arrow pointed towards it.

Final State :- If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles,it is also called the Accepting State.

A transition :- Is denoted by an arrow connecting two states, the arrow is labeled by the symbol (possibly e). The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows points to the destination state.



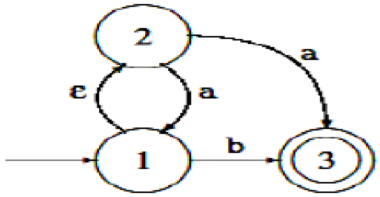
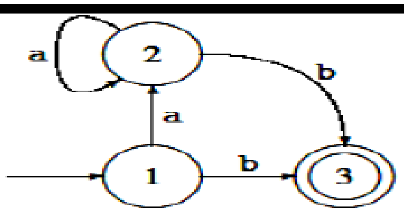
Two types of *FSA* :-

- Non-Deterministic Finite State Automata (NDFSA)
- Deterministic Finite State Automata (DFSA)

FSA is of NDFSA if one of these two conditions is satisfied:-

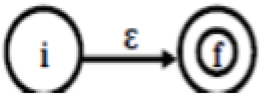
1. There are more than one transition have the same label from that state to another states.

2. There is a ε - transition.

A transition, represent FSA of type NDFSA.	A transition, represent FSA of type DFSA.
	

Formal method for converting R.E. to NDFSA :-

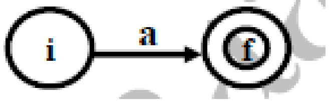
1- If we have an R.E.= ε then the NDFSA will be as follows:-



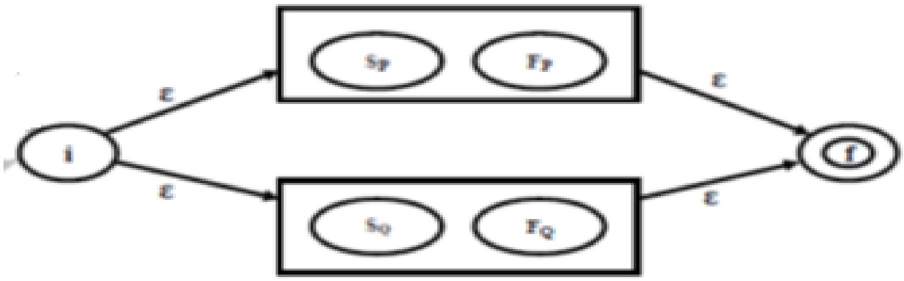
where i = initial state , f =final state

where i = initial state , f =final state

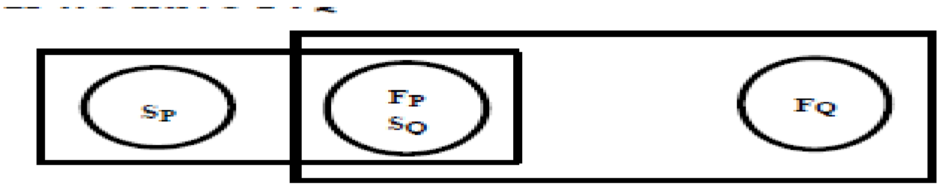
2- If we find a terminal symbol like a, then the NDFSA will be as follows:-



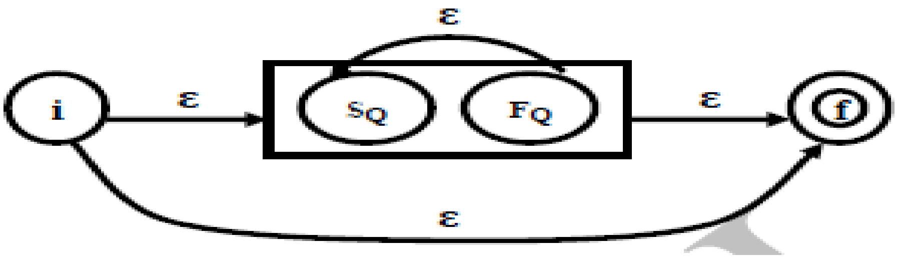
3- If we have P|Q



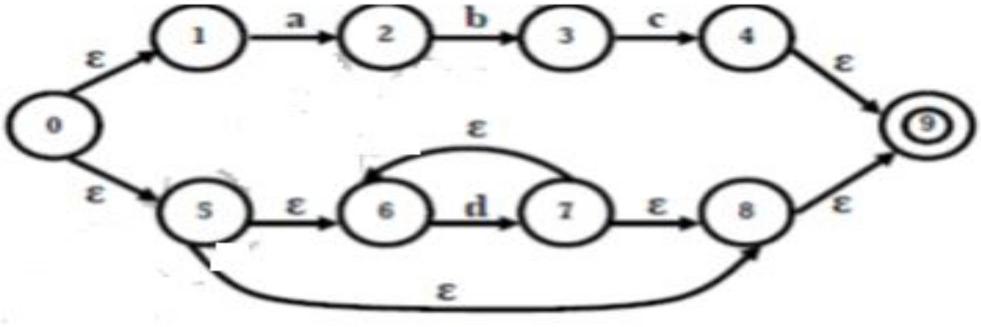
4- If we have P.Q



5- If we have Q*

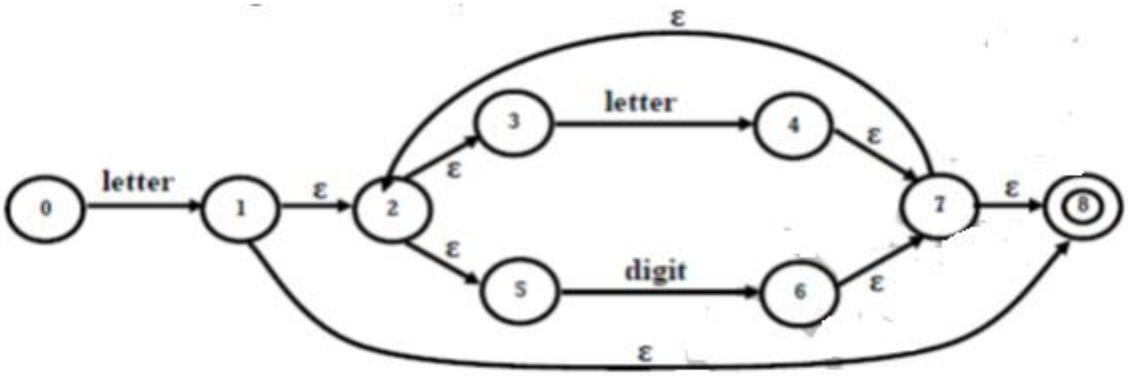


***Example* :-**
R.E.= abc|d*

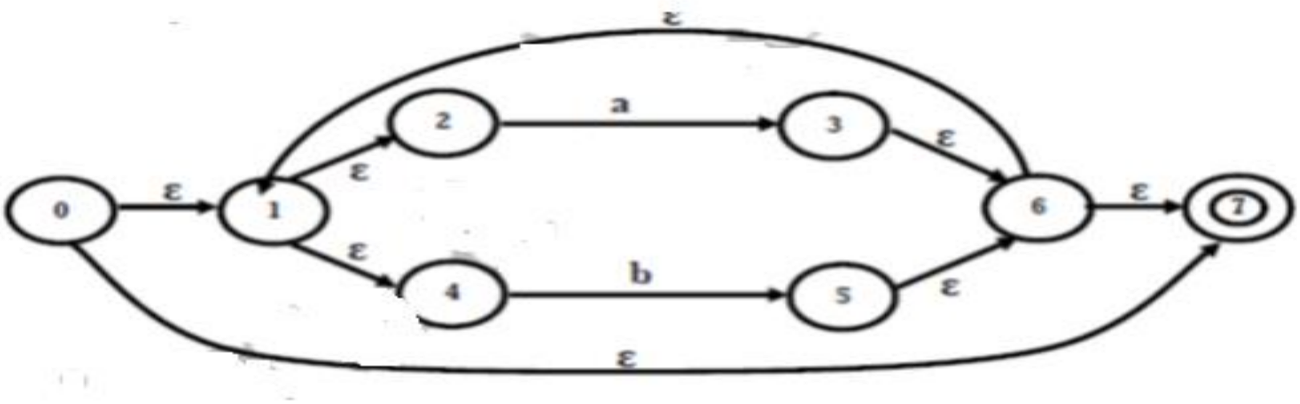


***Examples* :-**

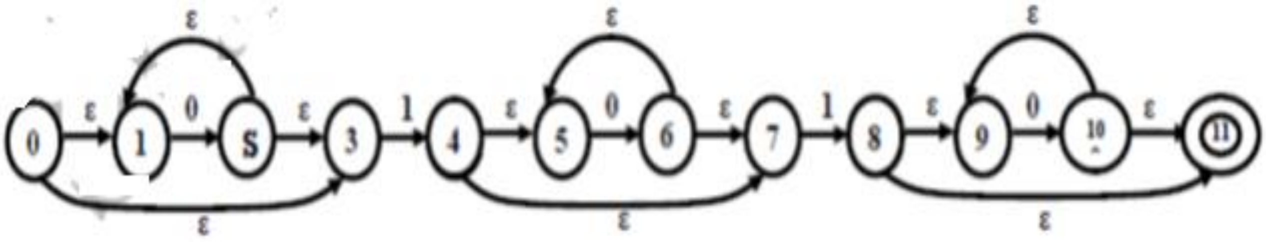
1. **RE= letter (letter | digit)***



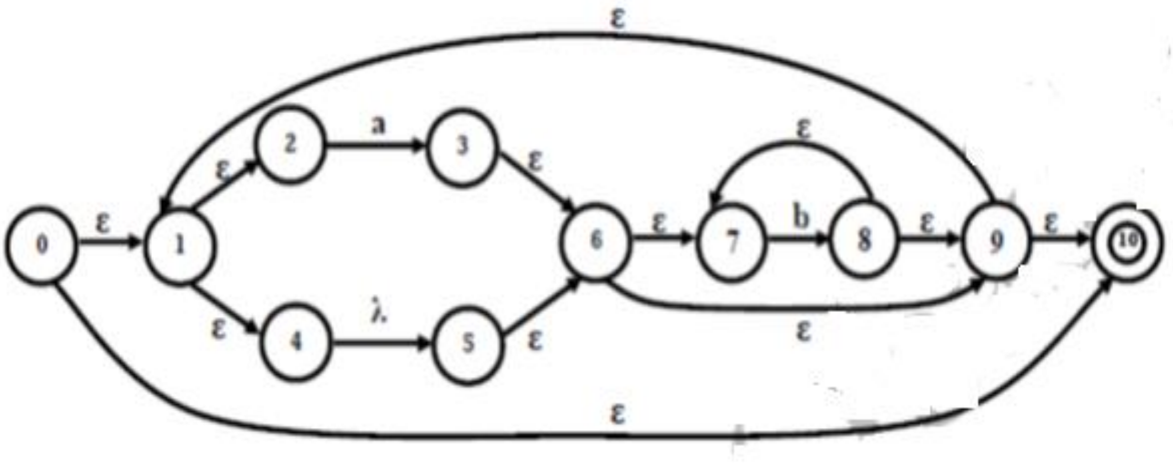
2. R.E.= (a | b) *



3. R.E.= 0*1 0*1 0*



4. R.E.= ((λ | a) b*)*



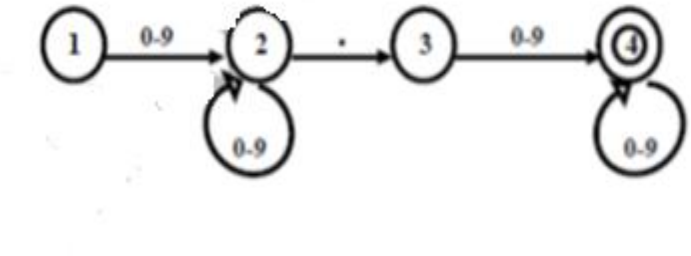
Data structure representation of FSA **:-**

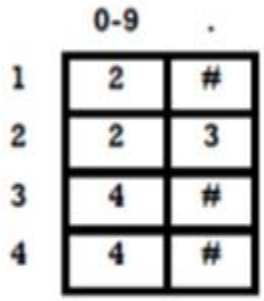
1- *Transition Matrix*

We must have a matrix with the number of its rows equal to the number of the FSA states in the diagram while the number of its columns in this matrix equal to the number of its inputs (labels).

This type of representation has a disadvantage that it contains many blank spaces, while the advantage of this type is that the indexing is fast.

For example:-



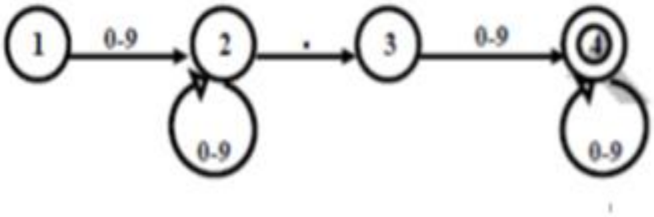


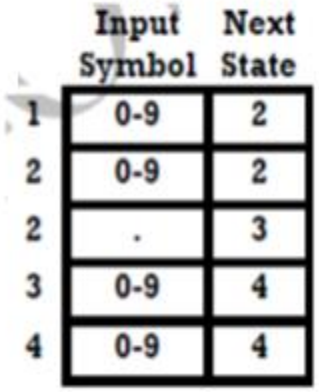
	0-9	.
1	2	#
2	2	3
3	4	#
4	4	#

- ***Graph Representation***

In this representation we have a fixed number of columns which is equal to 2 and the labels of these two columns are *Input Symbol* & *Next State* while the number of rows differs from one transition diagram to another and these rows are labeled by the number of states. The disadvantage of this representation is that it takes a long time for searching (search slow) while the advantage of this representation is that it is compact.

For the previous example:-





	Input Symbol	Next State
1	0-9	2
2	0-9	2
2	.	3
3	0-9	4
4	0-9	4

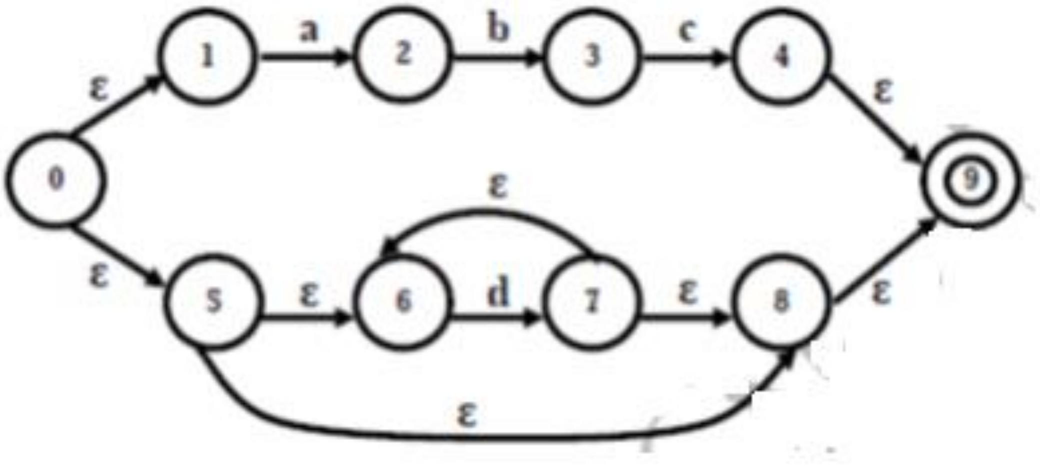
***Transformation of NDFSA to DFSA*:-**

Before we use an algorithm to convert the grammar which is NDFSA form to DFSA form, we must deal with a special function known as ε-*Closure Function*, which can be explained using the following procedure:-

***Function Ɛ-Closure* (M) :-**

```
Begin
  Push all states in M into stack;
  Initialize Ɛ-Closure (M) to M;
  While stack is not empty do
       Begin
       Pop S;
       For each state X with an edge labeled ε from S to X do
       If X is not in Ɛ-Closure (M) then
                Begin
                Push X;
                 Add X to Ɛ-Closure (M);
          End;
   End;
End;
```

***Example* :-**
R.E.= abc|d*



To compute randomly the ε-Closure for the following states:-

Ɛ-Closure ({0}) = {0, 1, 5, 6, 8, 9}

Ɛ-Closure ({1}) = {1}

Ɛ-Closure ({7, 8}) = {7, 8, 9, 6}

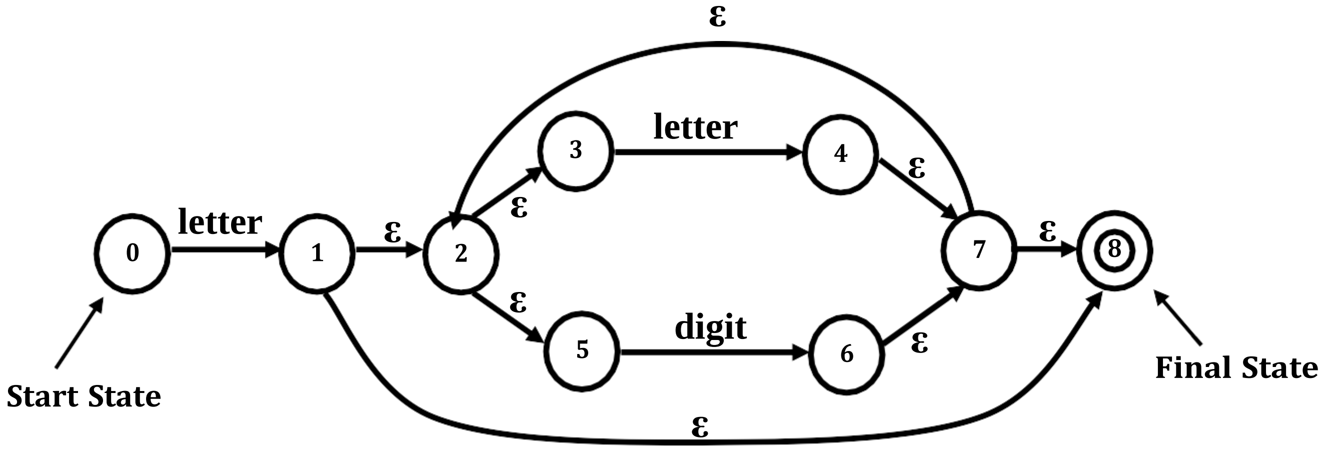
Ɛ-Closure ({2, 3, 4})={2, 3, 4, 9}

Algorithm for transforming NDFSA to DFSA:-

```
Initially let x= Ɛ-Closure ({$S_0$}) marked as the start state of DFSA, $S_0$ is the start state of NDFSA;
While there is unmarked states X = {$S_1$, $S_2$, ... ,$S_n$} of DFSA do
    Begin
        For each terminal symbol (a $\in \Sigma$) do
          Begin
            Let M be the set of states to which there is transition on a from some states $S_i$ in X ;
            Y = Ɛ-Closure ({ M });
            If Y has not yet been added to the set of states of DFSA then make Y an unmarked state of DFSA;
            Create an edge by adding a transition from X to Y labeled a if not present;
          End;
    End;
End {algorithm}
```

Examples:-

① **R.E. = Letter (letter | digit)***



Ɛ-Closure ({ 0 }) = {0}◂······ Create a new node called for example A

A → **letter ; M={1}; Ɛ-Closure ({1})={1,2,3,5,<u>8</u>} ◂······ Create a new node called for example <u>B</u> (must be a final node because of node 8).**

→ **digit ; M=∅;**

B → **letter ; M={4}; Ɛ-Closure ({4})={4,7,<u>8</u>,2,3,5} ◂······ Create a new node called for example <u>C</u> (must be a final node because of node 8).**

→ **digit ; M={6}; Ɛ-Closure ({6})={6,7,<u>8</u>,2,3,5} ◂······ Create a new node called for example <u>D</u> (must be a final node because of node 8).**

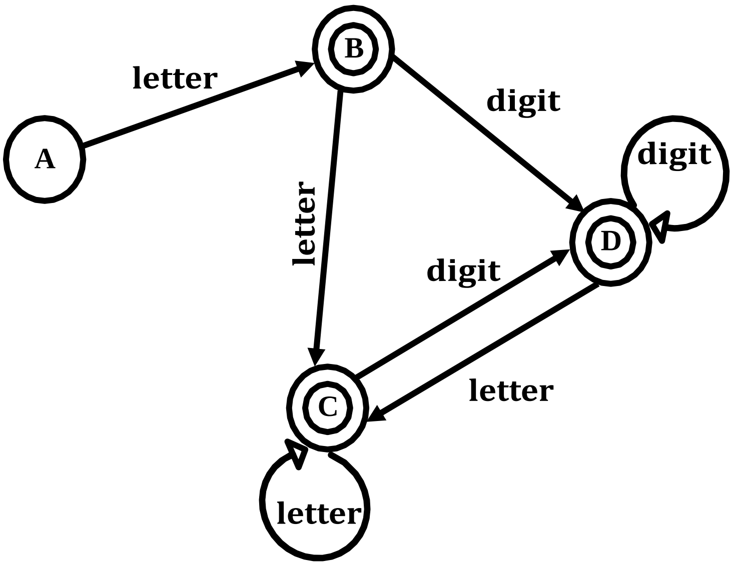
C → **letter ; M={4}; No need to create a new node because Ɛ-Closure ({4}) has been computed and by which we have node <u>C</u>.**

→ **digit ; M={6}; No need to create a new node because Ɛ-Closure ({6}) has been computed and by which we have node <u>D</u>.**

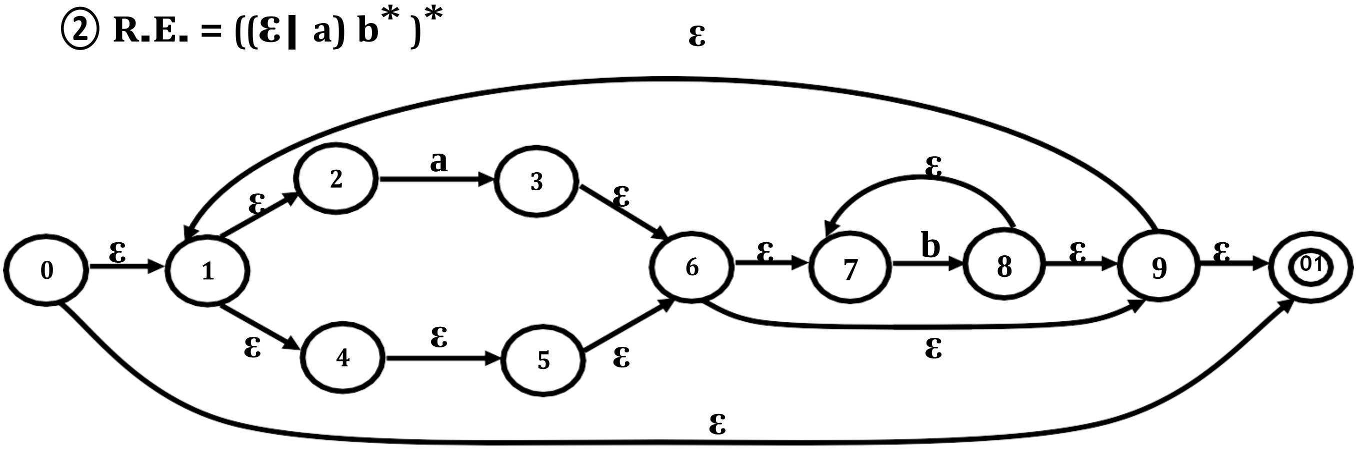
D → letter ; M={4}; No need to create a new node because Ɛ-Closure ({4}) has been computed and by which we have node C.

→ digit ; M={6}; No need to create a new node because Ɛ-Closure ({6}) has been computed and by which we have node D.

Since of no nodes will be created and all the created nodes have been manipulated, we will reach to the final step by which we have the DFSA, this step will convert all the above work into a graph as follows:-



② **R.E. = $((\varepsilon \,|\, a)\ b^{*})^{*}$**



Ɛ-Closure ({0}) = {0,1,2,4,5,6,7,9,<u>10</u>}◄······· Create a new node called for example <u>A</u>(must be a final node because of node 10).

A → a ; M={3}; Ɛ-Closure ({3})={3,6,7,9,<u>10</u>,1,2,4,5} ◄······· Create a new node called for example <u>B</u> (must be a final node because of node 10).

→ b ; M={8}; Ɛ-Closure ({8})={8,7,9,<u>10</u>,1,2,4,5,6} ◄······· Create a new node called for example <u>C</u> (must be a final node because of node 10).

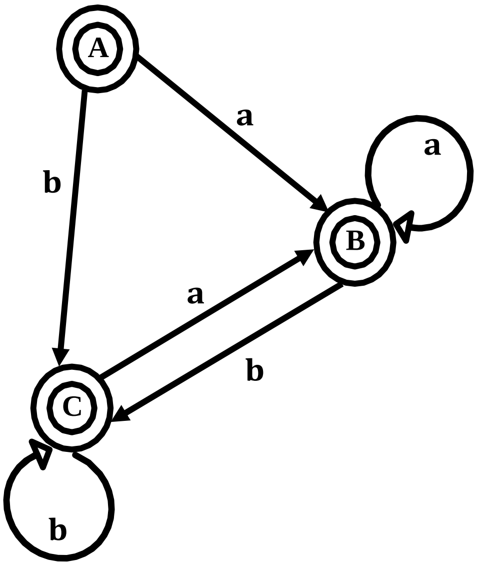
B → a ; M={3}; No need to create a new node because Ɛ-Closure ({3}) has been computed and by which we have node <u>B</u>.

→ b ; M={8}; No need to create a new node because Ɛ-Closure ({8}) has been computed and by which we have node <u>C</u>.

C → a ; M={3}; No need to create a new node because Ɛ-Closure ({3}) has been computed and by which we have node <u>B</u>.

→ b ; M={8}; No need to create a new node because Ɛ-Closure ({8}) has been computed and by which we have node <u>C</u>.

Since of no nodes will be created and all the created nodes have been manipulated, we will reach to the final step by which we have the DFSA, this step will convert all the above work into a graph as follows:-



③ R.E. = (a|b)*abb

***Minimizing of DFSA*:-**

The purposes of minimization are:-

1. **Efficiency.**
2. **Optimal DFSA.**

***Algorithm*:-**

1. **Construct an initial partition Л of the set of states with two groups: the accepting states F and the non-accepting states S-F; where S is the set of all states of DFSA.**
2. **for each group G of Л do**

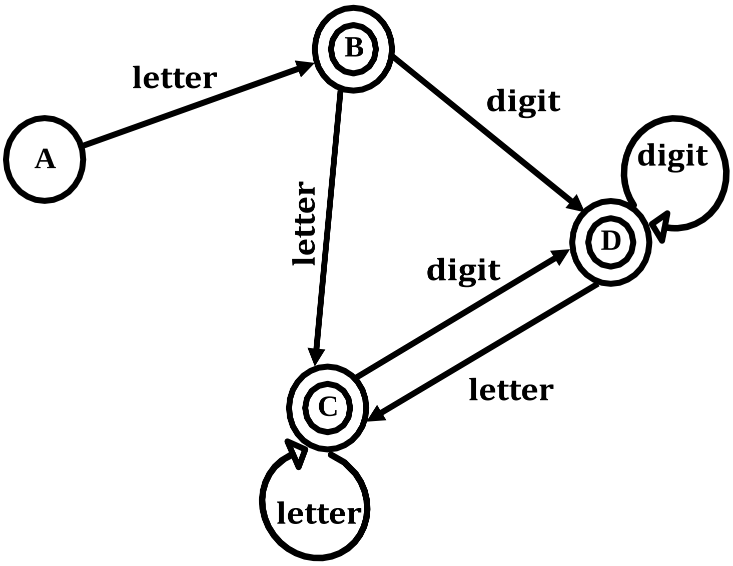
   Begin

   partition G into subgroups such that two states S and T of G are in the same subgroup if and only if for all input symbols *a*, and states' S and T have transitions on *a* to states in the same group of Л, replace G in Л$_{\text{new}}$ by the set of all subgroups formed .

   End
3. **If Л$_{\text{new}}$ = Л, let Л$_{\text{final}}$ = Л and continue with step (4), otherwise repeat step (2) with Л := Л$_{\text{new}}$**
4. **Choose one state in each group of the partition Л$_{\text{final}}$ as the representative for that group.**

Example **:-**

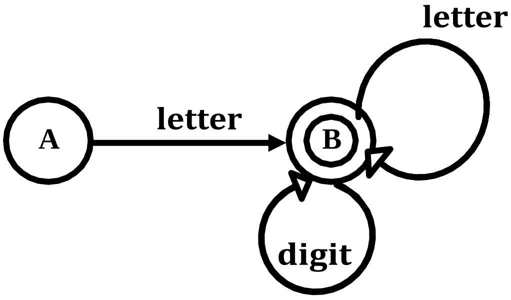
The DFSA for the R.E. = Letter (letter | digit)* is as follows:-



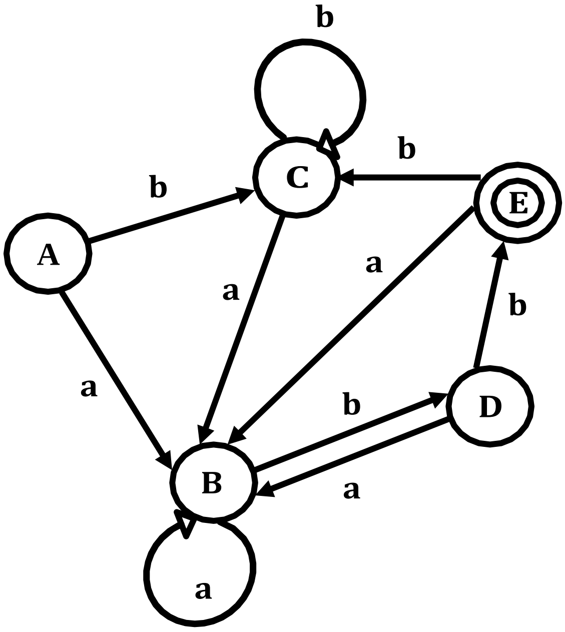
Group$_1$= {A} which represents the set of not final nodes while Group$_2$ = {B,C,D} which represents the set of final nodes.

Always minimization acts on the nodes of the same type (on the nodes of one group)

After applying the previous algorithm, the minimization figure will be as follows:-



Another example **:-**

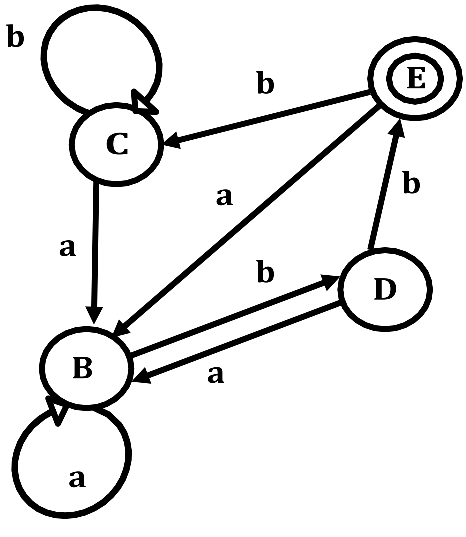


Group$_1$= {A,B,C,D} which represents the set of not final nodes while

Group$_2$ = {E} which represents the set of final nodes.

Always minimization acts on the nodes of the same type (on the nodes of one group)

After applying the previous algorithm, the minimization figure will be as follows:-



FSA Accepter (Recognizer):-

This will represents the final sub-phase for the lexical analyzer ,by using a specific algorithm shown below we can specify the input string or statement is accepted or not depending on a given grammar.

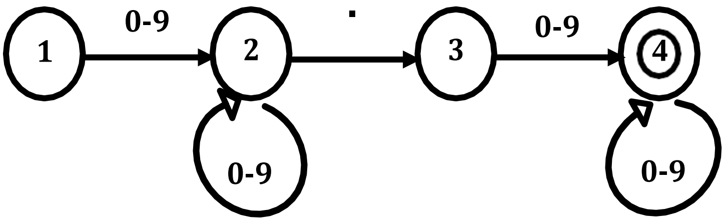
Never can apply the algorithm unless the grammar will be in *minimized form*.

First, a transition matrix must be created for a given FSA, then doing a table having two columns, the first represents the number of states while the other represents the symbols for a given input string.

Algorithm :-

```
Begin
   State = Start State of the FSA;
   Symbol = First Input Symbol;
     If Matrix [State, Symbol] ≠ Error Indication then
        Begin
          State = Matrix [State, Symbol];
          Symbol = Next Input Symbol;
        End
     Else Input is not accepted
   If State is a Final State of FSA then Input is accepted
     Else Input is not accepted
End;
```

Example **:- Having the following FSA representation shown below:-**



Depending on the above representation, for <u>1.3$</u> and <u>37$</u> ,you asked to recognize which one is accepted and which one is not accepted?

***Solution*:-**

The *Transition Matrix* for the above FSA:-

	0-9	.
1	2	#
2	2	3
3	4	#
4	4	#

For the String = <u>1.3 $</u>

State	Input symbol
1	1
2	.
3	3
4	$

It is accepted because state number 4 is a final State

For the String = <u>37 $</u>

State	Input symbol
1	3
2	7
2	$

It is not accepted because state number 2 is not a final state and the expression is finished

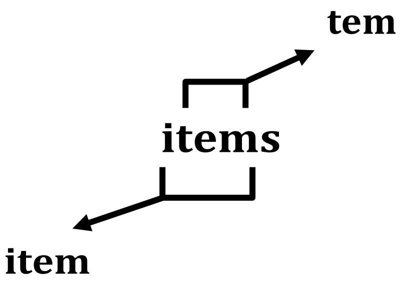
This algorithm was slow and overlapping token, so a new algorithm can be used to recognize the overlapping token.

***For example*:-**

Suppose that we have this language:

{"bit" , "byte" , "item" , "tem"}

Now if we take the word *items*, we will find two words overlapping with each other, these words are: *item* and *tem*

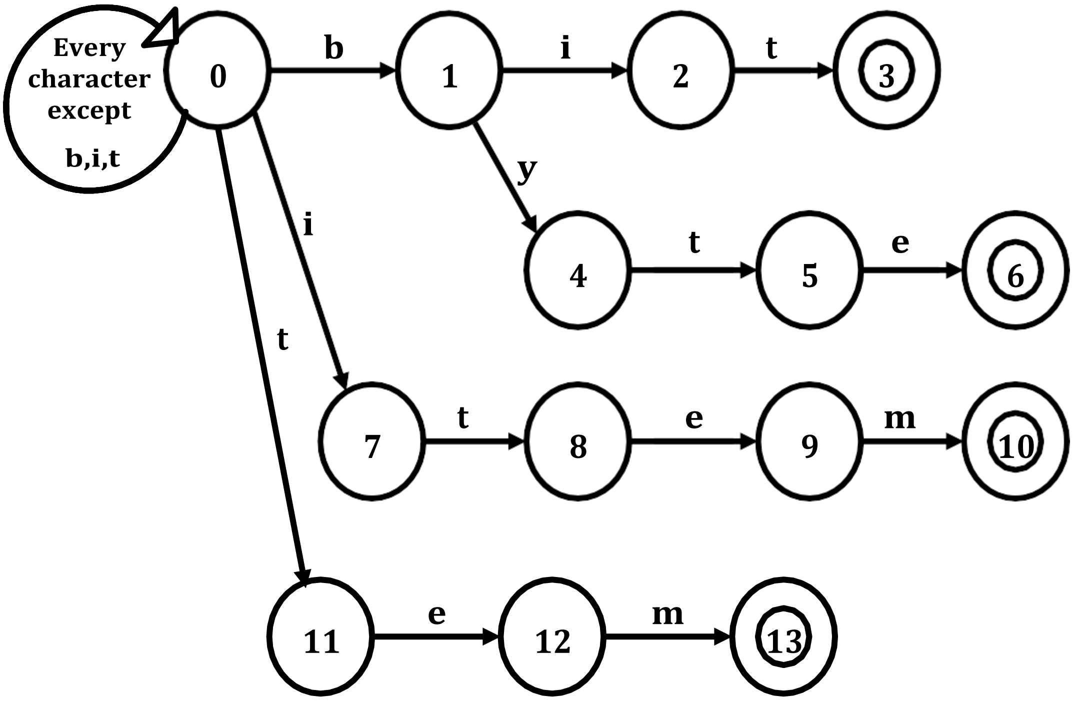


The new algorithm is known as <u>AHO Algorithm</u> and depends on the following steps:-

(For the above example)

Step 1:- Constructing Tree-Structured DFSA.

(Always the input for the first node is all letters except the letters that are outputted from it).



Step 2:- Determine fall back function f (Q) =R which is calculated as follows:-

- **Find largest route α which lead to Q <u>from a state that is not the start state</u>.**
- **Find the route α but this time <u>from the start state and finished in R</u>.**
- **F(Q)=R.**

Q	0	1	2	3	4	5	6	7	8	9	10	11	12	13
F(Q)	0	0	7	8	0	11	12	0	11	12	13	0	0	0

Step 3:- Construct the Matrix Representation for the DFSA, the number of rows in it equal to the number ob nodes found in DFSA, while the number of columns equal to the number of characters that form the input language.

	b	i	t	m	y	e
0	1	7	11	0	0	0
1	#	2	#	#	4	#
2	#	#	3	#	#	#
3	#	#	#	#	#	#
4	#	#	5	#	#	#
5	#	#	#	#	#	6
6	#	#	#	#	#	#
7	#	#	8	#	#	#
8	#	#	#	#	#	9
9	#	#	#	10	#	#
10	#	#	#	#	#	#
11	#	#	#	#	#	12
12	#	#	#	13	#	#
13	#	#	#	#	#	#

Step 4:- Apply the steps of AHO Algorithm which is shown below:-

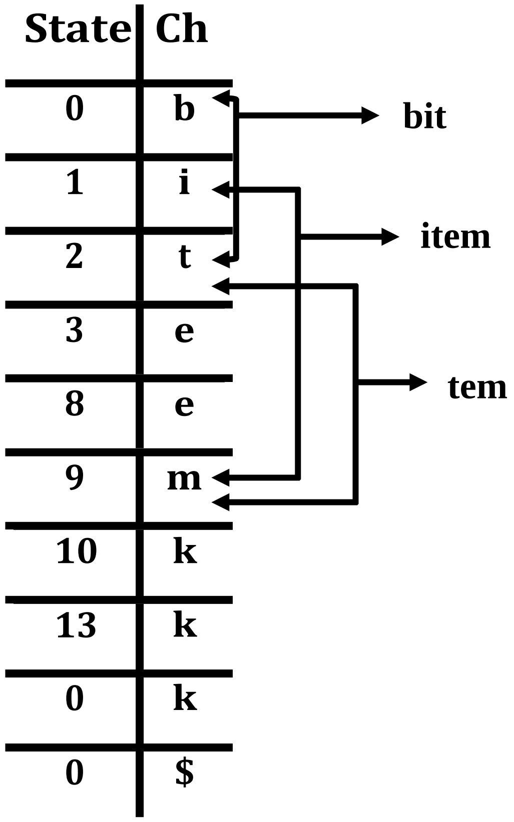
Algorithm **:-**

```
Begin
    State = Start State;
    Ch = First Character of Input;
    While input symbols are not already exhausted do
      If Matrix [State, Ch] ≠ error indication then
      Begin
          State = Matrix [State, Ch];
          Ch = next Character;
      End
      Else begin
    If State is a Final State then Signal;
    If State = 0  then   Ch= Next Character & State = Same State
       Else   State= f (State) & Ch=Same Character
      End;
End;
```

Example **:-**

Input String = <u>bitemk$</u> for the same language {"bit" , "byte" , "item" , "tem"}

After constructing Tree-Structured DFSA, and create a Transition Matrix for it with computing the value of the fall back function



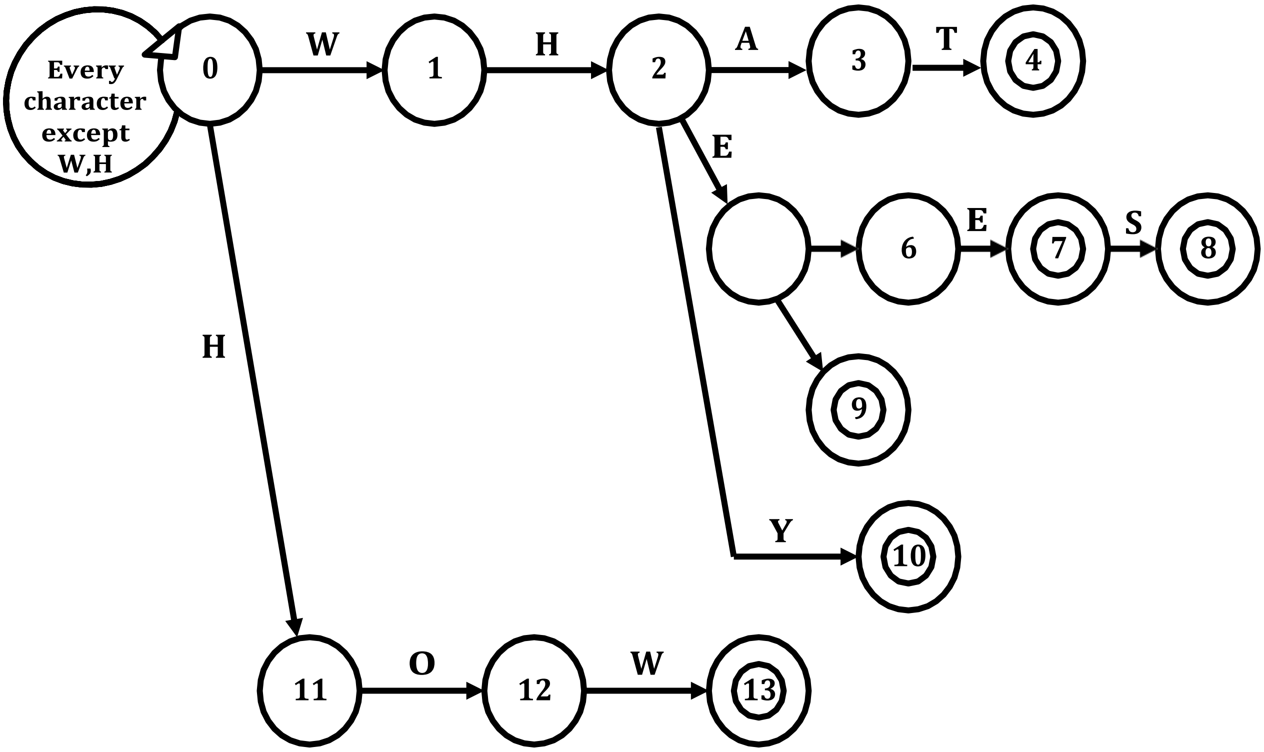
State	Ch
0	b
1	i
2	t
3	e
8	e
9	m
10	k
13	k
0	k
0	$

Example **:-**

If you have the following language:-

{"WHAT", "WHERE"," WHEN"," WHERES","HOW"," WHY"} and you asked to apply AHO algorithm on it to specify the words that are overlapped with each other in this string:- (WHYOWNSE$)

Step 1:- Constructing Tree-Structured DFSA.



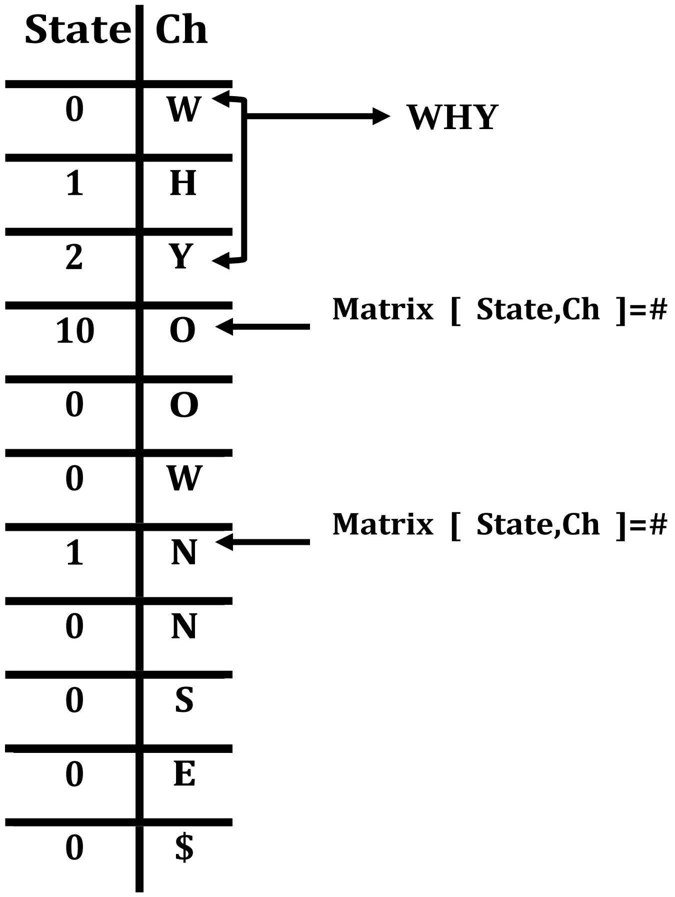
Step 2:- Compute fall back function f (Q) as follows:-

Q	0	1	2	3	4	5	6	7	8	9	10	11	12	13
F(Q)	0	0	11	0	0	0	0	0	0	0	0	0	0	1

Step 3:- Construct the Matrix Representation for the DFSA, the number of rows in it equal to the number ob nodes found in DFSA, while the number of columns equal to the number of characters that form the input language.

	W	H	A	E	Y	N	O	S	R
0	1	11	0	0	0	0	0	0	0
1	#	2	#	#	#	#	#	#	#
2	#	#	3	5	10	#	#	#	#
3	#	#	#	#	#	#	#	#	#
4	#	#	#	#	#	#	#	#	#
5	#	#	#	#	#	9	#	#	6
6	#	#	#	7	#	#	#	#	#
7	#	#	#	#	#	#	#	8	#
8	#	#	#	#	#	#	#	#	#
9	#	#	#	#	#	#	#	#	#
10	#	#	#	#	#	#	#	#	#
11	#	#	#	#	#	#	12	#	#
12	13	#	#	#	#	#	#	#	#
13	#	#	#	#	#	#	#	#	#

Step 4:- Apply the steps of <u>AHO Algorithm</u> on the string :- (<u>WHYOWNSE$</u>).



State	Ch
0	W
1	H
2	Y
10	O
0	O
0	W
1	N
0	N
0	S
0	E
0	$

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 <u>Parse Tree</u> or <u>Syntax Tree</u>.

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.

<u>Example (1):-</u>

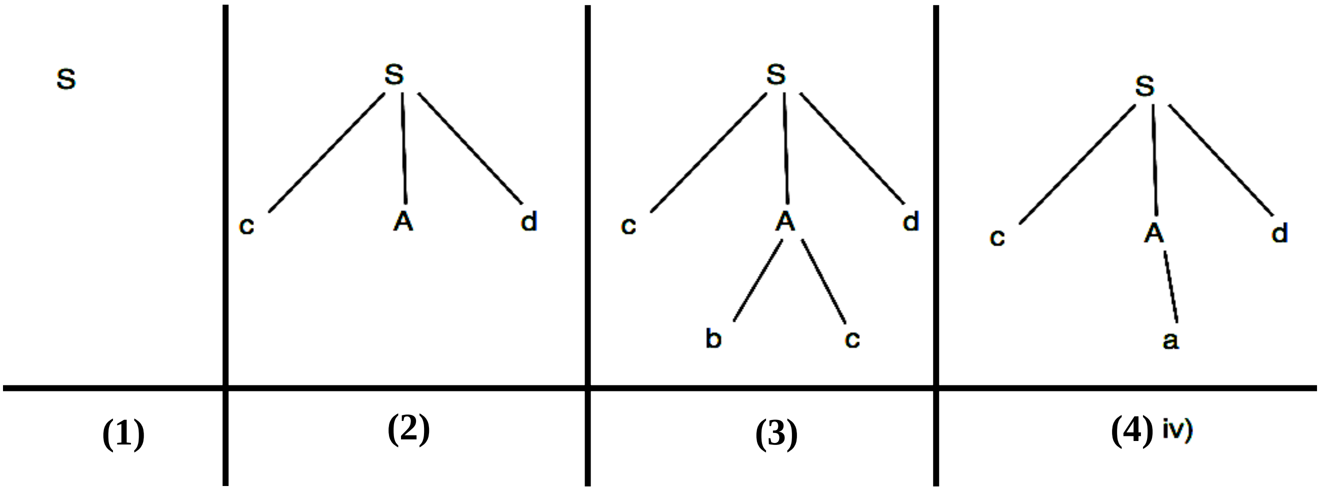
Suppose Production rules for the Grammar of a language are: S ⟶ cAd

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.

<u>Example (2):-</u>

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

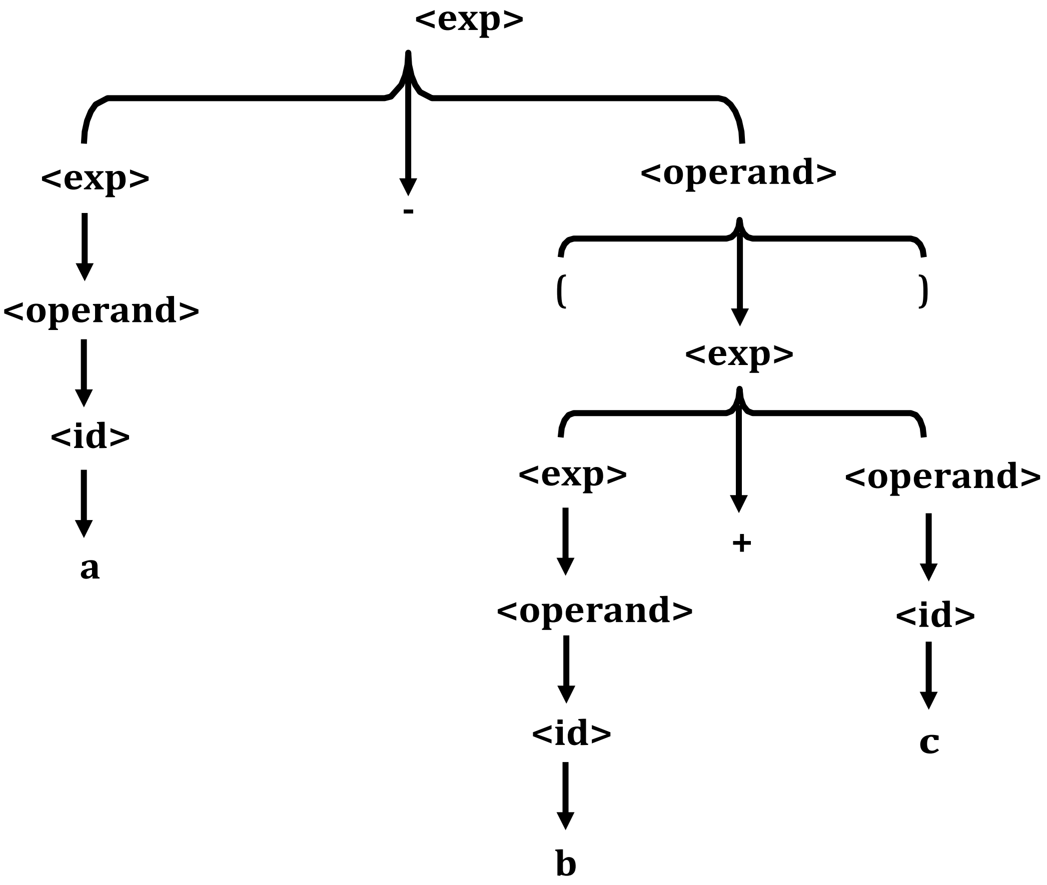
P=

<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 (*<u>Derivation Method</u>*).

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* :-**

$$S\rightarrow E\rightarrow T\times F\rightarrow F\times F\rightarrow P\times F\rightarrow a\times F\rightarrow a\times P\rightarrow a\times(S)\rightarrow a\times(E)\rightarrow a\times(E+T)\rightarrow$$

$$a\times(T+T)\rightarrow a\times(F+T)\rightarrow a\times(P+T)\rightarrow a\times(b+T)\rightarrow a\times(b+F)\rightarrow a\times(b+P)$$

$$\rightarrow a\times(b+a) \qquad \therefore a\times(b+a) \text{ is accepted}$$

Rightmost derivation :-

$$S \rightarrow E \rightarrow T\times F \rightarrow T\times P \rightarrow T\times(S) \rightarrow T\times(E) \rightarrow T\times(E+T) \rightarrow T\times(E+F) \rightarrow T\times(E+P) \rightarrow$$

$$T\times(E+a) \rightarrow T\times(T+a) \rightarrow T\times(F+a) \rightarrow T\times(P+a) \rightarrow T\times(b+a) \rightarrow F\times(b+a) \rightarrow$$

$$P\times(b+a) \rightarrow a\times(b+a) \qquad \therefore a\times(b+a) \text{ is accepted}$$

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.

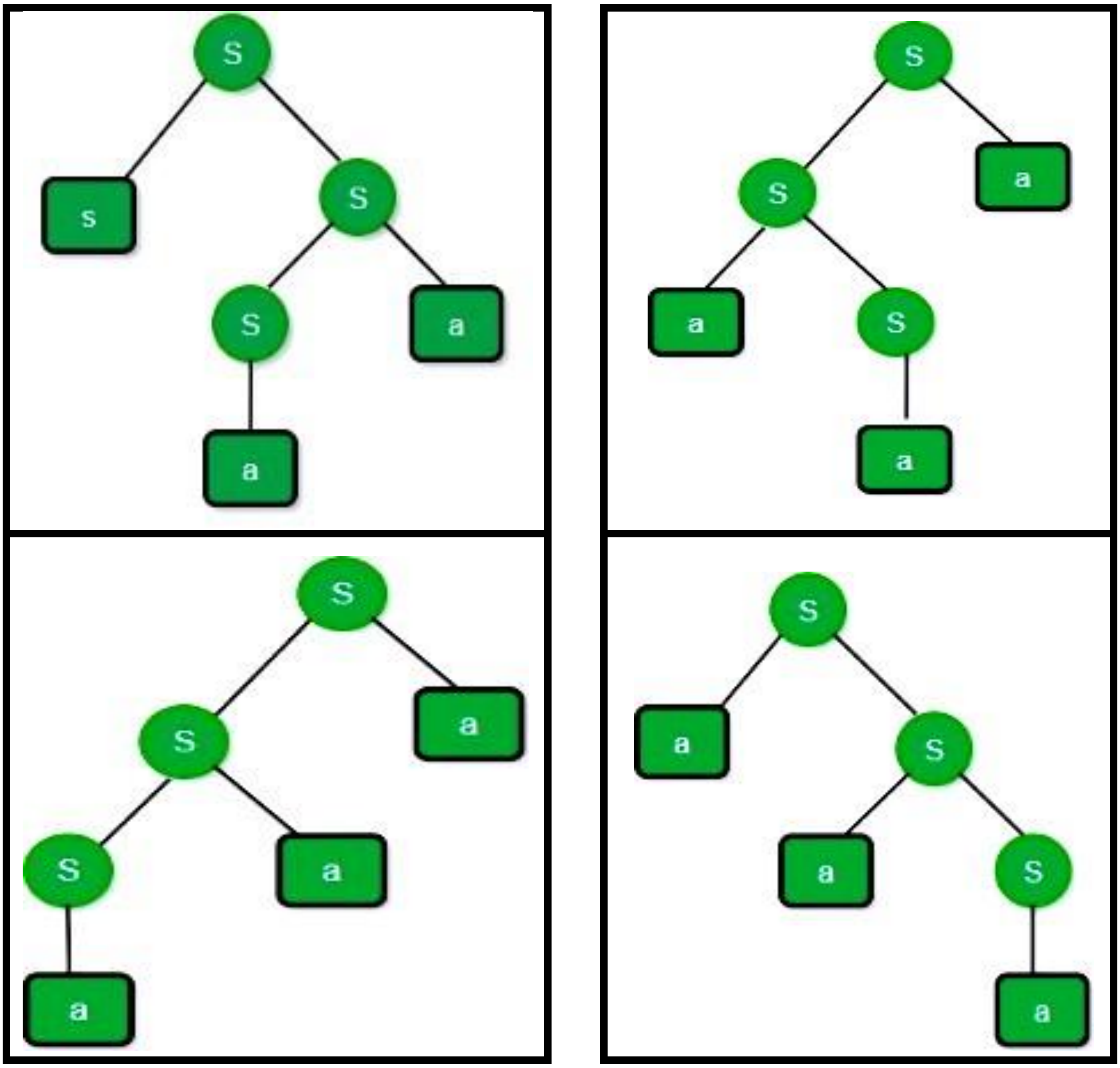
Ambiguity

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

Example:-
consider a grammar

$S \longrightarrow aS \mid Sa \mid a$

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


S
s
S
S
a
a
S
S
a
a
S
a
S
S
a
S
a
a
S
a
S
a
S
a

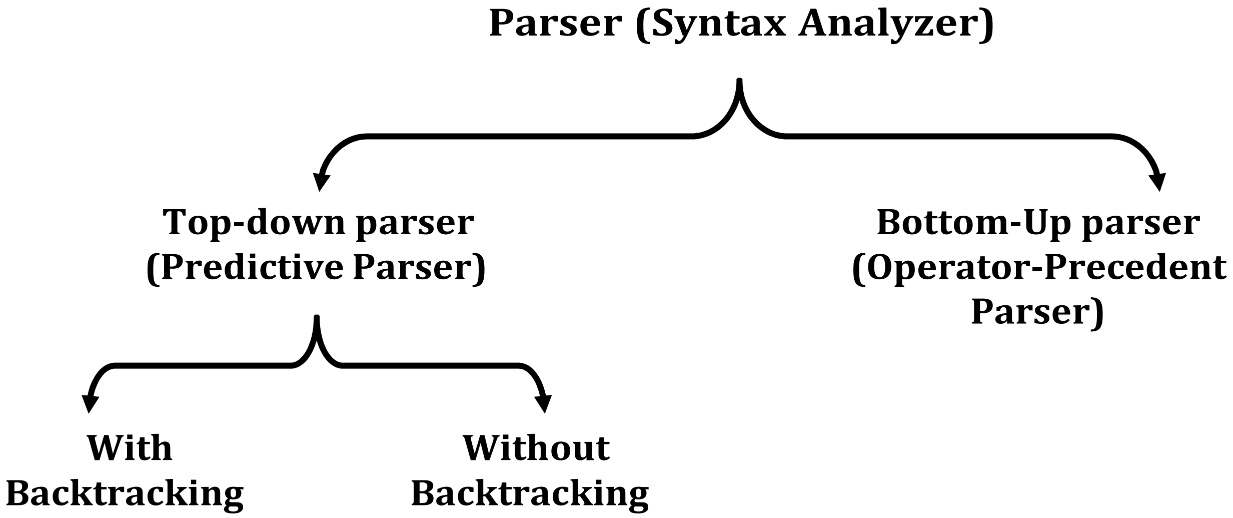
Parser Techniques

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:-


Parser (Syntax Analyzer)
Top-down parser (Predictive Parser)
Bottom-Up parser (Operator-Precedent Parser)
With Backtracking
Without Backtracking

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)

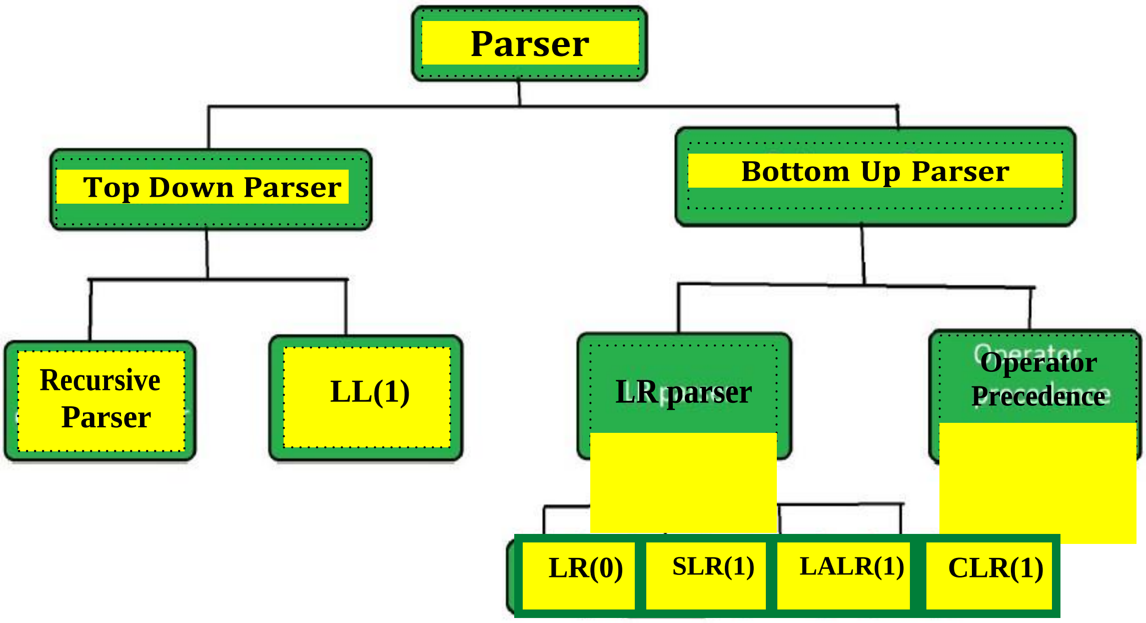
(b) SLR(1)

(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 ofany production.



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

1- Remove left recursion, because ambiguous not allowed inLL(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 **ألغاء تكرار العنصر في أقصى يسار الطرف الأيسر**

$\underline{E} \longrightarrow \underline{E}+A$

<u>Left Recursion Elimination</u> :-

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

$S \longrightarrow Sa \mid B$

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:-

$$A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_m \mid \mathcal{B}_1 \mid \mathcal{B}_2 \mid \ldots \mid \mathcal{B}_n$$

Where no $\mathcal{B}$i begins with an A. The main rule for removing the immediate backtracking is by generating two rules as follows:-

A $\longrightarrow \mathcal{B}_1\text{Á} \mid \mathcal{B}_2\text{Á} \mid \dots \mid \mathcal{B}_n\text{Á}$ ***(the first one depends on the part of the previous rule exactly on the part that not begins with A)***

Á $\longrightarrow \alpha_1\text{Á} \mid \alpha_2\text{Á} \mid \alpha_3\text{Á} \mid \dots \mid \alpha_m\text{Á} \mid \varepsilon$ ***(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| Ɛ

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´ | Ɛ

term ⟶factor term´

term´ ⟶ and factor term´ | Ɛ

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 $A_i \longrightarrow A_J \alpha$  by the production
        $A_i \longrightarrow \partial_1 \alpha / \partial_2 \alpha / \partial_3 \alpha /.../ \partial_k \alpha$;
Where
$A_J \longrightarrow \partial_1/\partial_2/\partial_3/.../\partial_k$ are the current $A_J$ productions;
     End;
Eliminate the immediate left recursion among the $A_i$ productions;
End;{of algorithm}
```

Example **(1):-**

$B \longrightarrow A\ c/d$

$A \longrightarrow Br/x$

***Solution*:-**

A_1=B A_2=A

$A_1 \longrightarrow A_2\ c/d$

$A_2 \longrightarrow A_1r/\ x$

Replace:- $A_i \rightarrow A_J \alpha$

By:- $A_i \rightarrow \partial_1 \alpha / \partial_2 \alpha / \partial_3 \alpha /.../ \partial_k \alpha$

Using:- $A_J \rightarrow \partial_1/\partial_2/\partial_3/.../\partial_k$

$A_2 \longrightarrow A_1r \ \therefore\ \alpha = r$

……………….

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

$A_1 \rightarrow \partial_1 / \partial_2$

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

I = 2	J = 1	α = r	$\partial_1 = A_2c$	$\partial_2 = d$

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

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

....

$$\left.\begin{array}{l} A_1 \rightarrow A_2c/d \\ A_2 \rightarrow A_2c\ r / d\ r / x \end{array}\right\} \longrightarrow$$

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

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

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

$\acute{A} \rightarrow \alpha_1\acute{A} / \alpha_2\acute{A} / \alpha_3\acute{A} / \ldots / \alpha_m\acute{A} / \varepsilon$

$B \rightarrow Ac/d$

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

The result will be:-

$B \rightarrow Ac/d$

$A \rightarrow d\ r\ \acute{A} / x\ \acute{A}$

$\acute{A} \rightarrow c\ r\ \acute{A} / \varepsilon$

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, α, $\partial 1$, $\partial 2$, $\partial 3$, ...

- **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.**
- **($\partial 1$, $\partial 2$, $\partial 3$, ...) 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 ($\partial 1$, $\partial 2$, $\partial 3$, ...), 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

$S \rightarrow A\ b \mid b$

$A \rightarrow Ac \mid \underline{A\ b\ d \mid b\ d} \mid e$

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

....

$S \rightarrow A\ b \mid b$

$A \rightarrow b\ d\ A' \mid e\ A'$

$A' \rightarrow c\ A' \mid b\ d\ A' \mid \varepsilon$

Now in this step, eliminate the immediate left recursion

....

Example **(2):-**

$B \rightarrow A\ c \mid d$ rule no.1

$A \rightarrow B\ r \mid x$ rule no. 2

$i=2 \quad j=1 \quad \partial 1 = A\ c \quad \partial 2 = d \quad \alpha = r$

$B \rightarrow A\ c \mid d$

$A \rightarrow (A\ c \mid d)\ r \mid x$

....

$B \rightarrow A\ c \mid d$

$A \rightarrow A\ c\ r \mid d\ r \mid x$

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

....

$B \rightarrow A\ c \mid d$

$A \rightarrow d\ r\ A' \mid x\ A'$

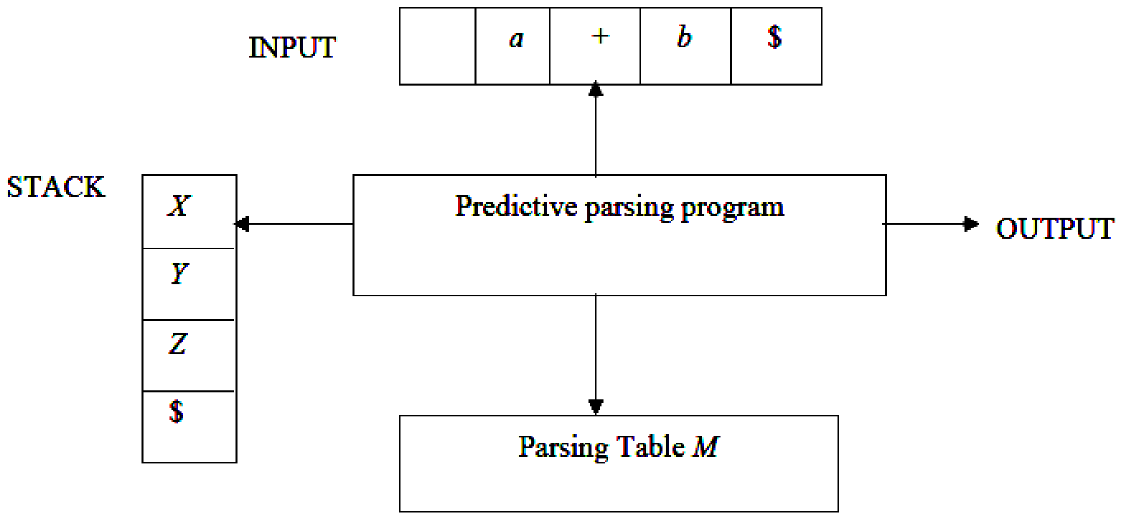
$A' \rightarrow c\ r\ A' \mid \varepsilon$

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 *a* 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

Production Rules of Grammar	**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 ter ls 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 $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.**
2. **For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a].**
3. **If ε is in FIRST(α), add $A \rightarrow \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and \$ is in FOLLOW(A) , add $A \rightarrow \alpha$ 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] = $X \rightarrow Y_1 Y_2 \ldots Y_k$ 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 بـالمقلوب ويبقـى 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				

FOLLOW Set in Syntax Analysis

Follow (X) to be the set of ter ls 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 $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.

2- For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a].

3- If ε is in FIRST(α), add $A \rightarrow \alpha$ to M[A, b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and \$ is in FOLLOW(A) , add A $\rightarrow \alpha$ 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  $X \rightarrow Y_1 Y_2 \dots Y_k$
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 بـالمقلوب 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 قبـل البـدء بعمليـة 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 *<u>not (true or false) $</u>*

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				

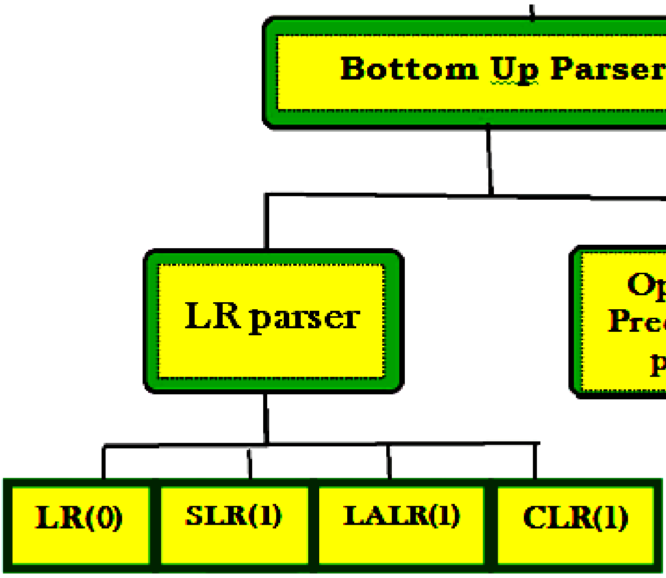
Compilers

University of Baghdad
College of Education for Pure Science
Ibn-AL-Haithem/ Dep. Of Computer Science

Asst.Prof. Shaimaa Al-Obaidy
2021-2022
Third Stage

Chapter Three

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).

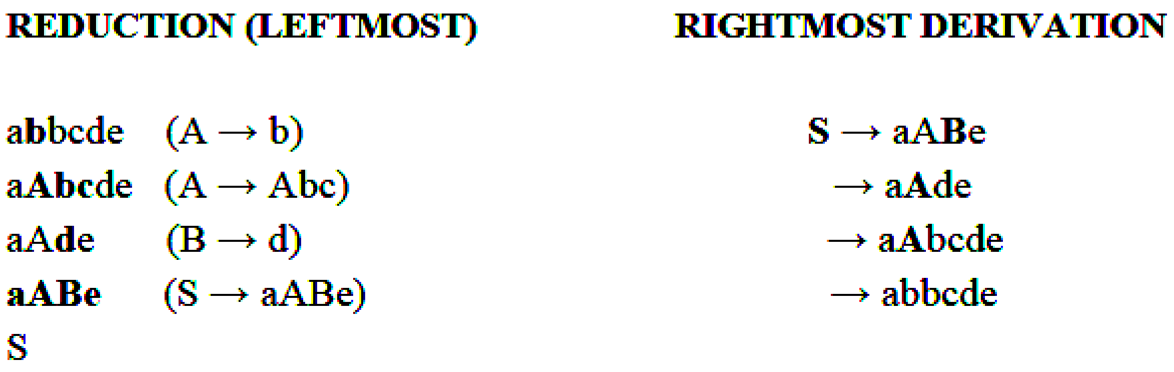
Example: **Consider the grammar:**

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 $.
- ❖ القيمة الابتدائية للعمود الثاني (Input) هي الجملة المطلوب إعرابها.
- ❖ القيمة الابتدائية للعمود الثالث والأخير تكون Shift وتمثل عملية Push للعنصر الموجود في أقصى يسار العمود الثاني ودفع العنصر في Stack.
- ❖ لابد من تطبيق Right Most Derivation على القواعد المعطاة.
- ❖ بعد الخطوة السابقة مباشرة وبالاعتماد عليها يتم تحديد ما يسمى بـ (Handle) والتي سوف يعتمد عليها قيم العمود الثالث (Action).
- ❖ اشتقاق القواعد باستخدام (Tree).
- ❖ أول مرحلة تمثل حالة إضافة العنصر الموجود في أقصى يسار الجملة المطلوب إعرابها وإضافته إلى (Top of Stack).
- ❖ ملاحظة إذا كان العنصر الذي تم إضافته إلى(Top of Stack) في الخطوة السابقة هل هو (Handle) أم لا، إذا كان (Handle) فيتم إرجاع العنصر إلى أصله وإذا لم يكن (Handle) فيتم إضافته إلى (Top of Stack).
- ❖ نستمر بالخطوات السابقة الى ان تكون قيمة الحقل الأول (Stack=<u>$Start Symbol</u>).

***Example* ①:-**

S → S×S / S+S / id **Input = <u>id×id+id$</u>**

<u>Sol.</u>

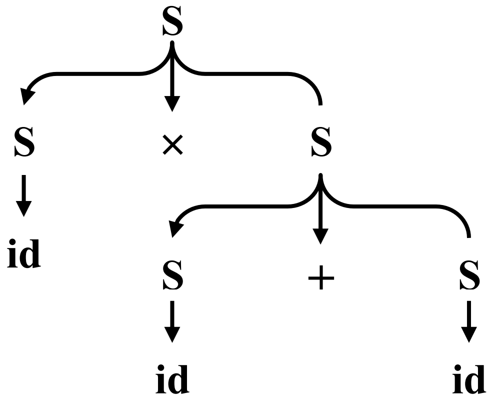
① ***<u>Derive this grammar using right most derivation</u>***

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

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

$$S \rightarrow \underline{S\times S} \rightarrow S\times \underline{S+S} \rightarrow S\times S+ \underline{id} \rightarrow S\times \underline{id} +id \rightarrow \underline{id} \times id+id$$

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 = $\underline{\text{-(id×(id-id) / id)\$}}$

Solution :-

E → $\underline{\text{-T}}$

→ -$\underline{\text{F}}$

→ -$\underline{\text{(E)}}$

→ -($\underline{\text{T}}$)

→-($\underline{\text{T/F}}$)

→-(T/ $\underline{\text{id}}$)

→-($\underline{\text{T×F}}$ / id)

→-(T× $\underline{\text{(E)}}$ /id)

→ -(T×($\underline{\text{E-T}}$) /id)

→ -(T×(E - $\underline{\text{F}}$) /id)

→ -(T×(E - $\underline{\text{id}}$) /id)

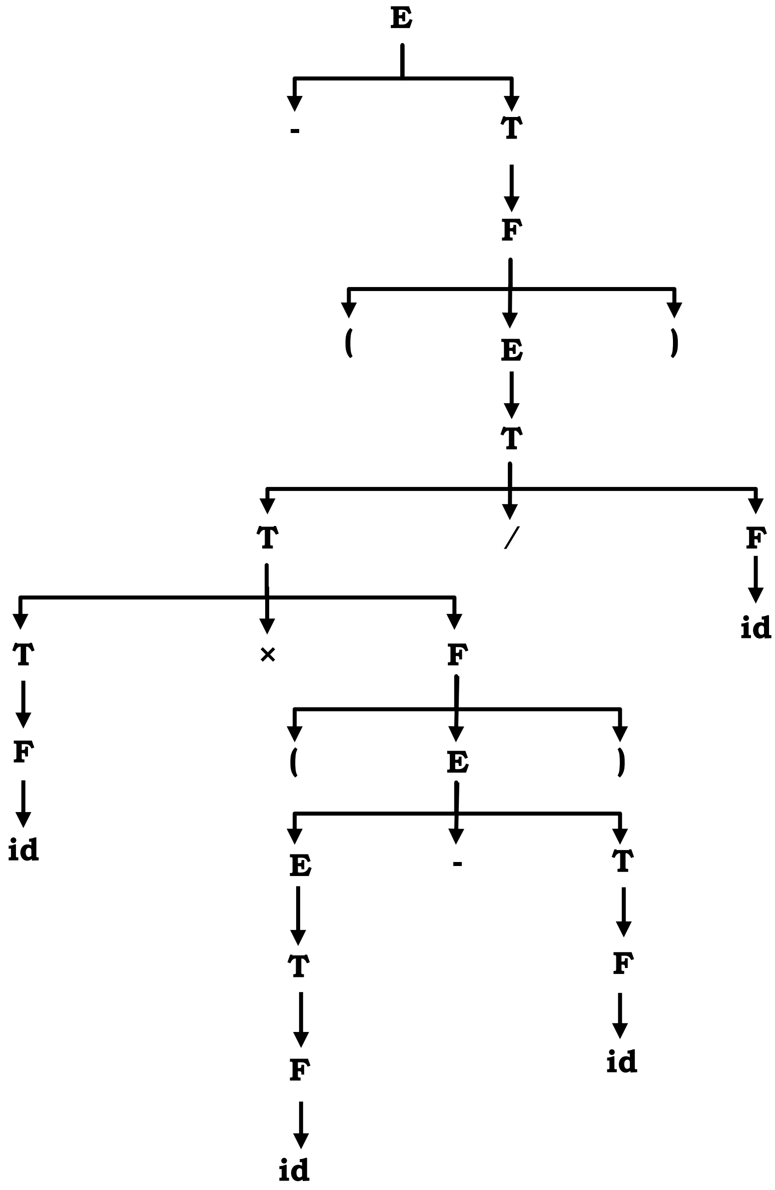
→ -(T×($\underline{\text{T}}$ - id) /id)

→ -(T×($\underline{\text{F}}$ - id) /id)

→ -(T×($\underline{\text{id}}$ - id) /id)

→ -($\underline{\text{F}}$ × (id - id) /id)

→ -($\underline{\text{id}}$ × (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.

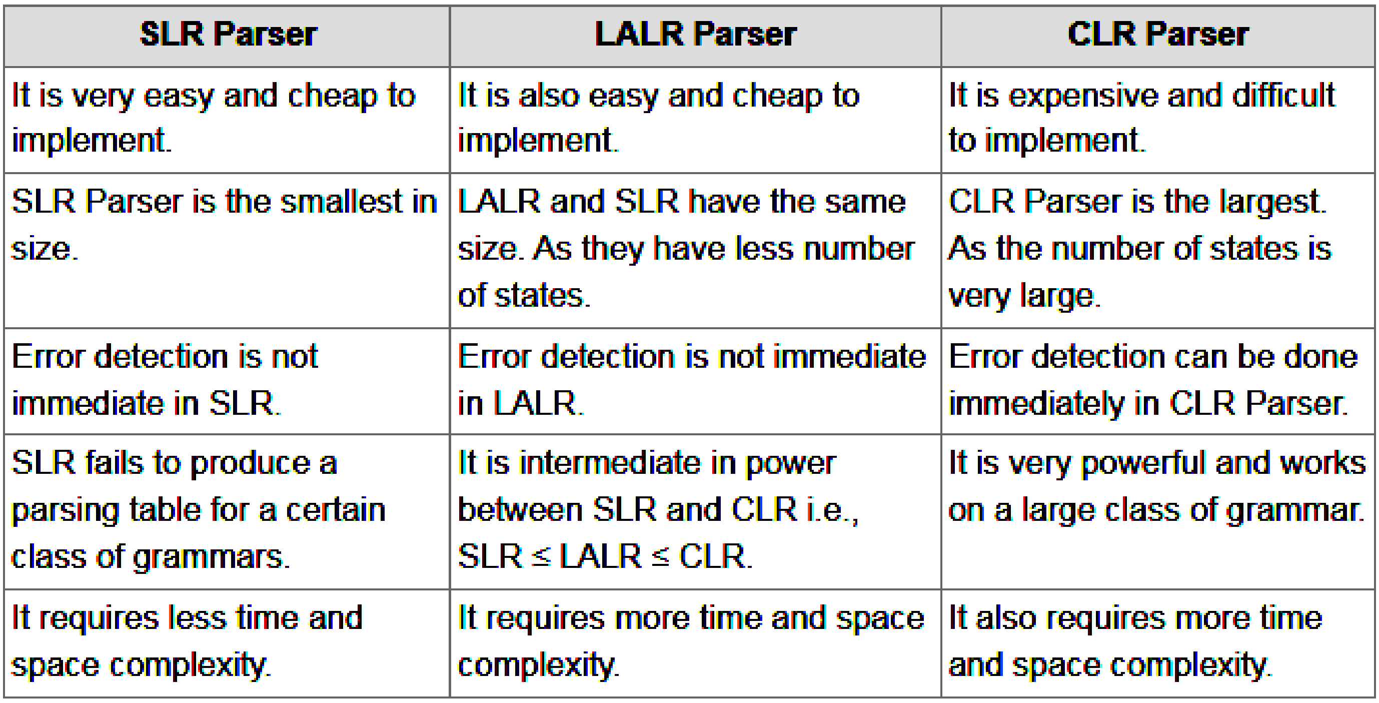
Advantages of LR Parser:-

- ✓ 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.

Semantic Analysis

Immediately followed the parsing phase (Syntax Analyzer). A semantic analyzer checks the source program for semantic errors. *Type-checking* is an important part of semantic analyzer.

The Semantic Analysis of the Compiler is implemented in two passes. The first pass handles the definition of names (check for duplicate names) and completeness (consistency) checks. The second pass completes the scope analysis (check for undefined names) and performs type analysis.

***Example* :- newval = oldval + 12**

The type of the identifier *newval* must match with type of the expression *(oldval+12)*.

If the declaration part for a any programming language segment code for example declares the type of newval as integer type and through the running of the program the value of oldval has a type of real then the Semantic Analysis of the Compiler is implemented through the first pass by giving an error message refers to the type inconsistency (type mismatch).

Two types of semantic Checks are performed within this phase these are:-

1. ***Static Semantic Checks* are performed at compile time like:-**
    - **Type checking.**
    - **Every variable is declared before used.**
    - **Identifiers are used in appropriate contexts.**

2. ***Dynamic Semantic Checks*** are performed at run time, and the compiler produces code that performs these checks:-
   - Array subscript values are within bounds.
   - Arithmetic errors, e.g. division by zero.
   - A variable is used but hasn't been initialized.

Intermediate Code Generator

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This representation should be easy to produce and easy to translate into the target program. These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes.

The forms of codes that are generated in the Intermediate Code Generator phase are:-

1. ***Polish Notation:-*** **which can be performed through the following:**
   - ***Infix Notation :-*** **In which the operation must be in the middle of the expression (between two operands) like A+B.**
   - ***Prefix Notation :-*** **In which the operation must prior the operands (in the left hand side of the operands) like +AB.**
   - ***Postfix Notation :-*** **In which the operation must be in the right hand side of the operands like AB+.**

Example 1:- **Having the following expression**

M= ((D∗E) – ((F + G) / (H + I)))

For Infix Notation **the expression will be as same because the operation is between the two operands.**

For Prefix Notation **the expression will be as shown step by step depending on the notation of the prefix rule which make the operation prior the operand by moving these operations to the left hand side of the operand as shown:-**

1- M= ((D∗E) – ((F + G) / (H + I)))

2- M= (∗(DE) – (+(FG) / +(HI)))

3- M= (∗(DE) – /(+(FG) +(HI)))

4- M= – (∗(DE) /(+(FG) +(HI)))

For Postfix Notation **the expression will be as shown step by step depending on the notation of the postfix rule moves the operations to the right hand side of the operand as shown below:-**

1- M= ((D∗E) – ((F + G) / (H + I)))

2- M= ((DE)∗ – ((FG)+ / (HI)+))

3- M= ((DE) – ((FG)+ (HI)+)/)

4- M= ((DE)∗ ((FG)+ (HI)+)/) –

Example 2:- Having the following expressions in infix form convert them to the two others forms:-

1. U+A∗B	2. (W∗L)-(A/(C∗D))	3. (A+B)∗(C+D)

2. ***Quadruples:-*** In which each expression is performed using the following format:-

Operator, $operand_1$, $operand_2$, result

Example :- Having the following expression M= (A ∗ B) + (Y + Z)

The *Quadruple* format will be:-

+ , Y , Z , T_1

∗ , A , B , T_2

+ , T_1 , T_2 , T_3

3. ***Triples:-*** In which each expression is performed using the following format:-

Operator, $operand_1$, $operand_2$

Example 1:- Having the following expression M= (A ∗ B) + (Y + Z)

The *Triples* format will be:-

Steps			
(1)	+	Y	Z
(2)	∗	A	B
(3)	+	(1)	(2)

Example 2:- **Having the following expression**

$$X= (X_1 + X_2) * (X_2 + X_3) * (X_3 + X_4)$$

The *Quadruple* format will be:-

OP.	$Operand_1$	$Operand_2$	Result	Meaning
+	X_1	X_2	$Temp_1$	ADD X_1, X_2 ,$Temp_1$
+	X_2	X_3	$Temp_2$	ADD X_2, X_3 ,$Temp_2$
+	X_3	X_4	$Temp_3$	ADD X_3, X_4 ,$Temp_3$
*	$Temp_1$	$Temp_2$	$Temp_4$	MULT $Temp_1$, $Temp_2$,$Temp_4$
*	$Temp_4$	$Temp_3$	$Temp_5$	MULT $Temp_4$, $Temp_3$,$Temp_5$
:=	$Temp_5$	----------	---------	MOV $Temp_5$, X

The *Triple* format will be:-

Steps	Operation	$Operand_1$	$Operand_2$
(0)	+	X_1	X_2
(1)	+	X_2	X_3
(2)	+	X_3	X_4
(3)	*	(0)	(1)
(4)	*	(3)	(2)
	:=	X	(4)

Three Address Code **Is a sequence of statements typically of the general form A := B op C, where A,B and C are temporary operands and op is the operation. The cause of naming this format by *Three Address Code* is that each statement or expression usually contains three addresses, two for operands and one for the result.**

The following expression X= $(X_1 + X_2) * (X_2 + X_3) * (X_3 + X_4)$ will performed using *Three Address Code* as shown below:-

Steps

T_1	+ , X_1 , X_2
T_2	+ , X_2 , X_3
T_3	+ , X_3 , X_4
T_4	* , T_1 , T_2
T_5	* , T_4 , T_3
X	= T_5

Code optimizer

Code Optimization
Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.
In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes

code optimizing process must follow the three rules given below:
-The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible.
- The program should demand less number of resources.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Optimization can be categorized broadly into two types: machine independent and machine dependent.

1- Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. (Machine Independent improvements address the logic of the program)
For example:

```
do

{

item = 10;
value = value + item;

}while(value<100);
```

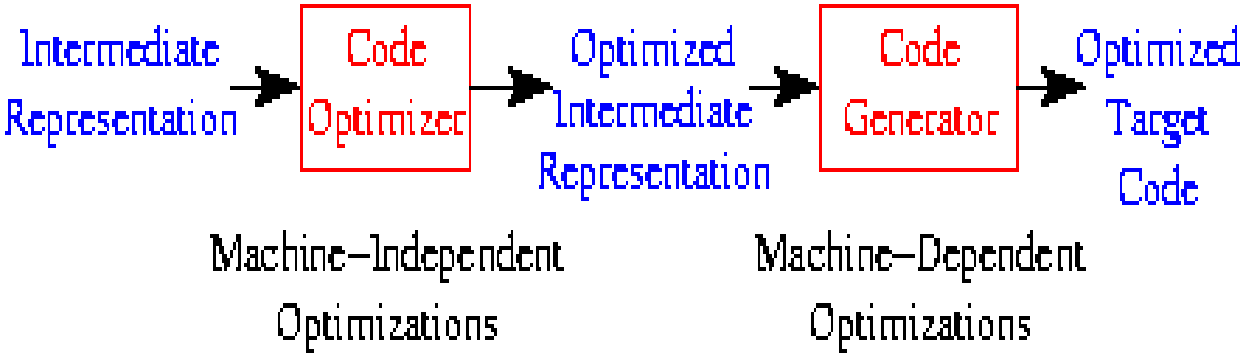
This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;

 do

{

value = value + item;

} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

2- Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.


Intermediate Representation
Code Optimizer
Optimized Intermediate Representation
Code Generator
Optimized Target Code
Machine-Independent Optimizations
Machine-Dependent Optimizations

Peephole optimization: - peephole optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognizing sets of instructions that can be replaced by shorter or faster sets of instructions.

Code Optimization has Two levels which are:-

1- Machine independent code Optimization
- Control Flow analysis
- Data Flow analysis
- Transformation

2- Machine dependent code- Optimization
- Register allocation
- Utilization of special instructions.

Code optimization can either be high level or low level:
– High level code optimizations.
– Low level code optimizations.
– Some optimization can be done in both levels.

Flow graph: - is a common intermediate representation for code optimization.

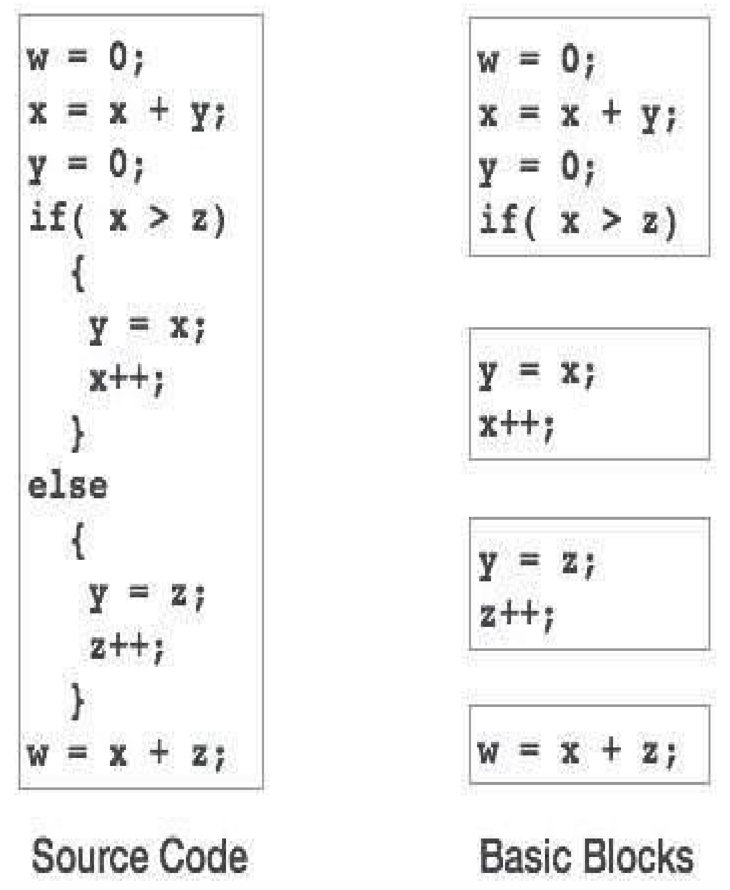
Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic blocks are important concepts from both code generation and optimization point of view.

Local Optimizations are performed on basic blocks of code
Global Optimizations are performed on the whole code



```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Code

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

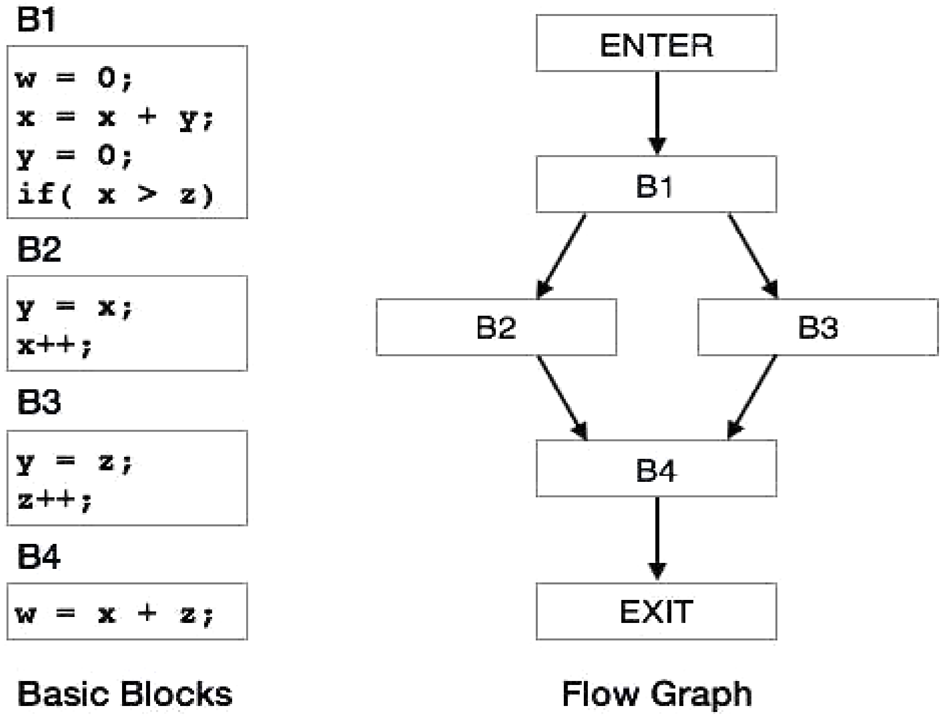
```
y = z;
z++;
```

```
w = x + z;
```

Basic Blocks

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.



```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

```
w = x + z;
```

Global Data Flow Analysis

Compiler collect information about all program that needed for code optimizer phase, Collect information about the whole program and distribute the information to each block in the flow graph.

DFA provide information for global optimization about how execution program manipulate data.

- *Data flow information*: Information collected by data flow analysis.
- *Data flow equations*: A set of equations solved by data flow analysis to gather data flow information.

Criteria for code-improvement Transformations

1. Transformations must preserve the meaning of programs
2. A transformation must, on the average, speed up programs by a measurable amount
3. A transformation must be worth the effort.

Function Preserving Transformations

1. Common sub expression eliminations
2. Copy propagations
3. Dead and unreachable code elimination
4. Constant Folding

Code Generation

Code Generation

Code generation is the final phase of compiler phases, It takes input from the intermediate representation with information in symbol table of the source program and produces as output an equivalent target program (see Figure 1).

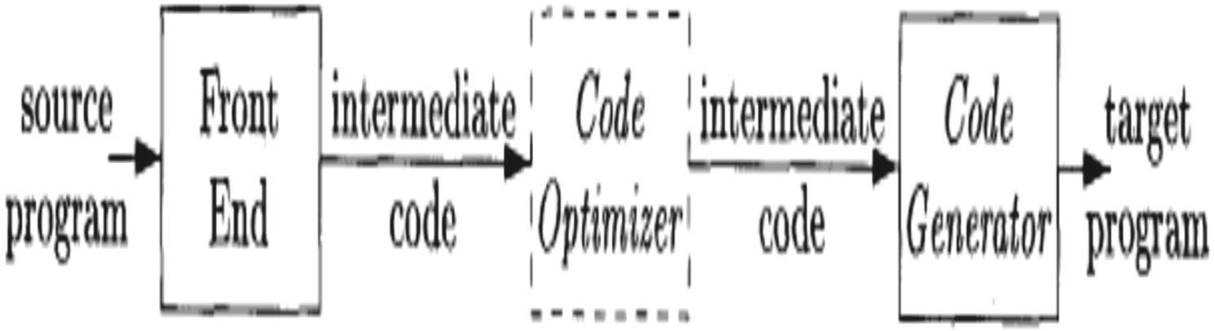


Figure 1: position of Code generation

Main Tasks of Code Generator

1- **Instruction selection**: choosing appropriate target-machine instructions to implement the IR statements.

The complexity of mapping IR program into code-sequence for target machine depends on:

– Level of IR (high-level or low-level)

– Nature of instruction set (data type support)

– Desired quality of generated code (speed and size)

2- **Registers allocation and assignment:** deciding what values to keep in which registers

3- **Instruction ordering:** deciding in what order to schedule the execution of instructions.

Issues in the design of code generator

1- Input to the code generator

• three-address presentations (quadruples, triples, …)

• Virtual machine presentations (bytecode, stack-machine, …)

• Linear presentation (postfix …)

• Graphical presentation (syntax trees, DAGs,…)

2- The target program

Instruction set architecture (RISC, CISC)

The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code. The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.

A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.

In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers. Stack-based machines almost disappeared because it was felt

that the stack organization was too limiting and required too many swap and copy operations.

Output may take variety of forms

1. Absolute machine language(executable code)
2. Relocatable machine language(object files for linker)
3. Assembly language(facilitates debugging)

Absolute machine language has advantage that it can be placed in a fixed location in memory and immediately executed.

Relocatable machine language program allows subprograms to be compiled separately.

Producing assembly language program as output makes the process of code generation somewhat easier.