Compilers

University of Baghdad **M.Sc. Shaimaa Abbas**
College of Education / Ibn-AL-Haithem **2008-2009**
Dep. Of Computer Science **Third Stage**

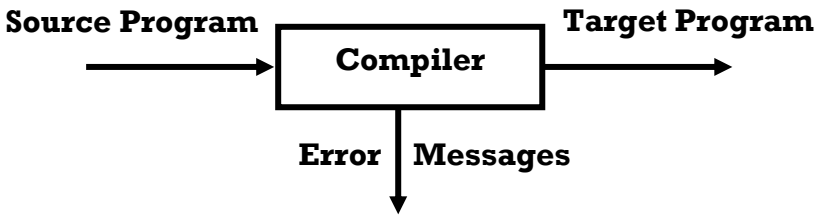
Chapter One

REFRENCES:-

- **Compilers Principles, Techniques and Tools by Alferd V.Aho.**
- **Compiler Construction for Digital Computers by David Gries.**
- **Introduction Theory of Computer Science by E.R. Krishuamurthy.**
- الأسس النظرية و التطبيقية لتصميم مترجمات لغات برمجه الحاسبة د.صباح محمد أمين د.جنان عبد الوهاب

Def.

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 -.**

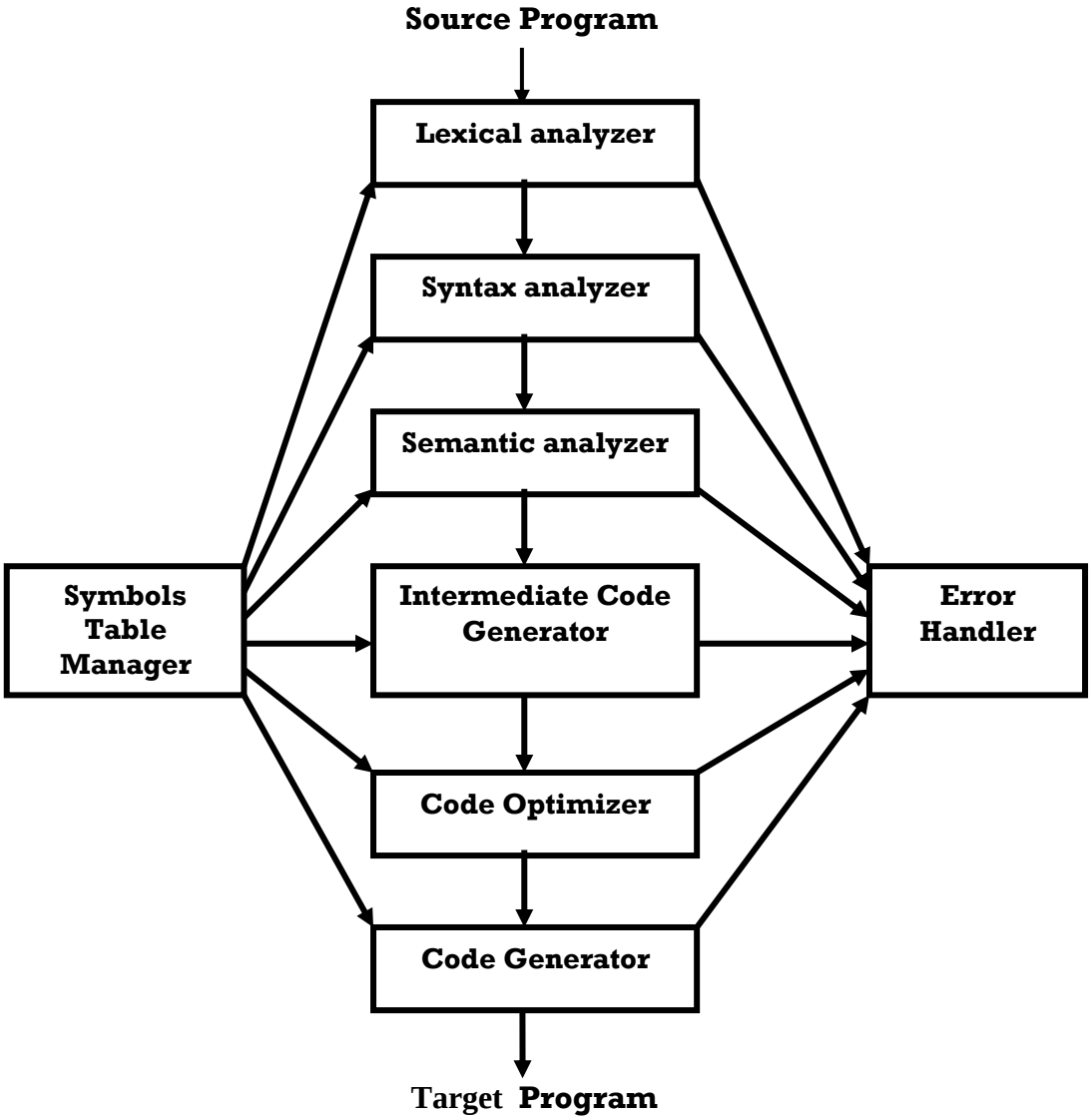


The Phases of a Compiler **:-**

A typical decomposition of a compiler is shown below, in practice, some of the phases may be grouped together.

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

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*** **:-**

Its main task is to read the source program (character by character) then translated into a sequence of primitive units called *tokens* like (keywords, identifier, constant, operators, etc.).

Some times this phase is divided into two phases, the first one known as "*Scanning*" while the second is known as "*Lexical Analysis*". The Scanning is responsible for doing simple tasks while the Lexical Analysis is suitable for doing complex tasks.

- ***Syntax analyzer*** **:-**

This phase begins when the lexical phase is terminated; the outputs from the previous phase (Lexical analyzer) will represent the input for this phase (Syntax analyzer).

- ***Intermediate Code Generator*** **:-**

After syntax and semantic analysis, some compilers generate an explicit intermediate representation for the source program. This phase has two important properties :- it should easy to produce and translate into the target program.

- ***Code Optimization phase*** **:-**

The code optimization phase attempts to improve the intermediate code which results into faster running machine code.

- ***Code Generator*** **:-**

The final phase of complier is the generation of target code, consisting of relocatable machine code or assembly code.

- ***Symbol Table*** **:-**

Symbol table is a data structure containing a record for each identifier, with fields for attributes of the identifier. This data structure allows us to store and retrieve data from the record quickly.

- ***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 a 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).**

Grammars **:-**

Chapter One

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

Def.
The set of rules which we use to reconstruct grammatical sentences are called *Syntax*.

Def.
The specification of the meaning of sentences in a language is called the *Semantics* of the language.

Def.
Let $\sum$ be any finite set of symbols, called an alphabet; the symbols are called the letters of the alphabet.

Def.
A word (String) X over $\sum$ is any finite sequence of letters from $\sum$, while the empty word, denoted by ε or λ, is the word consisting of no letters.

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

$$UV = \{ X \mid X = uv, u \text{ is in } U \text{ and } v \text{ is in } V \}$$

Note that:- $UV \neq VU$

$U(VW) = (UV)W$

***Example* ①:-**

Let $\sum = \{0,1\}$ and $U = \{000,111\}$ and $V = \{101,010\}$

⇨ $UV = \{000101, 000010, 111101, 111010\}$

⇨ $VU = \{101000, 101111, 010000, 010111\}$

$\therefore UV \neq VU$

***Example* ②:-**

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

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

Take first the left side,

U (VW) ={abd , bcd} {bcdda, bcdbd, cabda, cabbd}

= { abdbcdda, abdbcdbd, abdcabda, abdcabbd, bcdbcdda, bcdbcdbd, bcdcabda, bcdcabbd }

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 concatinating 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^*={ε , 01,11 , 0101,0111,1101,1111 , 010101 , 010111, … }

S^0 S^1 S^2 S^3

Fo tion:-

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 $\cup$ T =V and N $\cap$ 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 $\rightarrow$ aba), (SB $\rightarrow$b), (b$\rightarrow$bB), (b$\rightarrow\lambda$)}

This grammar is not a structure grammar because of the production rule b $\rightarrow$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.

Def.

The set of all sentences generated by G is called the language of G or L (G)

$$L(G)=\{X \mid X \in T^{*} \text{ and } S \xrightarrow[G]{} X\}$$

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)

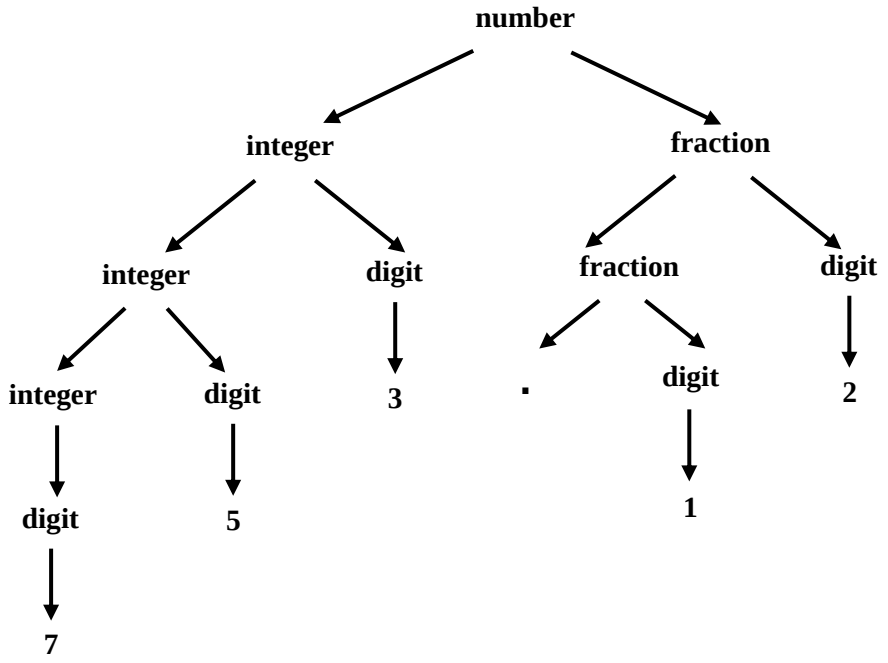
N= {number, integer, fraction, digit}

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 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. **If S,R any two R.E. denoting L_S,L_R then**
   - 3.1 **R | S is R.E. denoting $L_R \cup L_S$**
   - 3.2 **RS is R.E. denoting $L_R . L_S$**
   - 3.3 **R^* is R.E. denoting $\{\varepsilon\} \cup L_R \cup L_R^2 \cup \ldots \cup L_R^n$**

- **S | R ≡ R | S**
- **(R | S) | T ≡ R | (S | T)**
- **(R.S).T ≡ R. (S.T)**
- **R.S | R.T ≡ R. (S | T)**
- **R ≡ R.λ ≡ λ.R**

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

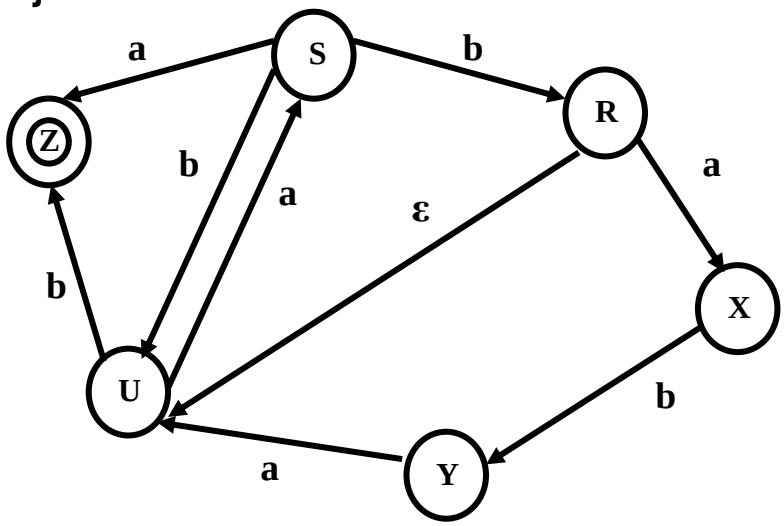
U → b

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>

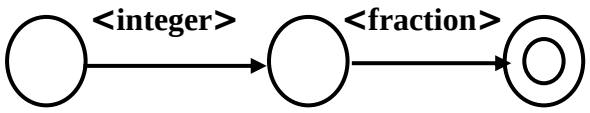
<integer> ::= <digit>|<integer> <digit>

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

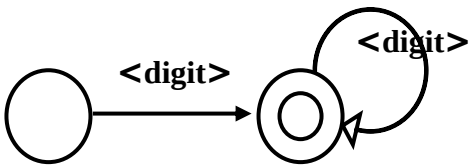
<digit> ::= 0|1|2|3|4|5|6|7|8|9

Now we take each production rule and draw to it a separate transition diagram:-

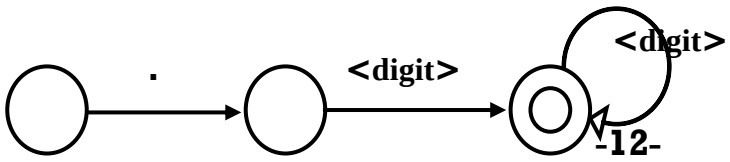
<number> ::= <integer> <fraction>



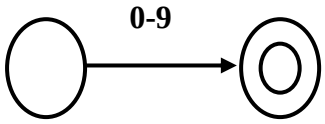
<integer> ::= <digit>|<integer> <digit>



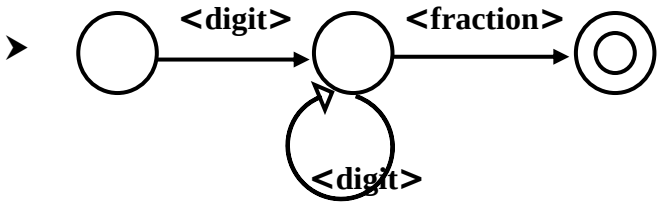
<fraction> ::= .<digit>|<fraction> <digit>

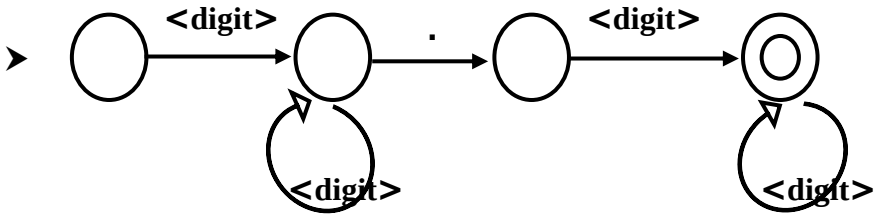


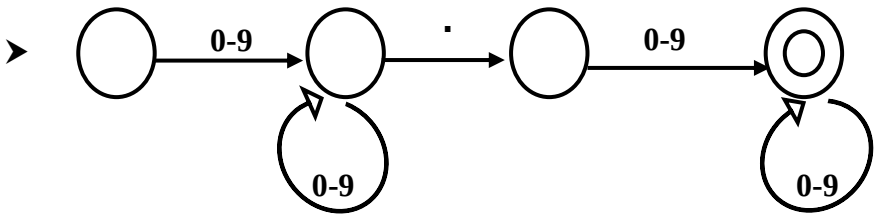
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



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



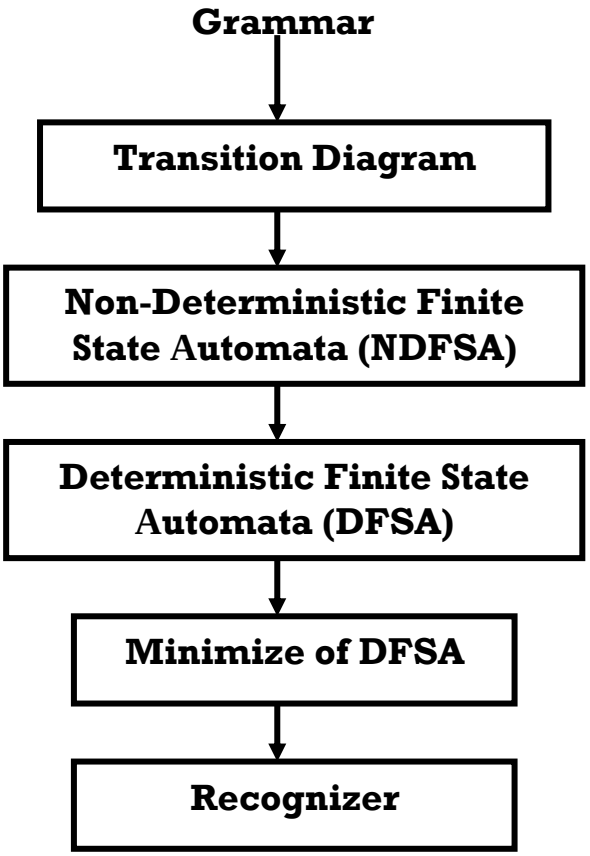




Chapter Two

Lexical Analyzer Design

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



- **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. **A set of transition functions**
3. **Initial state**
4. **Final state**
5. **A set of elements called states**

Two types of *FSA*:-

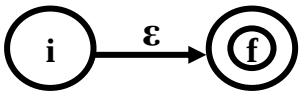
- **Non-Deterministic Finite State Automata (NDFSA)**
- **Deterministic Finite State Automata (DFSA)**

***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.**

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

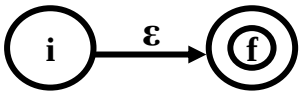
① If we have an ε



where i = initial state , f =final state

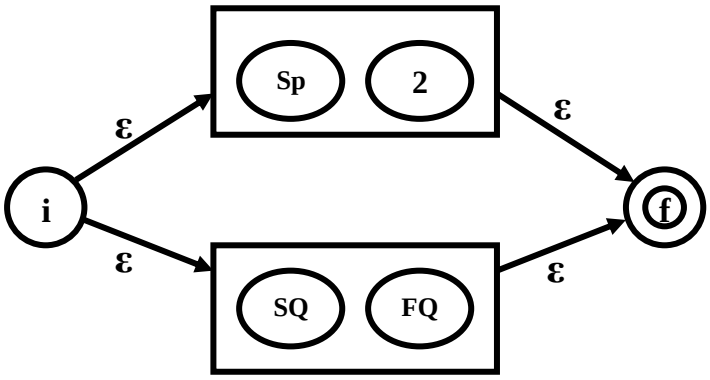
.....

② If we find a terminal symbol like a



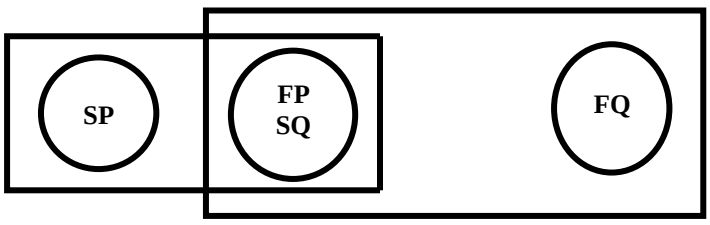
.....

③ If we have P | Q

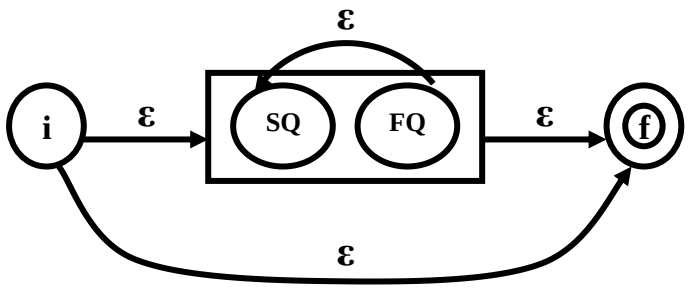


.....

④ If we have P.Q

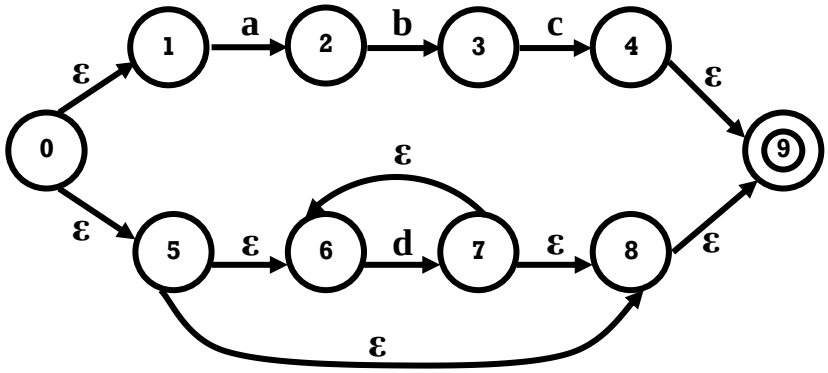


⑤ **If we have Q***



Example **:-**

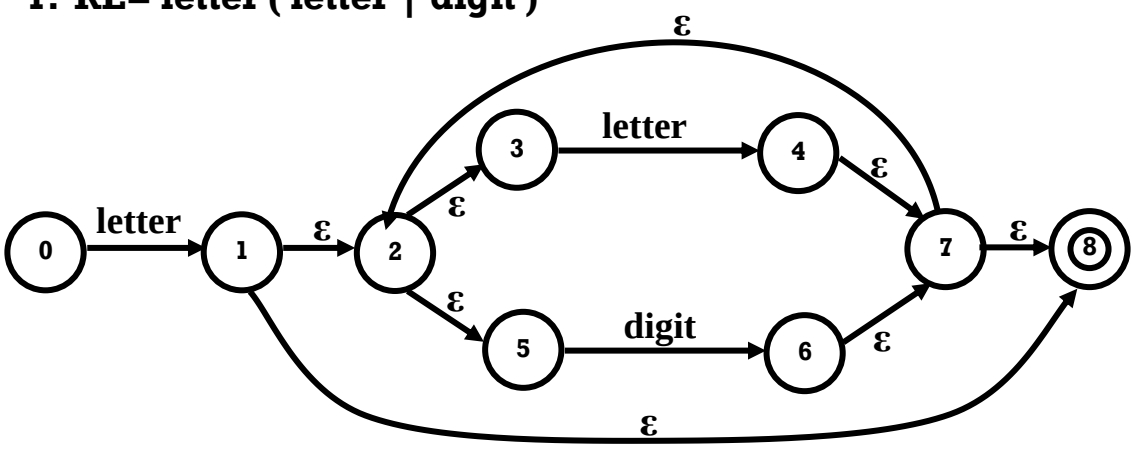
R.E.= abc|d*



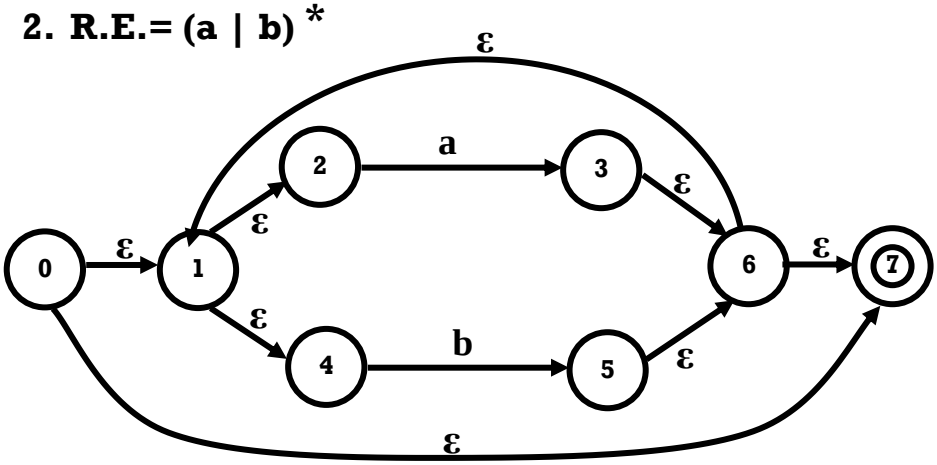
.....

Examples **:-**

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

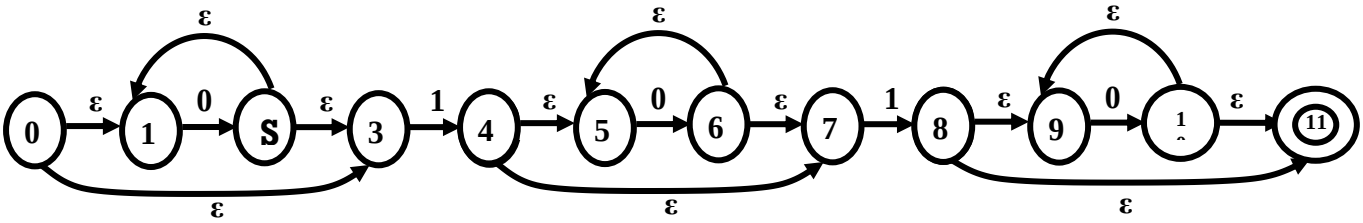


2. R.E.= (a | b) *



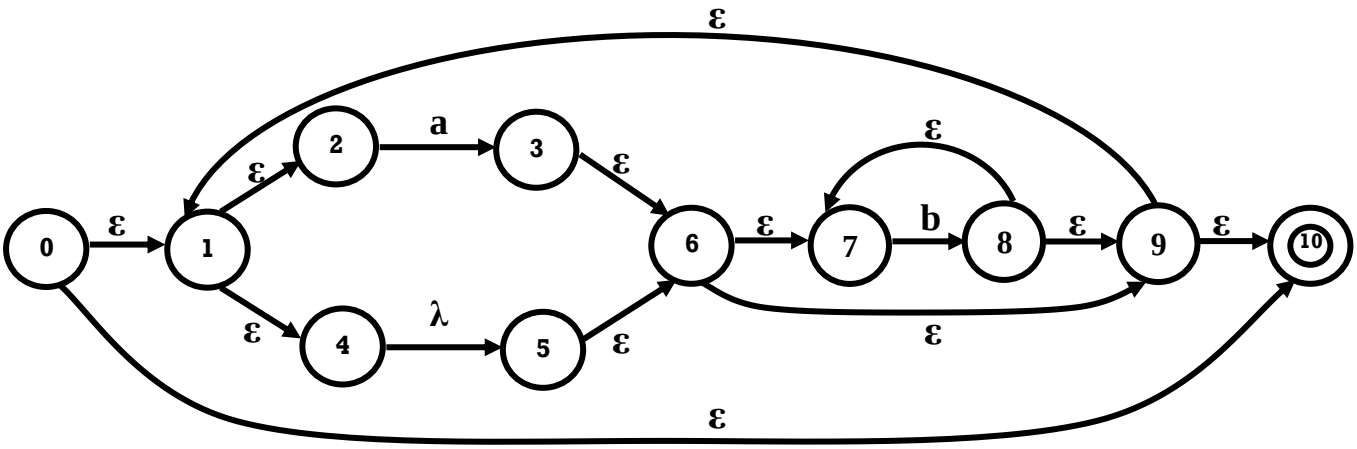
.....

3. R.E.= $0^*1\ 0^*1\ 0^*$



.....

4. R.E.= $((\lambda \mid a)\ b^*)^*$



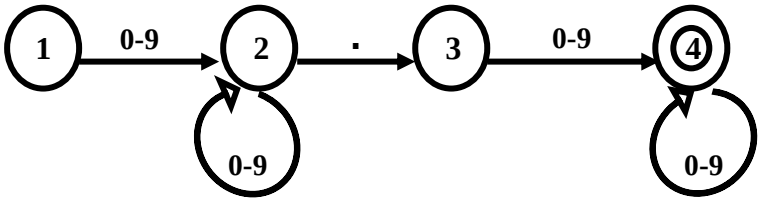
Data structure representation of FSA :-

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:-

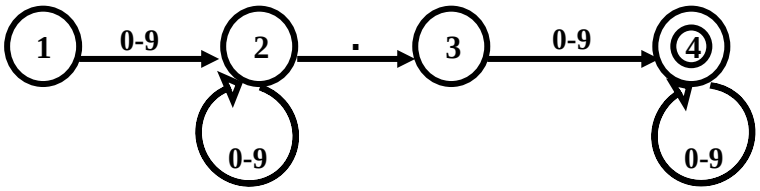


	0-9	.
1	2	#
2	2	3
3	4	#
4	4	#

② ***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:-

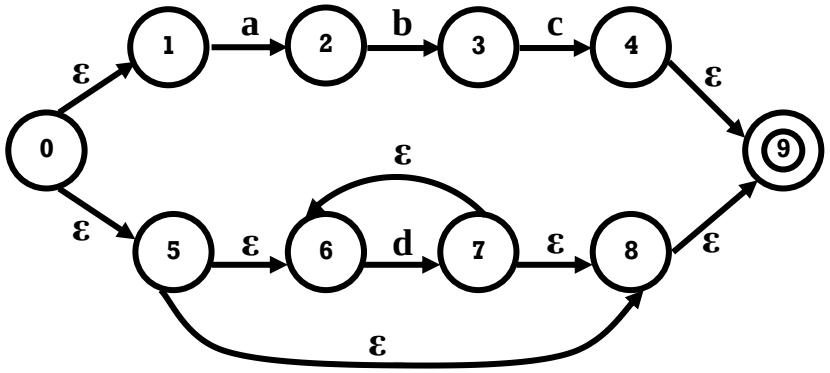
Before we use an algorithm to convert the grammar which is NDFSA form to DFSA form, we must deal with a special function known as ε-*Closure Function*, which can be explained using the following procedure:-

Function ε-Closure (M) :-

```
Begin
   Push all states in M  into stack;
   Initialize ε-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 ε-Closure (M) then
          Begin
             Push X;
             Add  X to ε-Closure (M);
          End;
      End;
End;
```

Example **:-**

R.E.= abc|d*



To compute randomly the Ɛ-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= ε-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 ∈ Σ) do

Begin

Let M be the set of states to which there is transition on *a* from some states S_i in X ;

Y = ε-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;

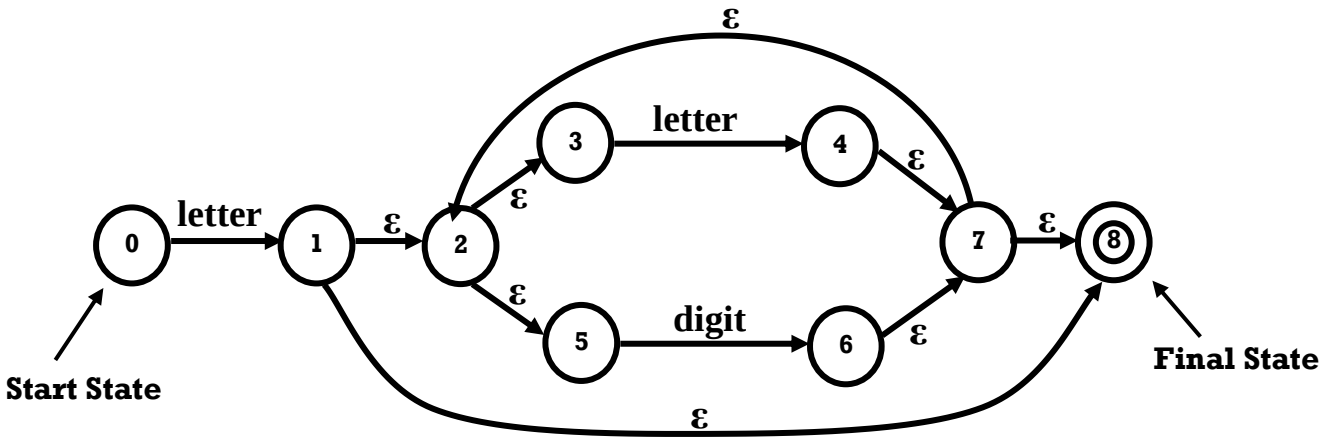
End;

End;

End {algorithm}

***Examples*:-**

① R.E. = Letter (letter | digit)*



ε-Closure ({ 0 }) = {0} ◄····· **Create a new node called for example A**

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

A → **digit ; M=∅;**

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

B → **digit ; M={6}; ε-Closure ({6})={6,7,8,2,3,5}** ◄····· **Create a new node called for example D (must be a final node because of node 8).**

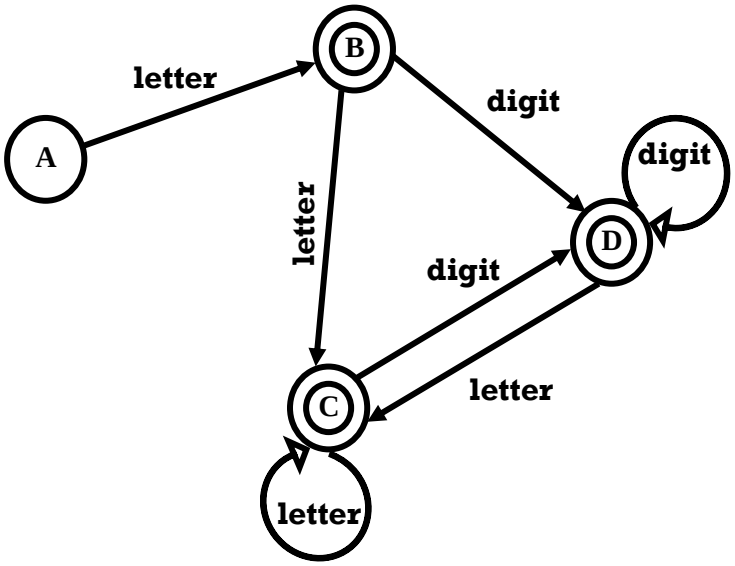
C → **letter ; M={4}; No need to create a new node because ε-Closure ({4}) has been computed and by which we have node C.**

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

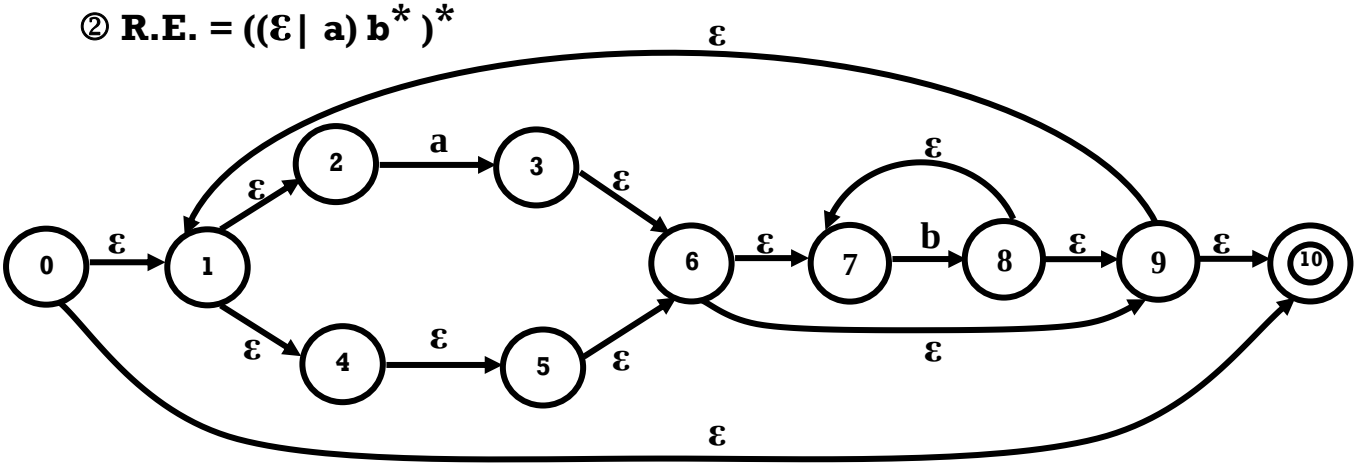
D → letter ; M={4}; No need to create a new node because ε-Closure ({4}) has been computed and by which we have node C.

digit ; M={6}; No need to create a new node because ε-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. = ((ε| a) b*)***



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

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

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

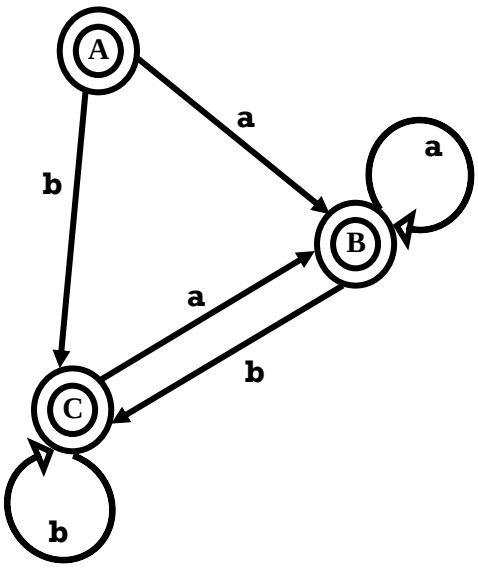
B → **a ; M={3}; No need to create a new node because ε-Closure ({3}) has been computed and by which we have node B.**

B → **b ; M={8}; No need to create a new node because ε-Closure ({8}) has been computed and by which we have node C.**

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

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

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**

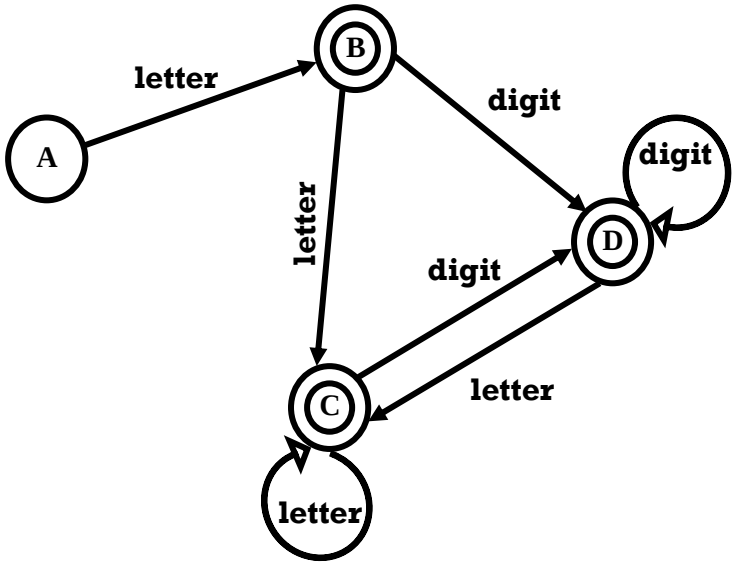
   Begin

   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 $Л_{new}$ by the set of all subgroups formed .

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

Example **:-**

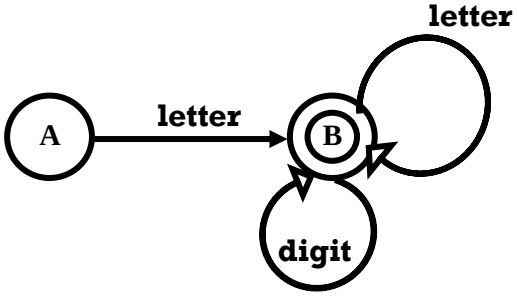
The DFSA for the R.E. = Letter (letter | digit)* is as follows:-



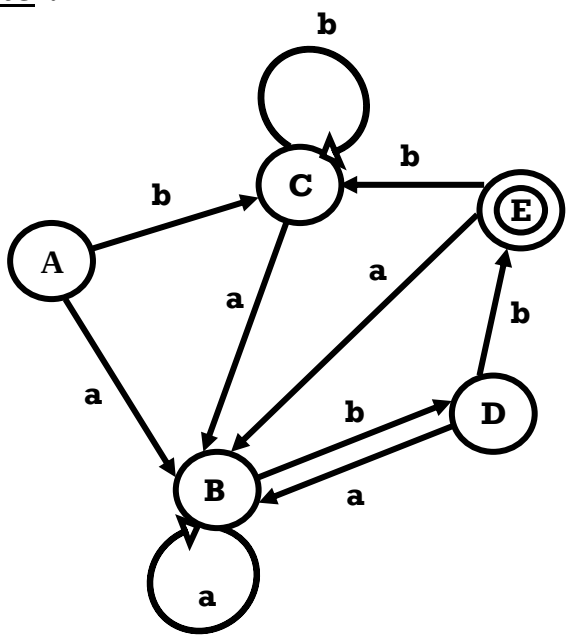
$Group_1$= {A} which represents the set of not final nodes while $Group_2$ = {B,C,D} which represents the set of final nodes.

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:-



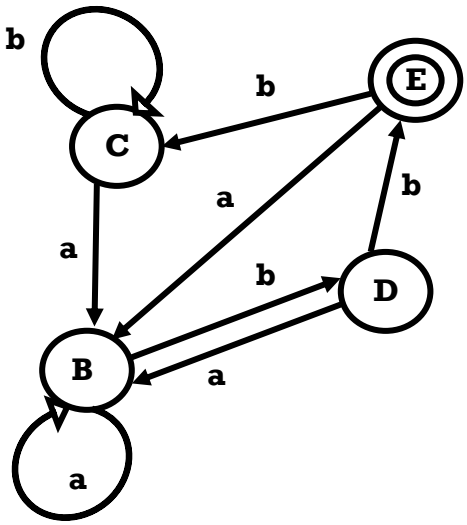
Another example **:-**



$Group_1$= {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.

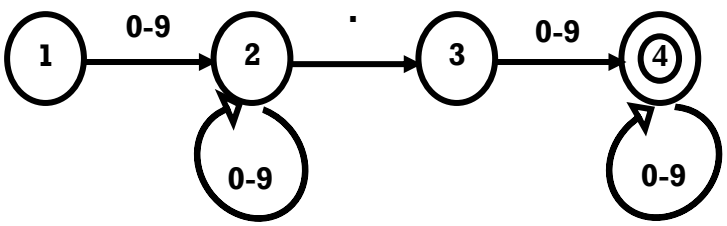
Never can apply the algorithm unless the grammar will be in *minimized form*.

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 1.3$ and 37$,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 = 1.3 $

State	Input symbol
1	1
2	.
3	3
4	$

It is accepted because state number 4 is a final State

For the String = 37 $

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

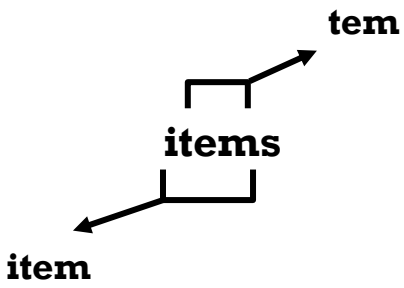
This algorithm was slow and overlapping token, so a new algorithm can be used to recognize the overlapping token.

***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*

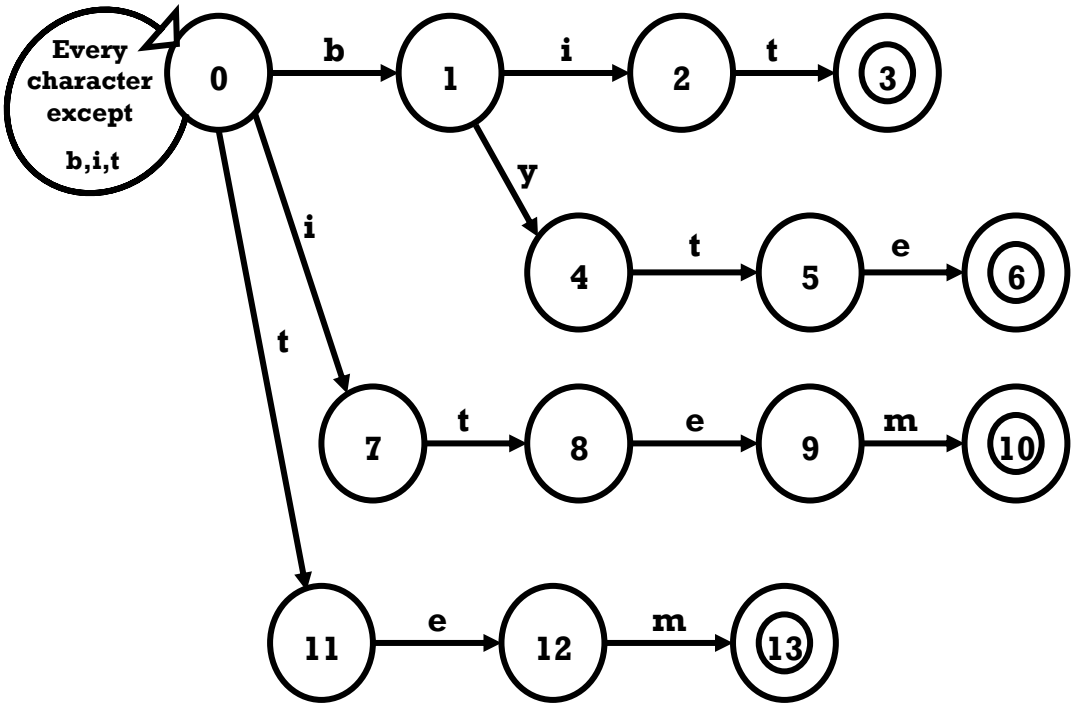


The new algorithm is known as <u>AHO Algorithm</u> and depends on the following steps:-

(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:-

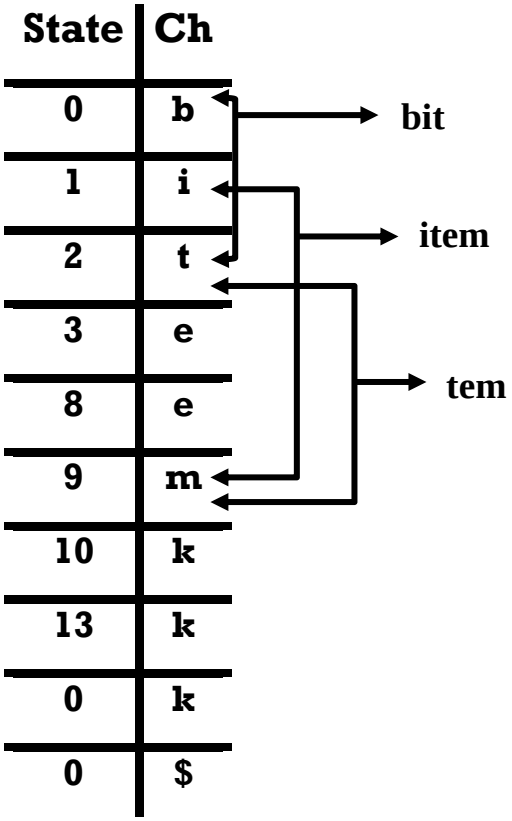
***Algorithm* :-**

```
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 = bitemk$ 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**

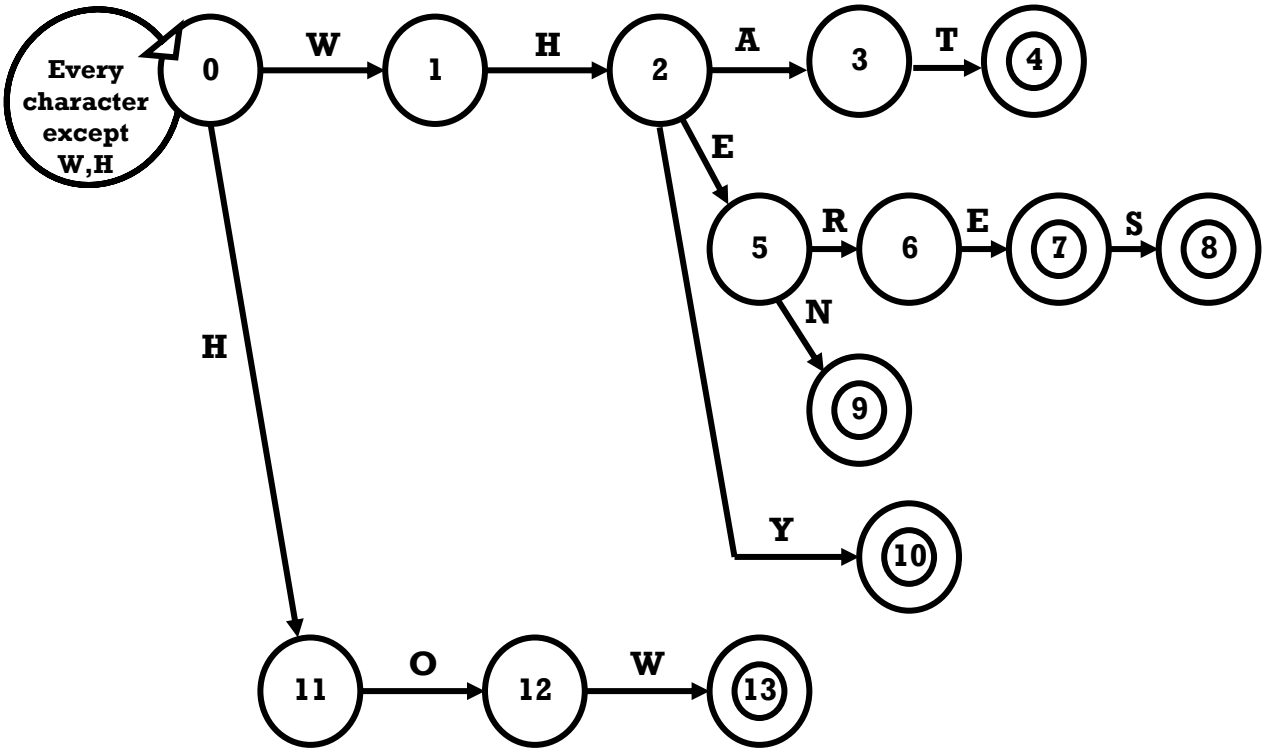


Example **:-**

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.



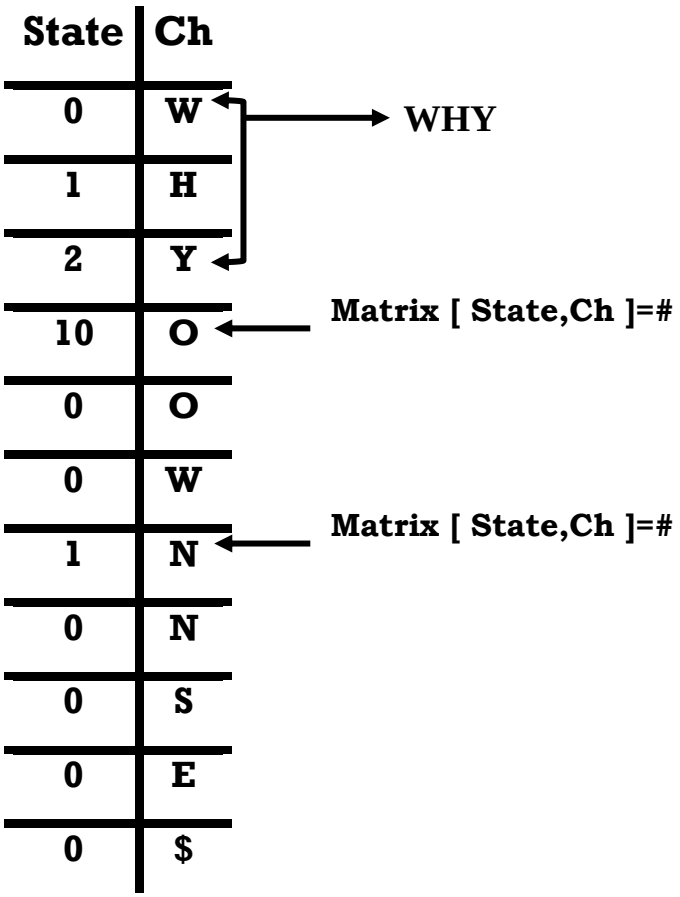
Step 2:- Compute fall back function f (Q) as follows:-

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>).



Syntax Analyzer

Example **:-**

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

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

P=

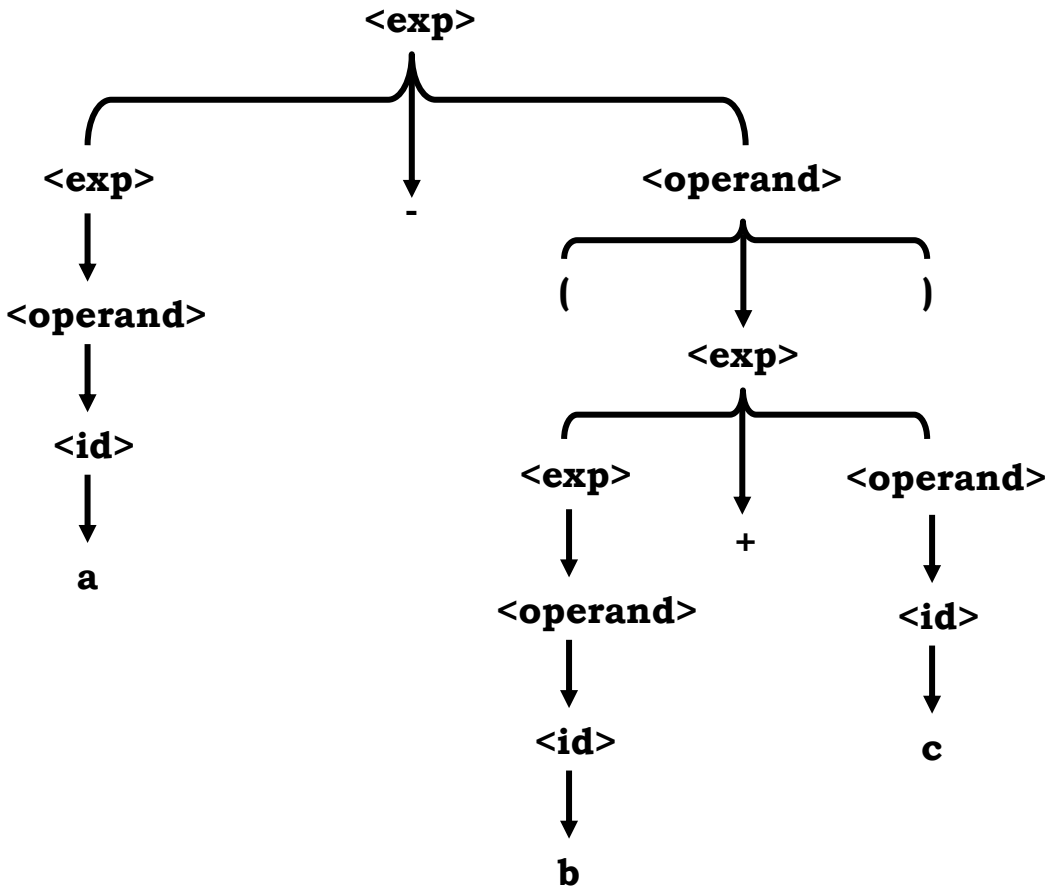
<exp> ::= <operand> | <exp> + <operand> | <exp> - <operand>

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

<id> ::= a | b |c

Syntax analyzer utilizes syntax trees to determine whether a statement is accepted or not.

For the above example, check if a-(b+c) accepted?



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

There are two types of derivation:-

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

Example :-

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

P=

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

Is a×(b+a) accepted or not?

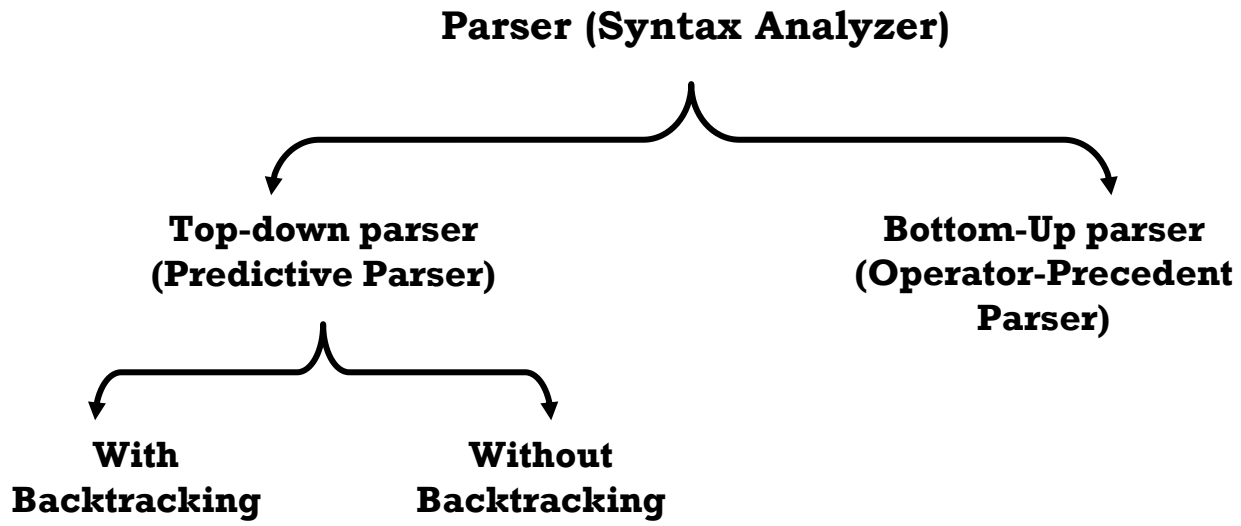
Leftmost derivation :-

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

Rightmost derivation :-

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

Parser Techniques :-



Backtracking manipulating:-

1. **Factoring**
2. **Substitution**
3. **Left-Recursion Elimination** إلغاء تكرار العنصر في أقصى يسار الطرف الأيمن

   E → E+A

Left Recursion Elimination :-

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

Immediate Left-Recursion Elimination :-

The main rule for this method:-

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

$A \rightarrow \beta_1 Á \mid \beta_2 Á \mid \ldots \mid \beta_n Á$

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

***Example* 1:-**

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

$A \rightarrow \beta\, Á$

$Á \rightarrow \alpha\, Á \mid \varepsilon$

***Example* 2:-**

$E \rightarrow abc \mid def \mid Erx$

$É \rightarrow abc \mid defÉ$

$É \rightarrow rxÉ \mid \varepsilon$

***Example* 3:-**

exp → exp or term | term

term → term and factor | factor

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

exp → term exp´

exp´ → or term exp´ | ε

term → factor term´

term´ → and factor term´ | ε

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

Not Immediate Left-Recursion Elimination **:-**

Algorithm:-

Arrange NT in any order;

For I :=2 to n do

For J := 1 to i-1 do

Begin

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

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

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

End;

Eliminate the immediate left recursion among the A_i productions;

End;{of algorithm}

Example **①:-**

B → A c/d

A → Br/x

***Solution*:-**

A_1=B A_2=A

$A_1 \rightarrow A_2 c/d$

$A_2 \rightarrow A_1 r/x$

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

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

.....

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

$A_1 \rightarrow \partial_1 / \partial_2$ $\because$ **$A_1 \rightarrow A_2c/d$** $\therefore$ **$\partial_1 = A_2c$ and $\partial_1 = d$**

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

$A_2 \rightarrow \partial_1 \alpha / \partial_2 \alpha$ $\therefore$ **$A_2 \rightarrow A_2c\ r / d\ r / x$**

.....

$A_1 \rightarrow A_2c/d$
$A_2 \rightarrow A_2c\ r / d\ r / x$ ⟶ **These two rules are converted to immediate backtracking which can be eliminated by the following rules:-**

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

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

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

B→ Ac/d

A → Ac r /d r/ x

The result will be:-

B → Ac/d

A → d r Á / x Á

Á → c r Á /ε

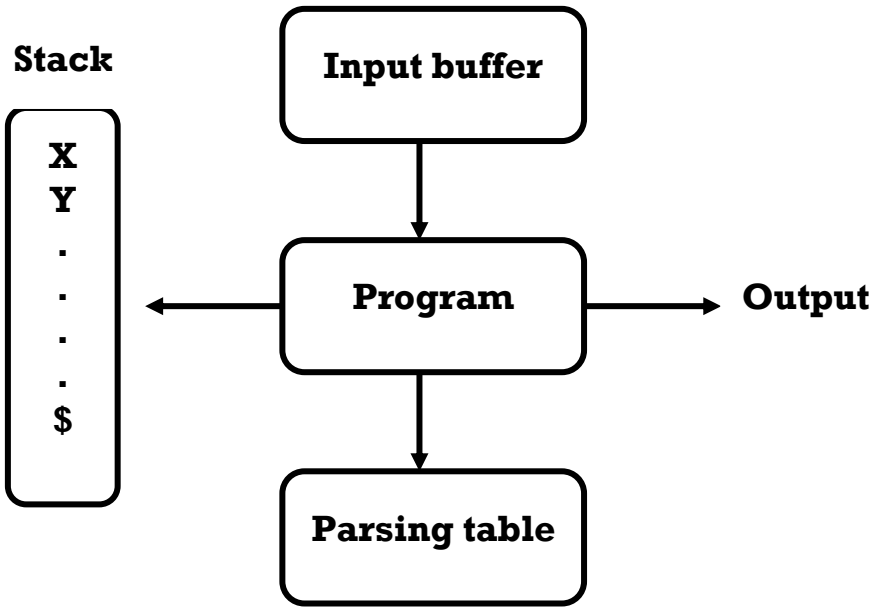
***Example* ②:-**

S → A b / b

A → Ac / Sd/ e

Predicative Parsing (Top Down Parser):-

Architecture:-



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 $Y_1 Y_2 \dots Y_k$ 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=$;

الشـــرط الأساســـي للإعـــراب بطريقـــة (Top-Down) هـــو خلـــو القواعـــد مـــن الرجـــوع الخلفـــي (Backtracking).

فإذا كانت القواعد تحتوي على الرجوع الخلفي فلابد من التأكد من نوع الرجوع الخلفي فيما إذا كان من النوع المباشـــر (Immediate Backtracking) أو غيـــر المباشـــر (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 إعطاء جدول بعدد من الأسطر والأعمدة حيث أن عناصر الأسطر تمثل عناصر Non-Terminal أما قيم أو عناصر الأعمدة فتمثل عناصر Terminal.

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 أي تقاطع السطر X مع العمود a وان تلك العلاقة سوف يتم إضافتها في العمود الخامس وسحب من Stack العنصر الموجود في القمة وعمل Push للطرف الأيمن من العلاقة ولكن بالمقلوب ويبقى حقل a بدون تغيير وكذلك حقل Input.

3. نستمر بتكرار الخطوات الأولى والثانية طالما قيمة Stack ≠ $.

Example ①:-

Having the following grammar:-

E→E+T / T

T→T×F / F

F→(E) / id

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

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

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:-

exp → exp or term / term

term → term and factor / factor

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

Depending on the following table, parse the following statement:-

not (true or false) $

	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	

***Bottom Up Parser* (*Shift-Reduce Parser*) :- 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 = NT, + = T, E=NT).

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

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.

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 .

ولا تهتم هذه الطريقة بوجود أو عدم وجود رجوع خلفي في القواعد المطلوب التعامل معها .

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

خطوات الخوارزمية :-

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

***Example* ①:-**

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

Sol.

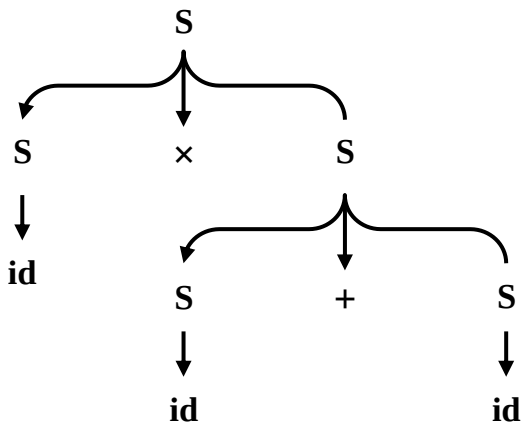
①*Derive this grammar using right most derivation:-*

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

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

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

③*Doing Syntax tree (parse tree):-*



④*Doing Parse table:-*

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

Example ②:-

E → T / E+T / E-T / -T **Input = -(id×(id-id) / id)$**

T → F / T×F/ T/F

F → (E) / id

Solution :-

E → -T

→ -F

→ -(E)

→ -(T)

→-(T/F)

→-(T/ id)

→-(T×F / id)

→-(T× (E) /id)

→ -(T×(E-T) /id)

→ -(T×(E - F) /id)

→ -(T×(E - id) /id)

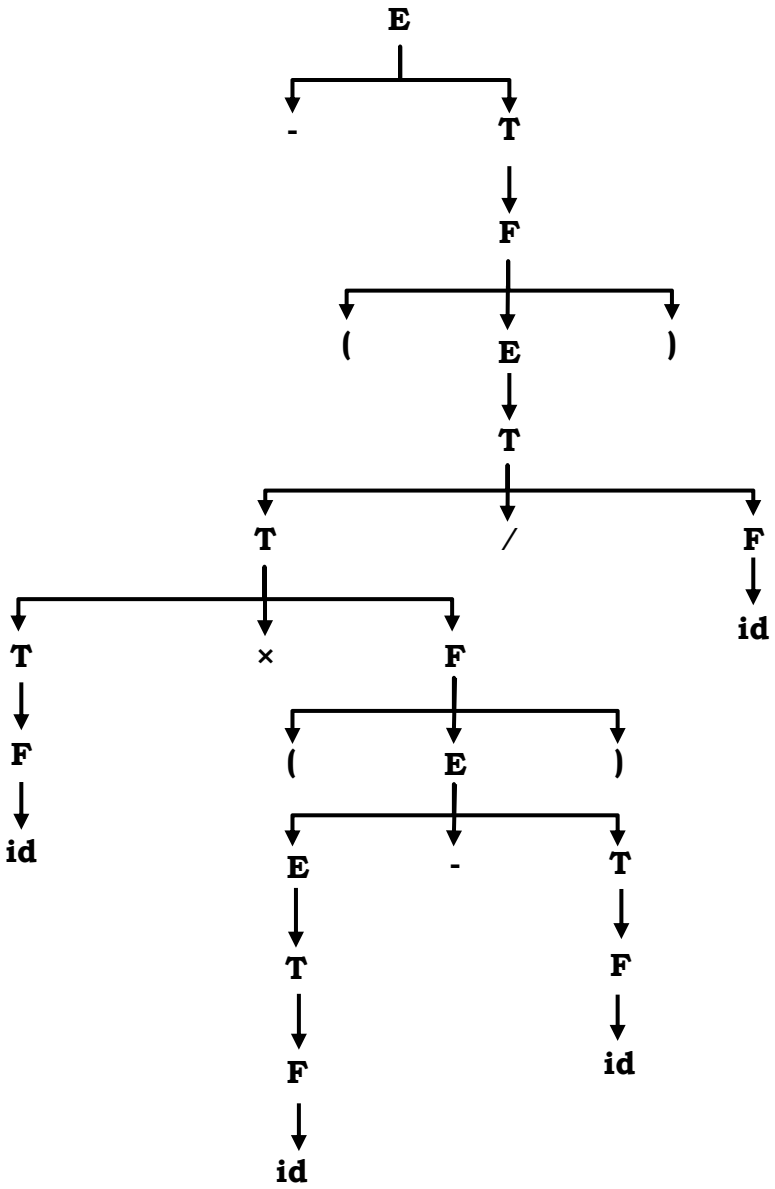
→ -(T×(T - id) /id)

→ -(T×(F - id) /id)

→ -(T×(id - id) /id)

→ -(F × (id - id) /id)

→ -(id × (id - id) /id)



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

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

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

M.Sc. Shaimaa Abbas
2009-2010
Third Stage

Chapter Five

❖ What is a Context-Free Grammar CFG?

Context free Grammars are more general than Regular Expression. There are more languages defined by CFGs (called Type-2 Language).

A Context Free Grammar G=(V,Σ,S,P), Where:-

- **V Is a finite set of variables or Non terminal symbols.**
- **Σ Is a finite set of terminal symbols T.**
- **S Is the start symbol.**
- **P Is a finite set of Production Rules (A→α) where A∈V and α∈ (V∪T)*.**

As an example, we have the following grammar:-

G = ({ E, T, F },{ (,) , a , + , *}, E , P)

Where P is given by:-

P={ E→T
E→E+T
T→F
T→T * F
F→a
F→(E) }

And to save space we may combine all the rules with the same left hand side, i.e.,

P={ E→T/E→E+T
T→F/T * F
F→a/(E) }

❖ ***Lexical Analyzer*** **:- Its main task is to read the source program (character by character) then translated into a sequence of primitive units called *tokens* like (keywords, identifier, constant, operators, etc.).**

Lexical Analyzer reads the source program character by character and returns the *tokens* of the source program. A *token* describes a pattern of characters having same meaning in the source program. (Such as identifiers, operators, keywords, numbers, delimeters and so on), puts information about identifiers into the symbol table.

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

⇛ **Tokens:**

newval	**identifier**
:=	**assignment operator**
oldval	**identifier**
+	**add operator**
12	**a number**

❖ ***Syntax Analyzer*** **:- A Context Free Grammar, CFG, (synonyms: Backus-Naur Firm of BNF) is a common notation for specifying the syntax of a languages. The syntax of a language is specified by a context free grammar (CFG). The rules in a CFG are mostly recursive. A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not. If it satisfies, the syntax analyzer creates a parse tree for the given program.**

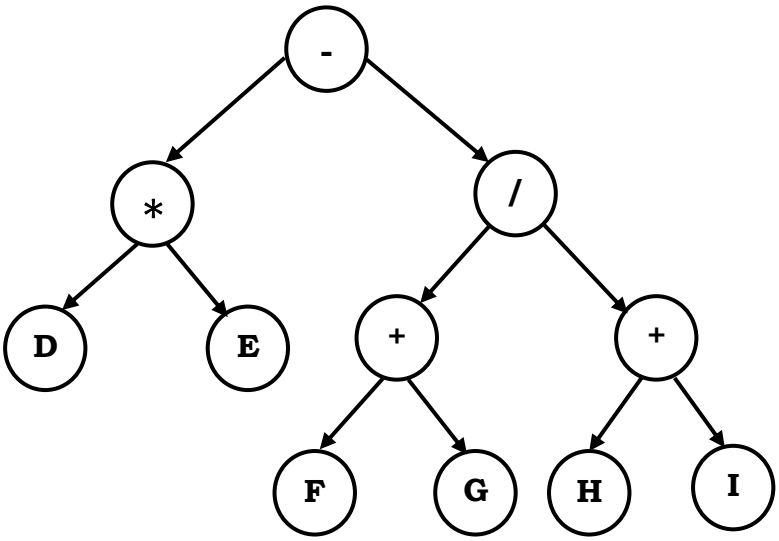
Therefore, Syntax analysis is applied by a compiler to check the syntax of a program by constructing a parse tree of the program.

Syntax Analysis generates a Parse Tree as shown below:-

***Example* 1:- If we have the following arithmetic expression:-**

$$M= (D*E) – ((F + G) / (H + I))$$

Then the parse tree for this expression will be as below:-



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

$$M= ((A - C) * B) – (A - B) / (A + (A - C) * B)$$

How to reconstruct the syntax parse tree?

❖ ***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. This attempts to catch programming errors based on the theory of types. In practice this is checking things like a variable declared as a string is not used in an expression requiring an integer.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 Pascal 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.**
   - **Check labels**
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 form 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*** **:- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space. This code optimization phase attempts to improve the intermediate code, so the faster-running machine code will result.**

Example **1:- newval =oldval + 12 This will means (id1 := id2 + 12)**

In code optimizer we have the following codes:-

```
ADD     id2, #12, temp1
MOV     temp1, id1
```

Example 2 **:- position := initial + rate * 60**

This will means (id1 := id2 + id3 * 60)

In code optimizer we have the following codes:-

```
temp1  :=  inttoreal (60)
temp2  :=  id3 * temp1
temp1  :=  id3 * 60.0
temp3  :=  id2 + temp2
id1    :=  id2 + temp1
id1    :=  temp3
```

❖ ***Code Generator*** **:- The final phase of the compiler is the generation of target program, the target program is normally a relocatable object file containing the machine codes.**

For example, having the following expressions:-

a := b + c

d := a + e

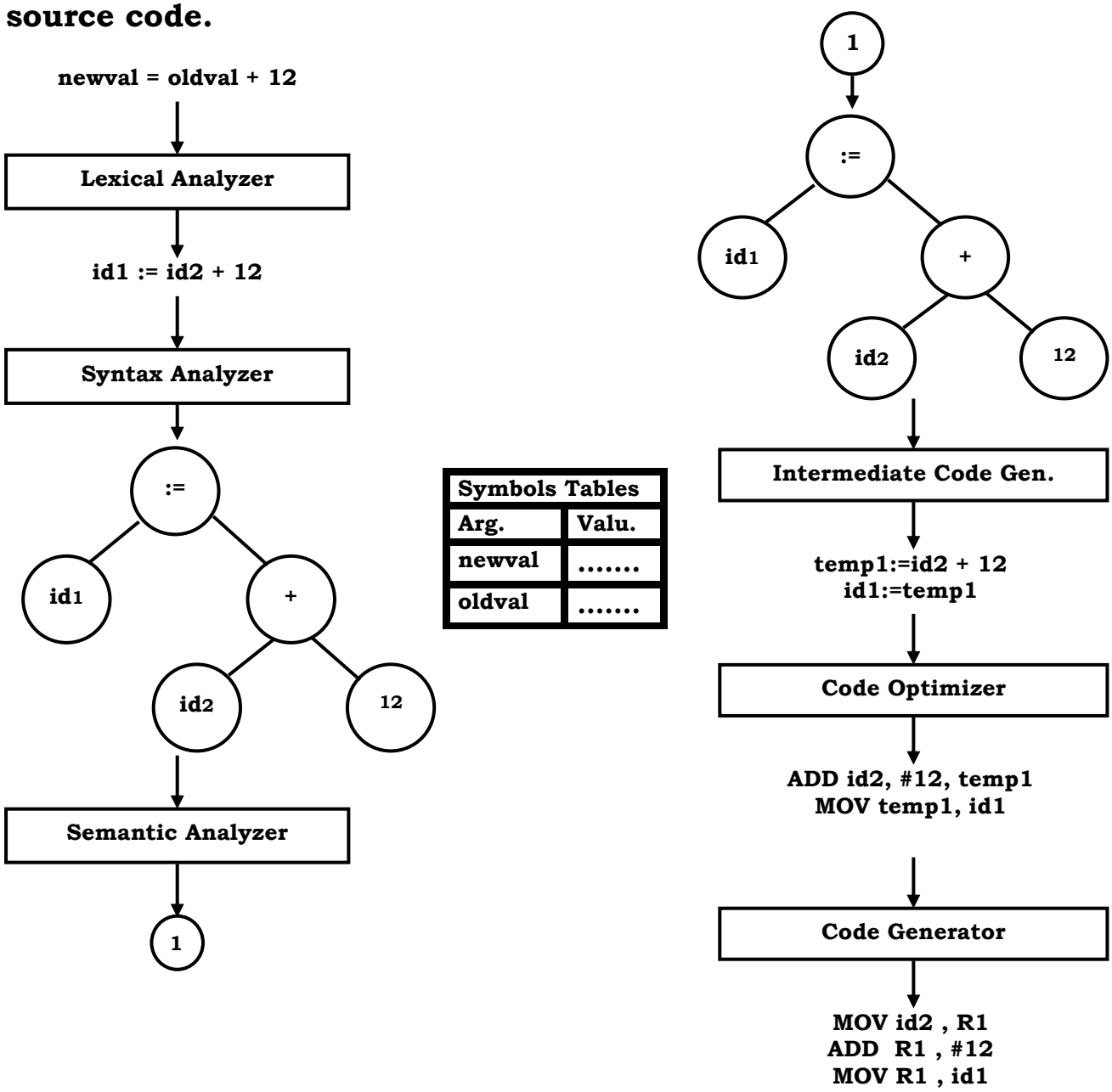
Inefficient assembly code is:

```
MOV  b , R0      R0  ← b
ADD  c , R0      R0  ← c + R0
MOV  R0 , a      a   ← R0
MOV  a , R0      R0  ← a
ADD  e , R0      R0  ←e + R0
MOV  R0 , d      d   ← R0
```

Example **:- having the following expression <u>newval =oldval + 12</u>**

The whole running of each phase of the compiler phases are shown in the following diagram:-

Now, <u>newval = oldval + 12</u> represents the source program or source code.



جامعة بغداد
كلية التربية للعلوم الصرفة ابن الهيثم
قسم علوم الحاسبات

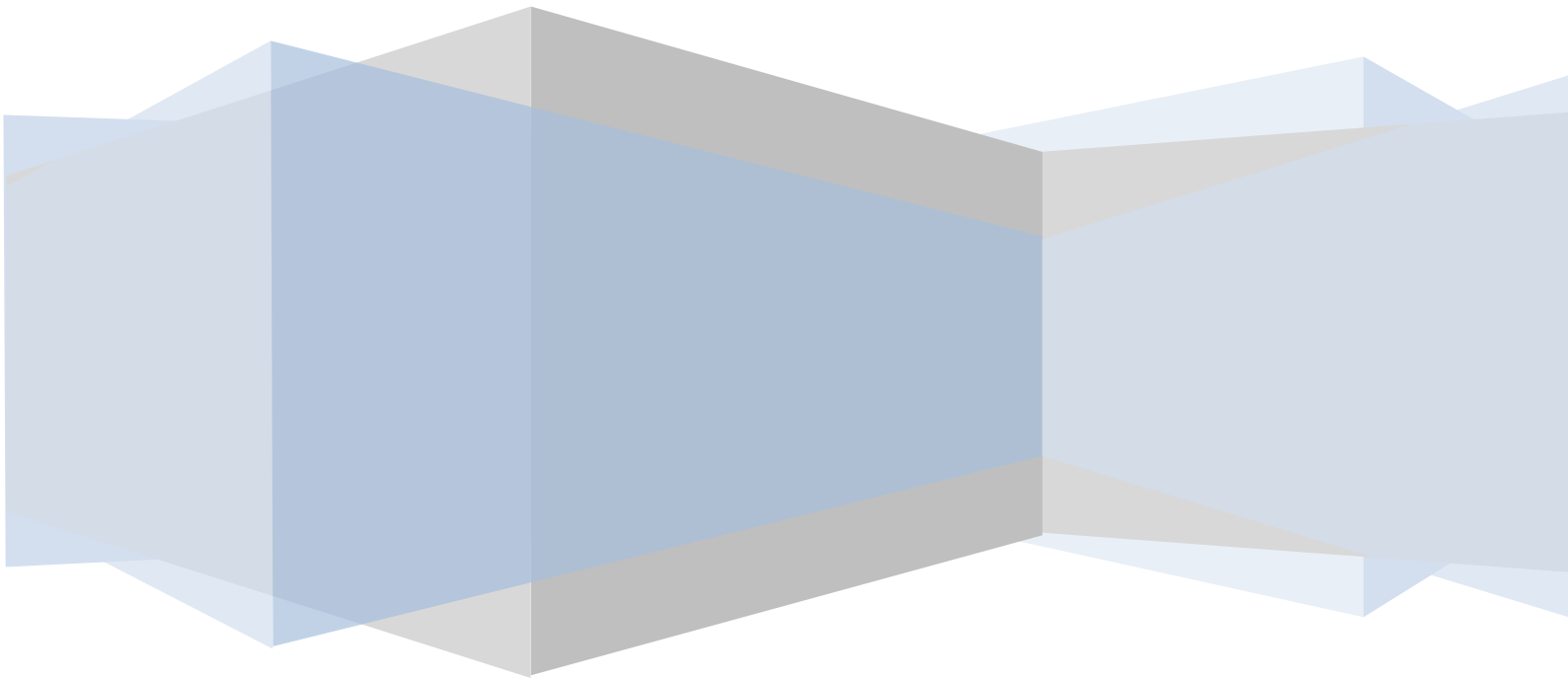
Compilers

Code optimizer

Third stage

M.Sc. Ahmed Rafid

2016-2017



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. A 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);
```

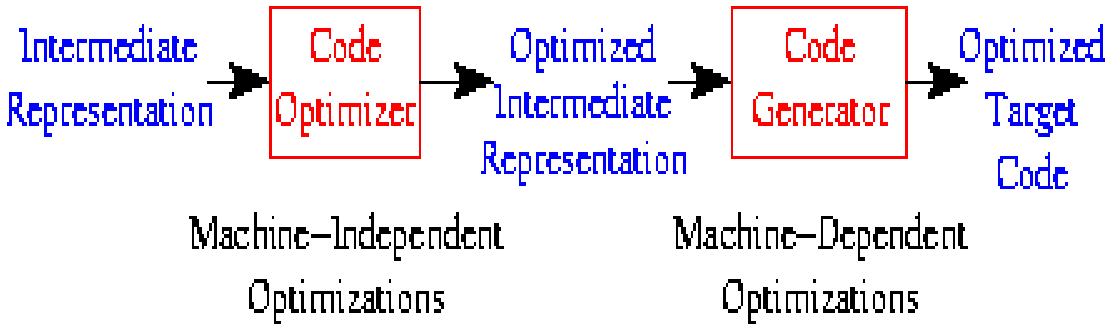
This code involves repeated assignment of the identifier item, which if we put this way:

```
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.



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

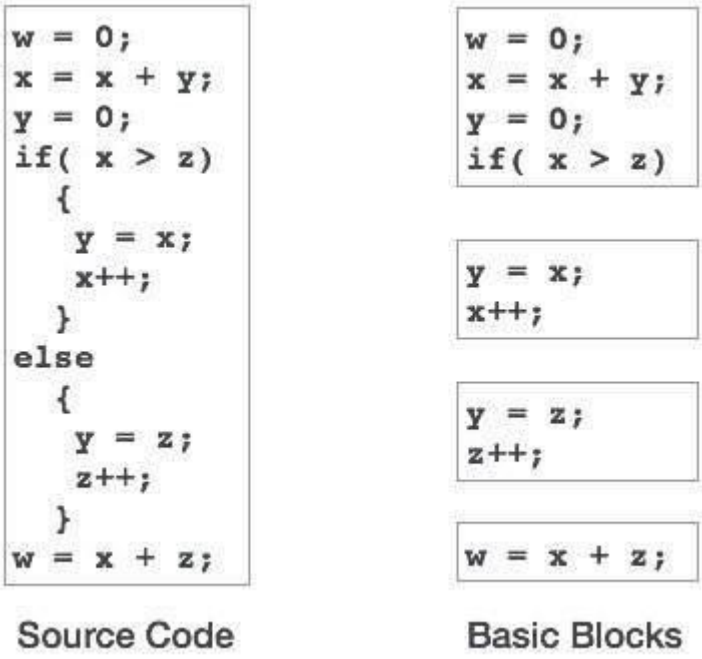
Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will 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.

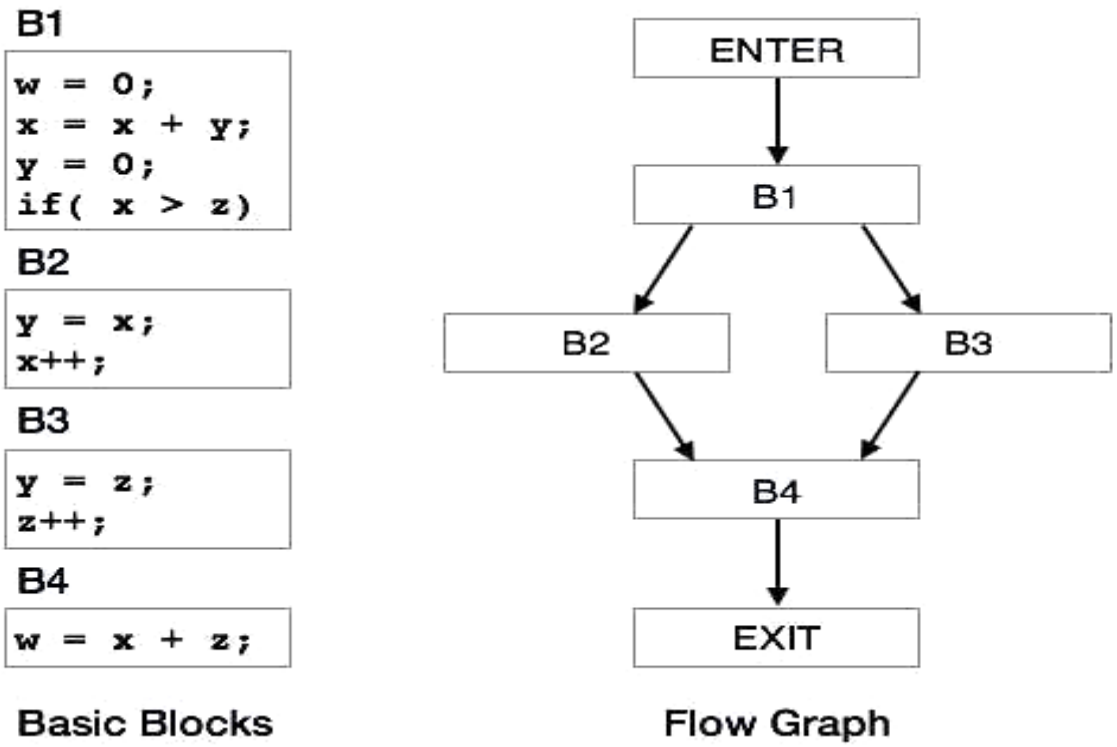
Local Optimizations are performed on basic blocks of code

Global Optimizations are performed on the whole code



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.



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

جامعة بغداد
كلية التربية للعلوم الصرفة ابن الهيثم
قسم علوم الحاسبات

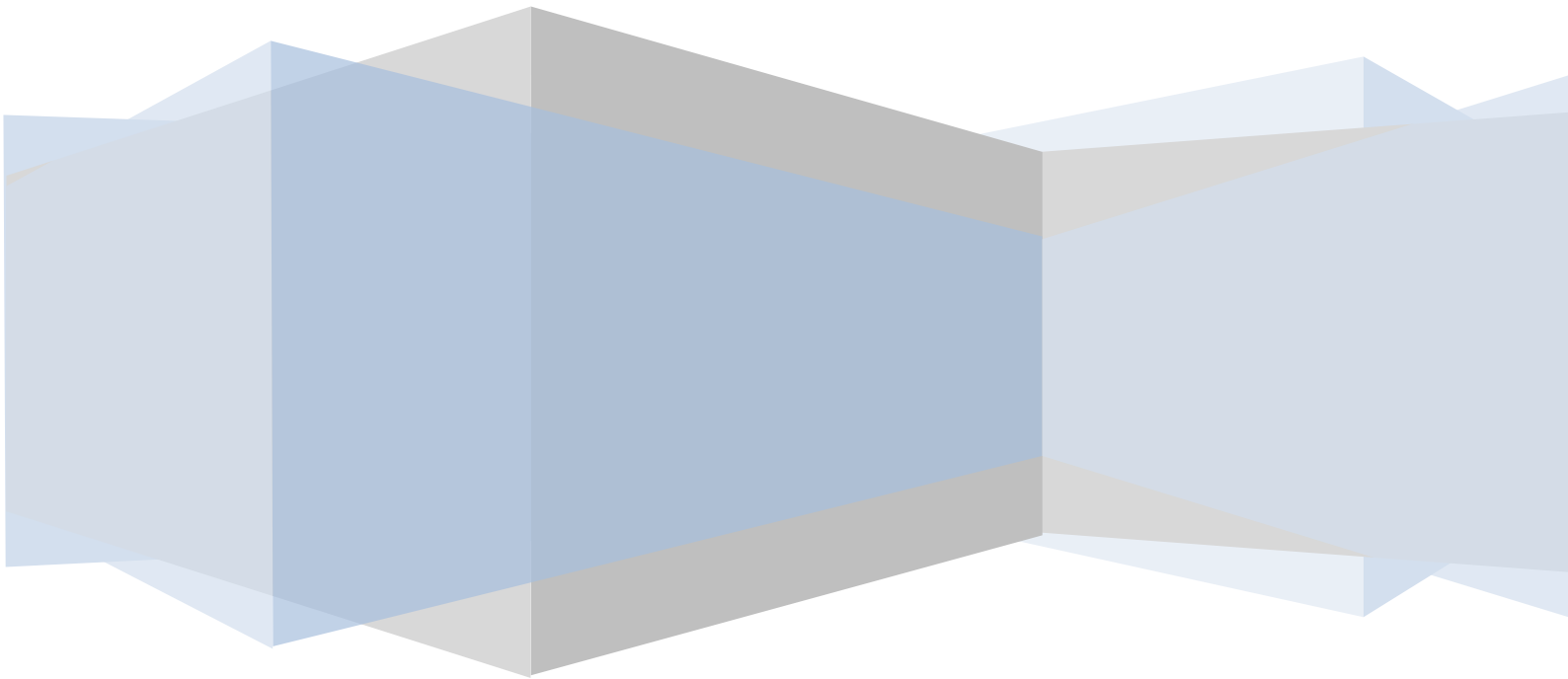
Compilers

Code Generation

Third stage

M.Sc. Ahmed Rafid

2016-2017



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).

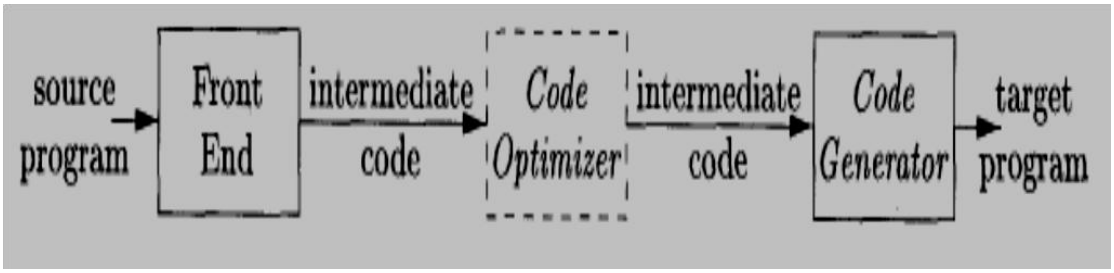


Figure 1: position of Code generation

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.