Grammar :

The formal definition of the syntax of a programming language is usually called 'Grammar'.

A Grammar consist of a set of rules (productions) the specify the sequence of characters that form allowable program in the language being defined.

A grammar specified using a strictly defined notations is called formal grammar.
two grammars are equivalent if they produce the same language

Regular Grammar :

* regular grammar are special cases of BNF grammar that turn out to be equivalent to the Finite state automata.

ex- grammar for binary string with last digit 0

$$A \rightarrow 0A \mid 1A \mid 0$$

Regular Expression-

Regular expressions represents a form of language definition that is equivalent to FSA and regular grammar.

* A regular expression is a special text string for describing a search pattern.
we use regular expression to describe token of a programming language.

A regular exp. recursively as—

- individual terminal symbols are regular expression.
- if a and b are regular exp. then

  $a|b$, ab, (a), a^*

  are regular exp.

Arden's theorem :—

To find a regular expression of a finite automata Arden's theorem is used.

Steps :—

① Let P and Q be two regular expressions

② if P does not contain null string then

$R = Q + RP$ has a unique solution

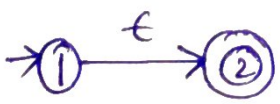
$\Rightarrow R = QP^*$

~~Example :—~~

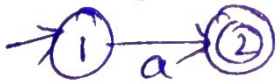
Thomson Construction :— (find NFA from regular exp.)

rules—

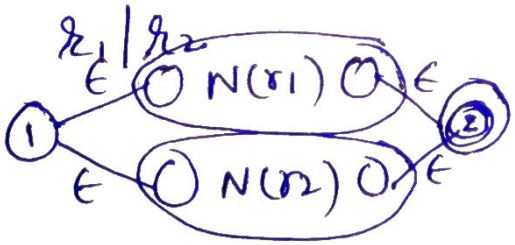
① for ϵ



② for a symbol in the alphabet Σ



③ for $r_1 | r_2$



④ for a b

⑤ for a^*

~~REGULAR GRAMMAR :—~~

⇒ Regular exp^n for set of strings of 0's and 1's not containing 101 as substring

$$r = (0^*1^*00)^*0^*1^*$$

⇒ Reg. exp^n over {a,b} for strings with even no. of a's followed by odd no. of b's.

$$\{L = a^{2n}b^{2m+1} \; ; \; n \geq 0, m \geq 0\}$$

$$r = (aa)^*(bb)^*b$$

⇒ Reg. exp^n for language $L = \{w \in \{0,1\}^* : w$ has no pair of consecutive zeros$\}$

(any occurance of 0 must be followed by 1

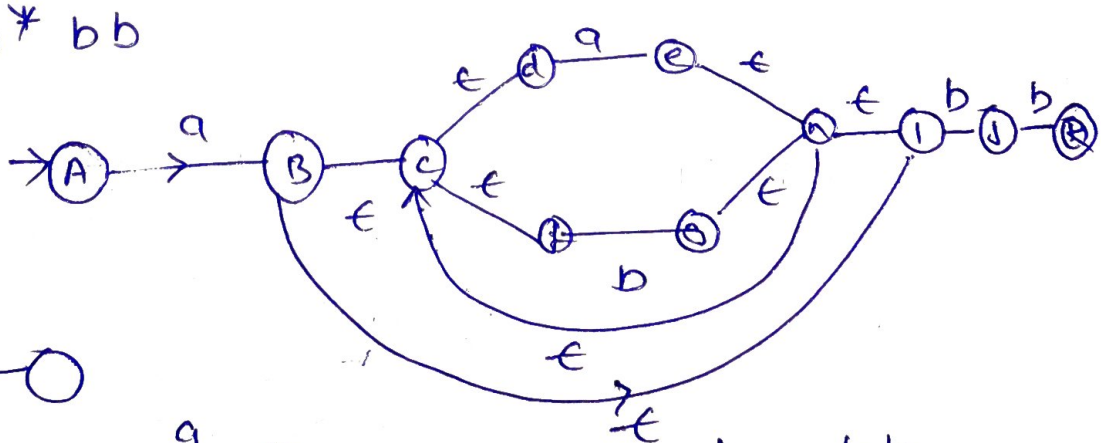
thus

$$\text{exp } r = (1+01)^*(0+\epsilon)$$

✱ Construction of FA for regular exp^n.

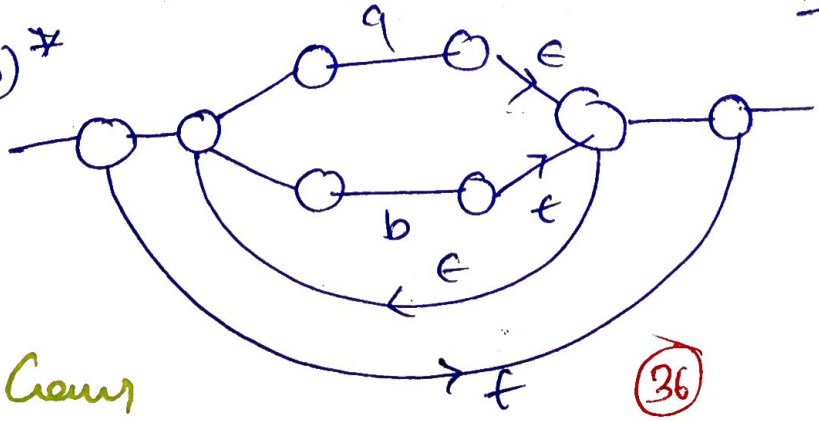
● $a(a+b)^*bb$

DFA ⇒

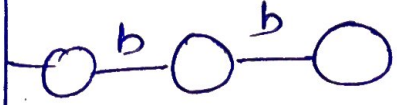


for a

for $(a+b)^*$



for bb



DAA – (Deterministic finite Automata)

$M = (Q, \Sigma, \delta, f, q_0)$

Q = non-empty set of finite states present in the finite control.

Σ = non-empty set of i/p symbol which can be passed to the finite state machine.

δ = transition function
(takes two argument – a state and a i/p symbol) and return a single state.

f = non-empty set of final states.

q_0 = starting state or initial state.

- DFA – finite acceptor.

Language of DFA →

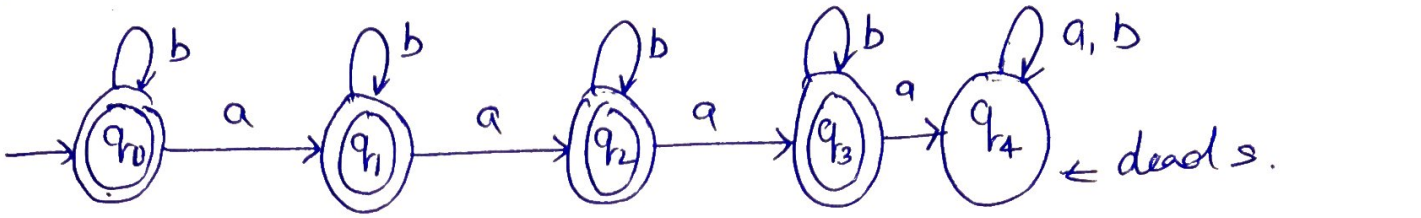
$$LM = \{ w$$

set of all strings(w) that takes the start state q_0 to one of the accepting states.

Dead State

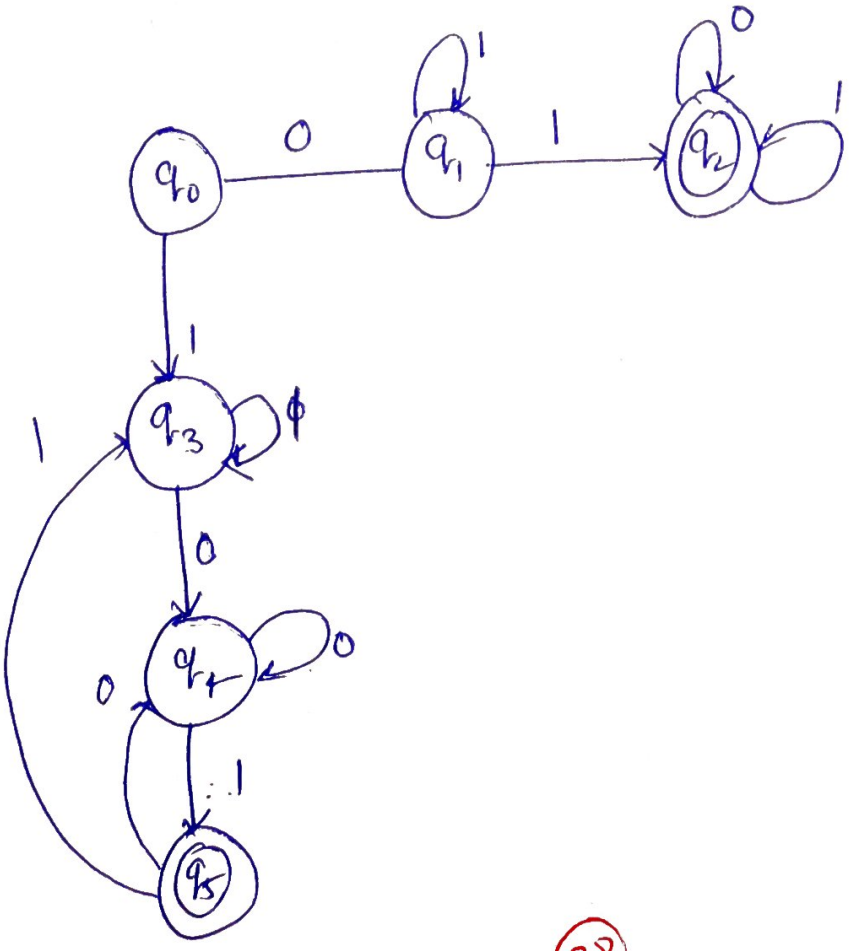
Dead state are those non-final states which transit in themselves for all the i/p symbol

ex -



$\epsilon, a, aa, aaa, b, bb, bbb \dots ba, baba \dots babab \dots$

ex - Design FA over $\Sigma=(0,1)$, which accepts the set of strings either start with 01 or end with 01.



Non Determination → NDFA →

Non-determination means a choice of moves for an automata, rather than prescribing a unique move in each situation.

definition –

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q = is a non-empty finite set of

Σ = non-empty finite set of i/p symbols

q_0 = initial state

F = non-empty finite set of final state.

δ = transition funct^{n}.
takes a state from Q and i/p symbol in Σ as arguments and returns a subset of Q.

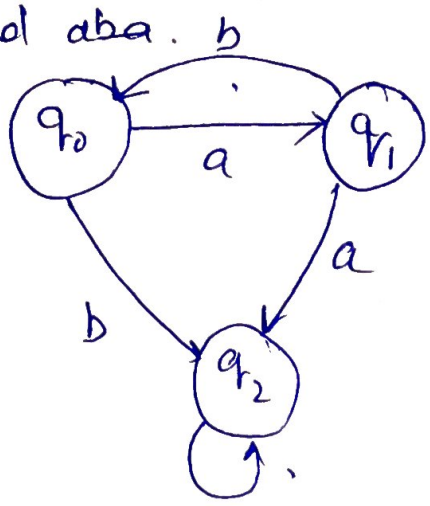
$$Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$$

So in non-deterministic accepter, the range of δ is the power set 2^Q. (not a single element of Q but a subset of it)

ex: NFA for L = all string over $\{0,1\}$ having at least two consecutive 0's & 1's

NFA for $L = (ab \cup aba)^*$

means either ab, aba or any combination of ab and aba.



★ Converting NFA to DFA :-

(Subset Construction Algo.)

put ϵ closure ({s0} as an unmarked state into the set of DFA (DS)

while (there is one unmarked S1 in DS) do

begin

mark S1

for each input symbol a do

begin

S2 ← ϵ closur (move (S1, a))

if (S2 is not in DS) then

add S2 into DS as an unmarked state

transfunc [S1, a] ← S2

end

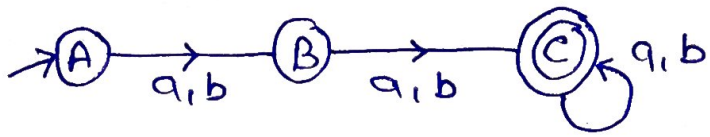
end

ϵ-closure → ϵ-closur() function takes a state and return the set of states reachable from it based on ϵ-transitions. (always include the state itself)

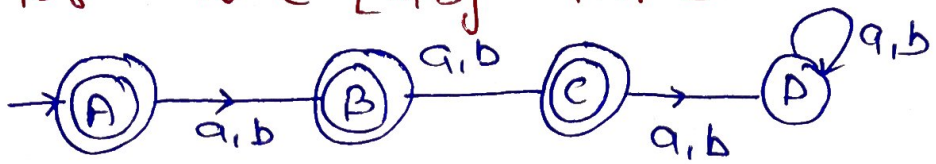
move() → The function move takes a state and a character and return the set of states reachable by one transition on this character.

Pragya Gaur

⇒ Construct DFA for $w \in \{a,b\}$ $|w| \geq 2$



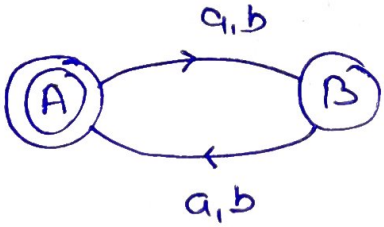
⇒ DFA for $w \in \{a,b\}$ $|w| \leq 2$



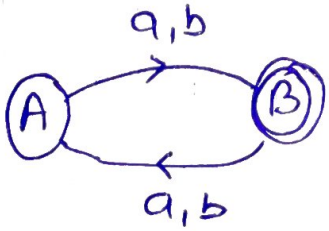
✡ for a minimal finite Automata

for length	n	length ≥ 2	length ≤ 2
no. of state	(n+2)	(n+1)	(n+2)

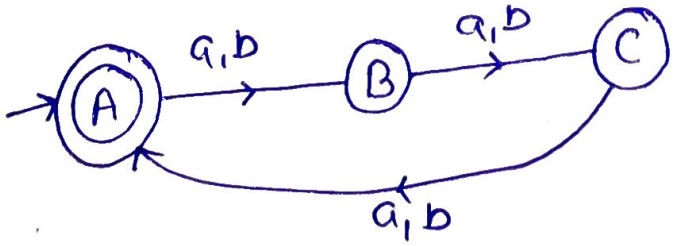
⇒ DFA for $w \in \{a,b\}$ $|w| \bmod 2 = 0$
(even length string)



⇒ DFA for $w \in \{a,b\}$ $|w| \bmod 2 = 1$
(odd length string)

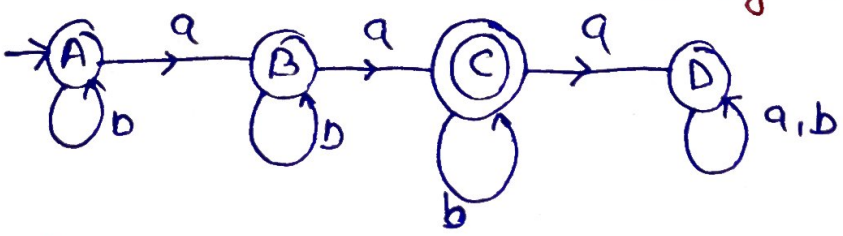


⇒ $|w| \bmod 3 = 0$ (~~not mod~~ $|w| \equiv 0 \bmod 3$)
(length should be multiple of 3

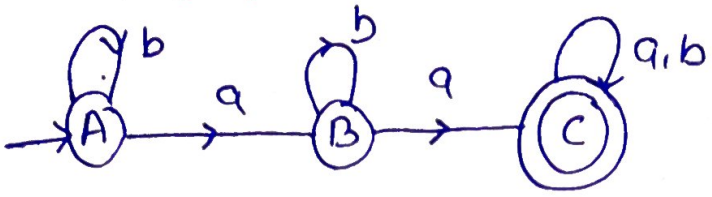


⇒ DFA for $w \in \{a,b\}$ $n_a(w) = 2$

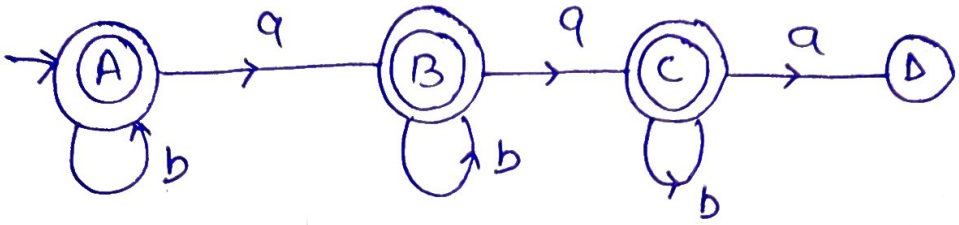
(no. of a in string is 2)



$n_a(w) \geq 2$



$n_a(w) \leq 2$



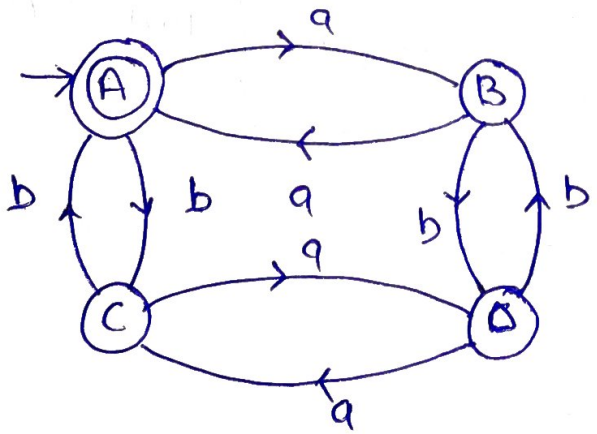
DFA for $w \in \{a,b\}$

$n_a(w) \equiv 0 \bmod 2$

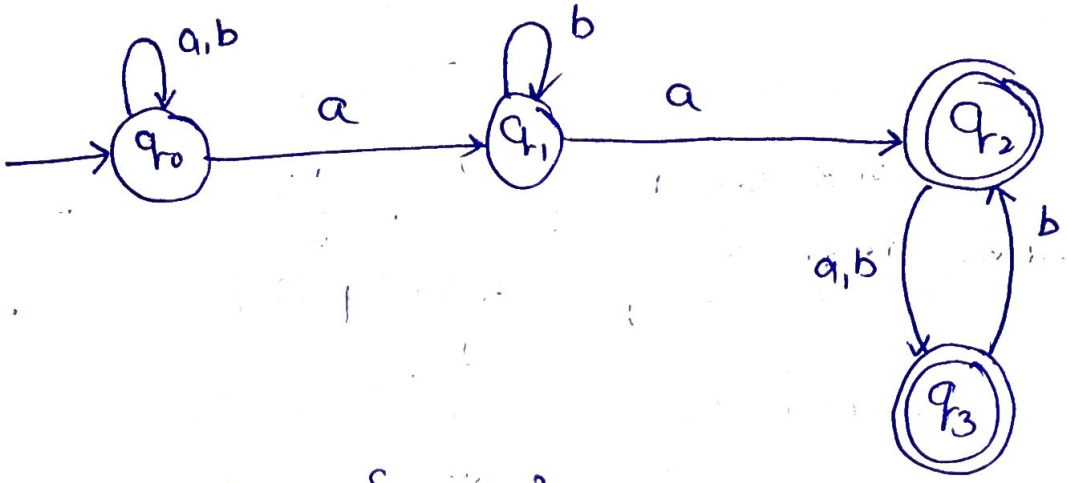
&&

$n_b(w) \equiv 0 \bmod 2$

(a and be are even)



Transformation of NFA to DFA (Subset Construction)



$\delta(q_0, a) = \{q_0, q_1\}$

$\delta(q_0, b) \neq q_2$

$\delta(\{q_0, q_1\}, a) = \delta(q_0, a) \cup \delta(q_1, a)$

$= (q_0, q_1) \cup (q_2)$

$= \{q_0, q_1, q_2) =$

$\delta((q_0, q_1), b) = \delta(q_0, b) \cup \delta(q_1, b)$

$= (q_0) \cup q_1$

$= (q_0, q_1)$

$\delta((q_0, q_1, q_2), a) = (q_0, q_1, q_2, q_3) =$

$\delta(q_0, q_1, q_2, b) = (q_0, q_1, q_3) =$

$\delta((q_0, q_1, q_2, q_3), a) = q_0 q_1 q_2 q_3$

$(\quad, b) = q_0 q_1 q_2 q_3$

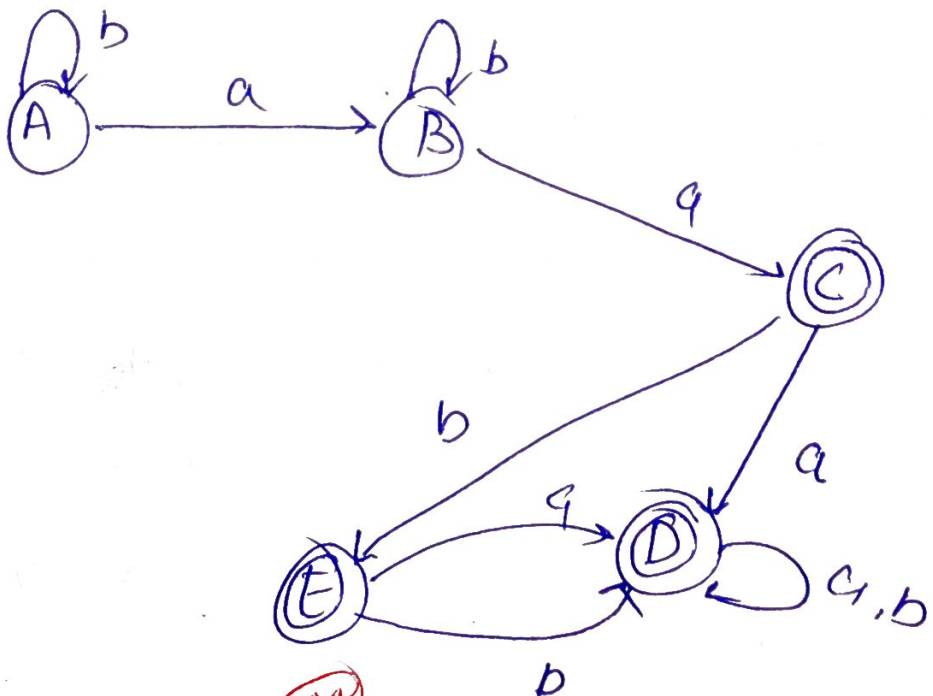
$\delta((q_0\ q_1\ q_3), a) = q_0\ q_1\ q_2\ q_3$

$b) = q_0\ q_1\ q_2\ q_3$

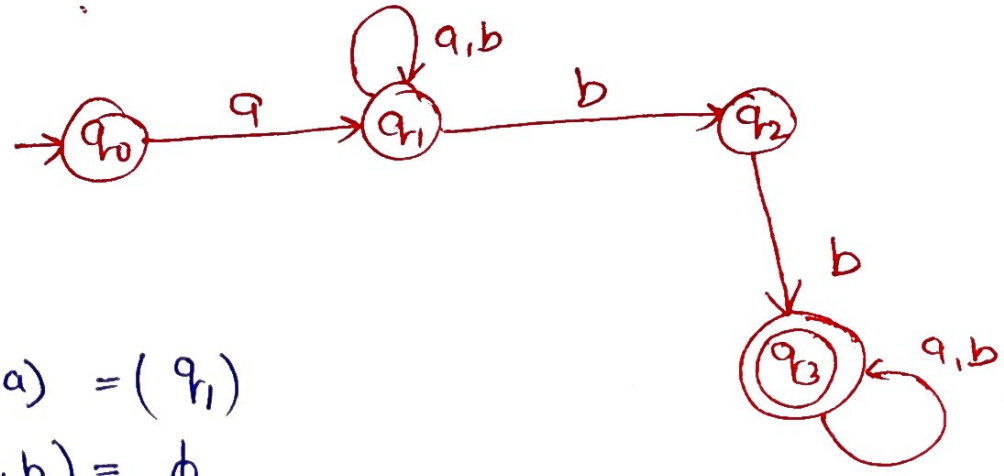
	δ/Σ	a	b
A	$\to q_0$	(q_0, q_1)	q_0
B	(q_0, q_1)	$q_0\ q_1\ q_2$	$q_0\ q_1$
C *	(q_0, q_1, q_2)	$q_0\ q_1\ q_2\ q_3$	$q_0\ q_1\ q_3$
D *	(q_0, q_1, q_2, q_3)	$q_0\ q_1\ q_2\ q_3$	$q_0\ q_1\ q_2\ q_3$
E *	(q_0, q_1, q_3)	$q_0\ q_1\ q_2\ q_3$	$q_0\ q_1\ q_2\ q_3$

δ/Σ	a	b
A	B	A
B	C	B
C	D	E
D	D	D
E	D	D

DFA



example-2



$\delta(q_0, a) = (q_1)$

$\delta(q_0, b) = \phi$

$\delta(q_1, a) = (q_1)$

$\delta(q_1, b) = (q_1, q_2)$

$\delta((q_1, q_2), a) = \delta(q_1, a) \cup \delta(q_2, a)$

$= (q_1) \cup \phi$

$= (q_1)$

$\delta((q_1 q_2), b) = \delta(q_1, b) \cup \delta(q_2, b)$

$= (q_1 q_2) \cup (q_3)$

$= (q_1 q_2 q_3)$

$\delta((q_1 q_2 q_3), a) = \delta(q_1, a) \cup \delta(q_2, a) \cup \delta(q_3, a)$

$= (q_1) \cup (q_3)$

$= (q_1, q_3)$

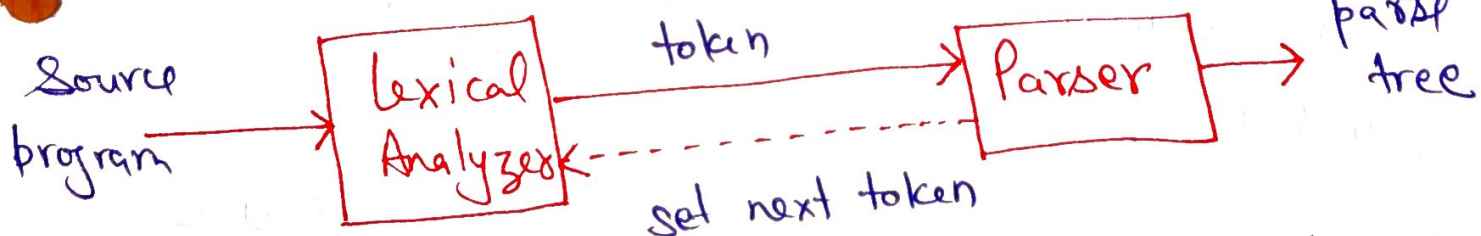
$\delta((q_1 q_2 q_3), b) = \delta(q_1, b) \cup \delta(q_2, b) \cup \delta(q_3, b)$

$= (q_2 q_2 q_3)$

Syntax Analysis :-

In the second phase of compiliation the syntax Analyzer creates the syntactic structure of the given source program.
this syntactic structure is mostly a parse tree.
So syntax analyzer is also known as "parser".
The syntax of a programming is described by a context free grammar.
The syntax analyzer checks whether a give source program satisfy the rules implied by CFG or not-
if it satisfy, the parser creates the parse tree of that program otherwise the parser gives the error message.

Parser :-



parser works on stream of tokens, and can be categorized into two groups-

1) Top down parser
2) bottom up parser

Priyya Gaur

CONTEXT FREE GRAMMAR :-

"Context free grammar are a notation for specifying concrete syntax."

(Concrete syntax of a language describes its written representation, including lexical details)

(lexical syntax specifies the correspondence b/w the written representation of the language and the token in a grammar.

"Inherently recursive structure of a programming language are defined by a context free grammar."

CFG consist -

① A finite set of terminals. (atomic symbols in the language.)

② A finite set of Non-terminals (variables representing constructs in the language)

③ A finite set of production rules (to identify the component of a construct)

$A \rightarrow \alpha$

where A represent non-terminal
α represent a string of terminals & Non-Ter.

④ A start Symbol. (Non-terminal)

~~CFG = {V, E, R, S}~~

CFG = { N, T, P, S }

CFG for string containg only a

N = {S}

T = {a}

P = { $S \Rightarrow_S aS$, $S \rightarrow \epsilon$ }

BNF Grammar :-

Backus Nour form is one notation used to write grammar.

rules :-

::= ⇒ is defined as

| ⇒ or

<> ⇒ Non-terminals are enclosed b/w them.

example-

A empty string can be denoted as- <empty>

* <expression> ::= <expression> + <term>
|<expression> - <term>
| <term>

E ::= E+T | E-T | T

EBNF :- (Extended Backus Nour form)

to extend BNF the following notantional extensions are used to allow easier description of language.

rules :-

① An optional element may be indicated by enclosing the element in square brackets [].

② An arbitrary sequence of instances of an element may be indicated by enclosing the element in braces followed by an astrisk

{}*

Context free grammer !.

TOKEN! The syntax of a programming language is specified in terms of units called token / terminals.

Lexical Syntax:- lexical syntax specifies the correspondence b/w the written representation of the language and the token / terminal in a grammer for the language

token - +, - <=

The concrete syntax of a language describes its written representation, including lexical details

" Context free grammer are a notation for specifying concrete syntax."

BNF is a way of writing grammer.

A CFG has four parts

1) set of terminals. (atomic symbols in the language)
2) set of non terminal (variables representing constructs in the language)
3) set of rules called production to identify the component of a construct.
4) starting non terminal.

CFG - {V E R S}

T.

ex:- CFG for string - containing only a

$N = \{S\}$

$T = \{a\}$

$P = \{S \to aS, \; S \to \epsilon\}$

$S = S$

BNF Grammer:

BNF (Backus-Naur-form) is one notation used to write grammer.

As we know CFG, consisting of terminals, non terminals production and a starting Non-terminal is independent of the notation used to write grammer.

example: BNF Grammer for real number -

→ ::= ⇒ ~~Can be~~ / is defined as

| ⇒ or

<> ⇒ Non terminal are enclosed b/w them.

ex- empty string is written as <empty>

<real number> ::= <integr-part> . <fraction>

<integer-part> ::= <digit> | <integer-part> <digit>

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

<digit> ::= 0 | 1 | 2 | 3 | . . . | 9

Extended BNF notation:

to extend BNF the following notational extensions are used to allow easier description of language.

- An optional element may be indicated by enclosing the element in square brackets [ ]

ex- BNF –

<signed integer> ::= + <integer> | – <integer>

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

<signed Integer> ::= [+|-] <digit> {<digit>}*

- An arbitrary sequence of instances of an element may be indicated by enclosing the elements in braces followed by an astrisk, { }*

<identifier> ::= <letter> {<letter> | <digit>}*

choice |

optional construct [ ]

Repetation { } or { }*

Grouping ( )

termination or separation ( )

example -

* BNF <statment-list> ::= <empty> | <statment>; <statment-list>

EBNF <statment-list> ::= { <statment>; }*

* BNF <real no> ::= <integer part>.<fraction> | <fraction>

EBNF <real no> ::= [<integer part>]. <fraction>

BNF:

<expression> ::= <expression> + <term>
| <expression> - <term>
| <term>

E ::= E + T | E - T | T

<term> ::= <term> * <factor>
| <term> / <factor>
| <factor>

T ::= T * F | T / F | F

<factor> ::= num | name | (<expression>)

F ::= Num | Name | (E)

EBNF:

<expression> ::=

<term> [(+|-) <term>]*

<term> ::= <factor> [(*|/) <factor>]

<factor> ::= '(' <expression ')' | name | num

EBNF:- (for assignment statment)

<assignment statment> ::= <variable> = <arithmetic expression>

<arithmetic expression> ::= <term> { [+|-] <term> }*

<term> ::= <primary> { [x | /] <primary> }*

<primary> ::= <variable> | <number> | (<arithmetic expression>)

<variable> ::= <identifier> | <identifier> [<subscript list>]

<subscript list> ::= <arithmetic expression> { , arithmetic expression> }*

BNF:

<assignment statment> ::= <variable> = <arithmetic expression>

<arithmetic Expression> ::= <term> | <arithmetic expression> + <term>
| <arithmetic expression> - <term>

<term> ::= <primary>
| <term> * <primary>
| <term> / <primary>

<primary> ::= <variable> | <number> |
| (<arithmetic expression>)

<variable> ::= <identifier> |

<subscript list> ::= <arithmetic expression> | <subscript list> , <arithmetic expression>

Removel of ϵ production –

to eliminat ϵ production forom G –

" if $A \to \epsilon$ is a production, then consider all productions having A on right side and replace each occurence of A in each of these productions to obtain the non-ϵ productions

and add these non-ϵ productions to the grammar to keep the language generated, the same.

ex. $S \to aA$

$A \to b \mid \epsilon$

replac A by ϵ in $S \to aA$

$\Rightarrow S \to a$

$\Rightarrow S \to aA$

$S \to a$

$A \to b$

ex $S \to ABAC$

$A \to aA \mid \epsilon$

$B \to bB \mid \epsilon$

$C \to C$

for $A \to \epsilon$

$S \to ABAC$ becom

$S \to ABC$

$\mid BAC$

$\mid BC$

$A \to a$

After Removel of $A \to \epsilon$

G:

$S \to ABAC \mid ABC \mid BAC \mid BC$

$A \to aA \mid a$

$B \to bB \mid \epsilon$

$C \to c$

for $B \to \epsilon$

$S \to AAC \mid AC \mid C$

$B \to b$

thus after removel of

$A \to \epsilon$

$B \to \epsilon$

G: $S \to ABAC \mid ABC \mid BAC$

$\mid AAC \mid AC \mid BC \mid C$

$A \to aA \mid a$

$B \to bB \mid b$

Removel of Unit Production :-

A production of the form

Non-terminal → One Non-terminal

is called unit production.

⇒ for a unit production $A \to B$, such that there exist a production $B \to a$, where a is a terminal.

for every non-unit production $B \to a$

Add $A \to a$ to the grammar

eliminate $A \to B$ from the grammar.

ex-

$S \to AB$

$A \to a$

~~$B \to \epsilon$~~

$B \to C \mid b$

$C \to D$

$D \to a$

unit productionare

$B \to C$

$C \to D$

we can remore $C \to D$ using $D \to a$

thus

$C \to D$

$D \to a$

Can be replacd by

$C \to a$

$D \to a$

$B \to C$ can be remove by $C \to a$

$\Rightarrow$ $B \to C$, $C \to a$ $\Rightarrow$ $B \to a$, $C \to a$

$\Rightarrow$

$S \to AB$

$A \to a$

$B \to a$

$B \to b$

$C \to a$

$D \to a$

as we can see C and D are useless as

$C \to a$

$D \to a$

will never be used

thus

$\Rightarrow G'$:

$S \to AB$

$A \to a$

$B \to a \mid b$

✱ $S \to \alpha$

→ if α contains non-terminals, it is called as a sentential form of G.

→ if α contains terminal (does not contain non-terminal) it is called as a sentence of G.

Derivations :

Consider the set of production

$E \to E+E \mid E-E \mid E*E \mid E/E \mid -E$

$E \to (E) \mid id$

→ $E \to E+E$ means that $E+E$ derives from E

$E \to E+E \Rightarrow id+E \to id+id$

means that a sequence of replacement of non-terminal symbols is called a derivation of $id+id$ from E.

at each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement always

✱ if we choose the left most non-terminal in each step, this derivation is called left most derivation.

ex-

$E \Rightarrow -E \to -(E) \to -(E+E) \to -(id+E)$

$\downarrow$

$-(id+id)$

* if we always choose the right most terminal in each derivation of the step, this derivation is called as right most derivation.

ex-

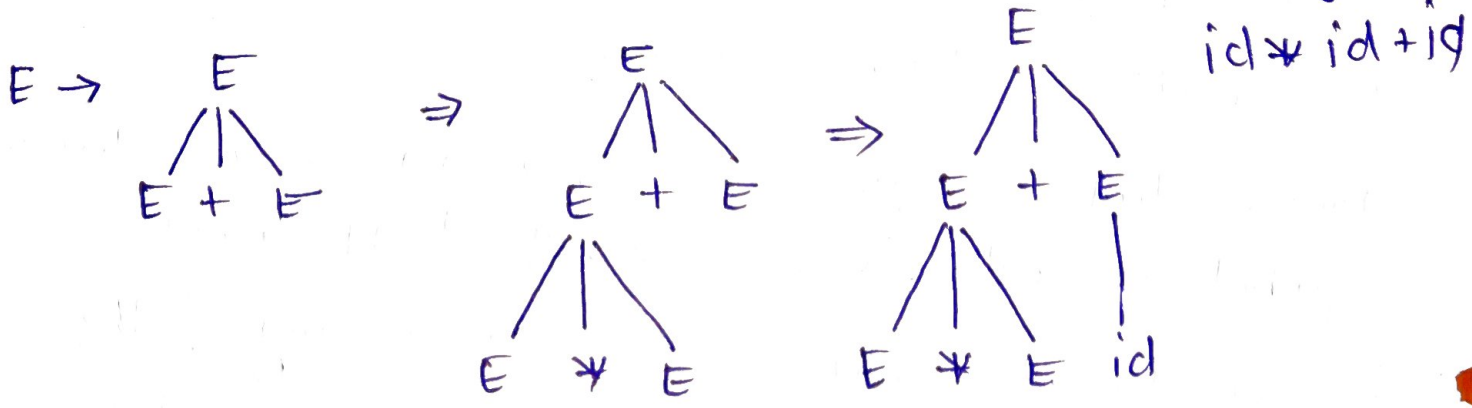
$$E \to -E \to -(E+E) \to -(E+id) \to -(id+id)$$

Parse tree:

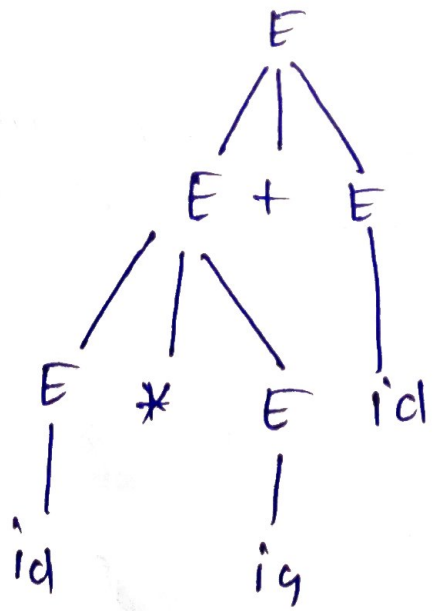
parse tree is the graphical representation of derivation.

the inner nodes of a parse tree are non-terminal while the leaves are terminal symbols.

$$E \to E+E \to E*E+E \to id*E+E \to id*id+E \to id*id+id$$



Le



Left Recursion:

A grammar is left recursive if it has a non-terminal A such that there is a derivation:

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

top down parsing can not handle left Recursion so we have to convert out left recursive grammar into an equivalent grammar which is not left recursive

immediate left recursion:

if rea left recursion appears in a single step of derivation.

~~Example~~

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

eliminate immediate left recursion

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

ex.

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T * F \mid F$$
$$f \rightarrow id$$

$\Rightarrow$

$$E \rightarrow \underset{A}{E}\underset{A}{+T}\underset{\alpha}{} \mid \underset{\beta}{T}$$
$$T \rightarrow \underset{A}{T}\underset{A}{*F}\underset{\alpha}{} \mid \underset{\beta}{F}$$
$$f \rightarrow id$$

after elimination of immediate left recursion Gramar is—

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$f \rightarrow id$$

Pragya Gaur

✡ In General-

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \beta_n$$

$$\Downarrow$$

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

✡ by just eliminating the immediate left recursion, we may not get a grammar which is not left recursive. means a grammar cannot be immediate left recursive but it still can be left recursive

example

$$S \to Aa \mid b$$
$$A \to Sc \mid d$$

} indirect left Recursion

this grammar is not immediate left recursive but it is still left recursive (indirect left recursive)

$$S \to Aa \to Sca$$

or

$$A \to Sc \to Aac$$

✡ Algorithm for the elimination of left recursion:

~~for i from 1 to n do {~~

for i ~~from~~ 1 to n do {

Algorithm:

Arrange non terminals in some order: A1 -- An

```
for i from 1 to n
  do
    {
      for j from 1 to n
        do
          {
            replace each production of the form
              Ai → Ajγ|
                by
              Ai → α1γ | α2γ --- | αkγ
                where Aj → α1 | --- αk
          }
      eliminate immediate left recursion
          among Ai productions
    }
```

Consider the examkel –

S → Aa | b
A → Ac | Sd | f

Case 1 – order of non-terminals, S, A

for S
- there is no immediate left recursion
- don't enter the inner loop.

for A
- replace A → Sd with S → Aa | b

A → Ac | Aad | bd | f

Pragya Gaur

remove immediate left recursion

$A \to bdA' \mid fA'$

$A' \to cA' \mid adA' \mid \epsilon$

Resulting Grammar is

$S \to Aa \mid b$

$A \to bdA' \mid fA'$

$A' \to cA' \mid adA' \mid \epsilon$

Case-2:

Order of Non-terminal - A, S

for A

- remove immediate left recursion

$A \to Ac \mid sd \mid f$

$\Downarrow$

$A \to sdA' \mid fA'$

$A' \to cA' \mid \epsilon$

for S

replace $S \to Aa$ with $A \to sdA' \mid fA'$

$\Downarrow$

$S \to SdA'a \mid fA'a \Rightarrow S \to SdA'a \mid fA'a \mid b$

remove immediate left recursion

$S \to fA'as' \mid bs'$

$S' \to dA'as' \mid \epsilon$

resulting Grammar

$S \to fA'as' \mid bs'$

$S' \to dA'as' \mid \epsilon$

$A \to sdA' \mid fA'$

$A' \to cA' \mid \epsilon$

Left factoring: (eliminating non-determination)

Left factoring is a grammar transformation useol to produce the grammar suitable for predictive parsing

if $A \to \alpha\beta_1 | \alpha\beta_2$ are two productions, for string begin with α, we do not know whether to expend A to $\alpha\beta_1$ or to $\alpha\beta_2$, then left fectoring is done.

$$A \to \alpha\beta_1 | \alpha\beta_2$$

$$\Downarrow$$

$$A \to \alpha A'$$
$$A' \to \beta_1 | \beta_2$$

Algorithm:-

for each non terminal A find the longest prefix α common to two or more of it's alternatives.

if $\alpha \neq \epsilon$

replace all the A production

$$A \to \alpha\beta_1 | \alpha\beta_2 | \cdots \alpha\beta_n | \gamma$$

where γ represents all alternatives that do not begin with α

by.

$$A \to \alpha A' | \gamma$$
$$A' \to \beta_1 | \beta_2 | \cdots | \beta_n$$

Pragya Gaur

here A' is the new non-terminal, repeatedly apply this transformation until no two alternatives for a non-terminal have a common prefix

example –

$$S \rightarrow \underbrace{iEtS}_{\alpha_1} \mid \underbrace{iEtS}_{\alpha_2}\ eS \mid \underbrace{a}_{\beta}\ \underbrace{}_{\gamma}$$

$$E \rightarrow b$$

⇓ after left factoring

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

$$A \rightarrow ad \mid a \mid ab \mid abc \mid b$$

⇓

$$A \rightarrow aA' \mid b$$

$$A' \rightarrow d \mid \epsilon \mid b \mid bc$$

⇓

$$A \rightarrow aA' \mid b$$

$$A' \rightarrow d \mid \epsilon \mid bA''$$

$$A'' \rightarrow \epsilon \mid \epsilon$$

Ambiguity:

A grammar produces more than one parse tree for a sentence is called as an ambiguous grammar or we can say -

" an ambiguous grammar is one that produces more than one leftmost derivation or more than one right most derivation for the same sentence.

example-

Consider the Grammer

$E \rightarrow E+E \mid E*E \mid id$ ⇒

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id$

Consider the sentence -

id + id * id

consider the left most derivation

$E \rightarrow E+E$
$\rightarrow$ id $+E$
$\rightarrow$ id $+E*E$
$\rightarrow$ id $+$ id $*E$
$\rightarrow$ id $+$ id $*$ id

$E \rightarrow E*E$
$\rightarrow E+E*E$
$\rightarrow$ id $+E*E$
$\rightarrow$ id $+$ id $*E$
$\rightarrow$ id $+$ id $*$ id