Intermediate Code Generation

- We construct a code which is independent of the machine on which we are going to compile it

position := initial + rate * 60

3-address code

no more than 3 operands

↑

temp1 = inttoreal (60)
temp2 = rate * temp1
temp3 = initial + temp2
position = temp3

} can be reduced further to lesser no. of statements

Code optimization (optimizing intermediate code)

~~temp2 = rate * inttoreal (60)~~
~~temp3 = initial + temp2~~

This phase depends on the platform for which we are designing our compiler

Lexical analyser → depends on language for which we are designing analyzer (compiler)

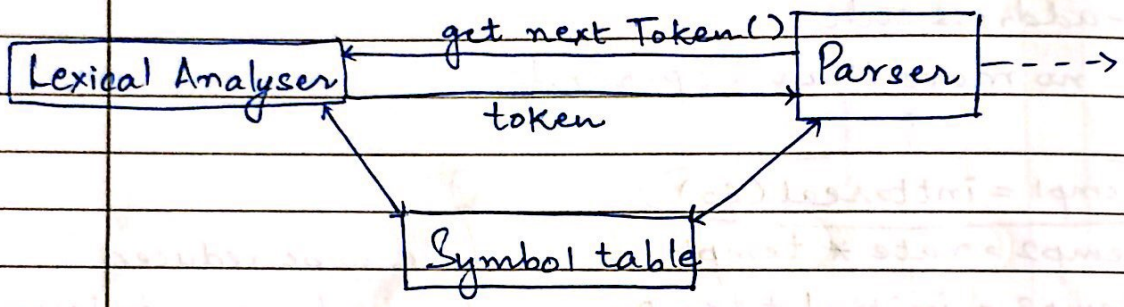
Syntax analyser → depends on language
Semantic analyser → " " language
Intermediate Code Gen → " " language

First 4 phases → front-end of the compiler → dependent ~~of~~ on language

20/08/18

Code optimization } depend upon platform
Code Generation } → back-end of compiler

↳ optimized code converted to machine language




```
int main()
{
    intabc xy_12
}
```

intabc → not a keyword
stores this identifier in the symbol table & returns a pointer to this entry to the parser

xy_12 → underscore not allowed in Pascal
xy12 → underscore hatake aage badho

~~Spanning Code~~ Panic Mode Recovery

Symbol - 0, 1, a, b, A
Alphabet Σ = {0, 1}
Strings - {sequence}

20/08/18

Union: a/b a or b (commutative) (associative)

Concatenation: a.b (⊗ associative)

a*

a+

a.(b/c) = a.b / a.c (Distributive)

Missed a class (24th Aug)

27/08/18

Transition diagram for token relop (relational operators)

<

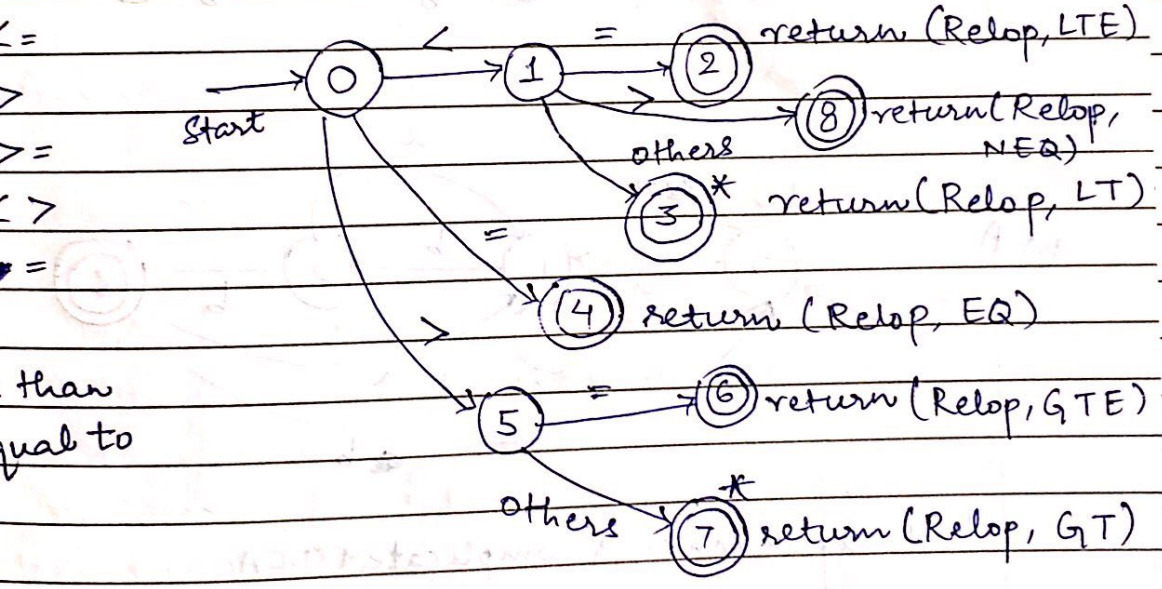
<=

>

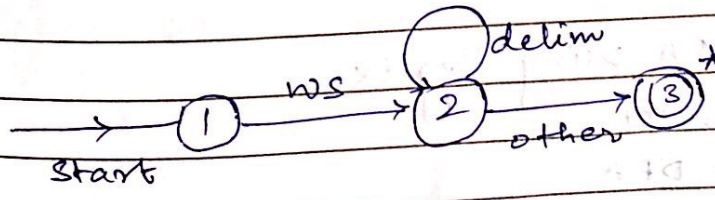
>=

<>

*=



LTE → Less than Equal to



ws → white space, delim → delimiter

delim → newline character, tab etc.

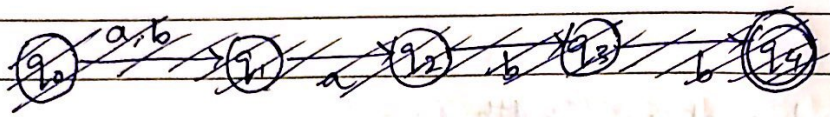
Others mein * isliye taaki backtrack karke others vaala lexeme read kar sakein, otherwise wo read nahi jayega

27/08/18

NFA → $(Q, \Sigma, \delta, q_0, F)$

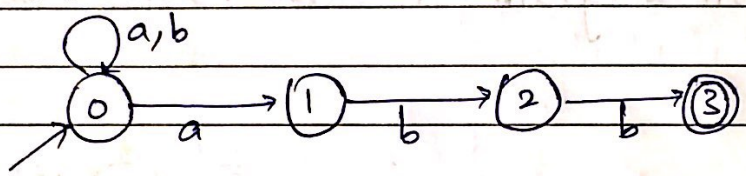
DFA → $(Q, \Sigma, \delta, q_0, F)$

DFA is faster than NFA but takes more space

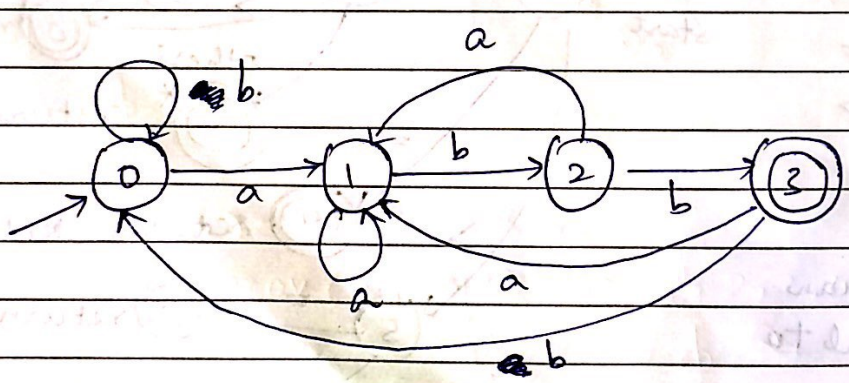


$(a+b)^*abb$

NFA



DFA



Best way to make a complicated DFA

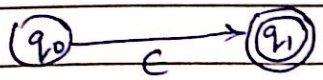
1) Make ϵ-NFA using Reg Ex
2) Convert to NFA
3) Convert to DFA
4) Minimize DFA

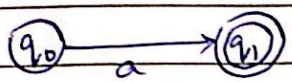
① Subset Construction Method
ϵ-NFA → DFA

② Thompson Construction
Reg Ex → ϵ-NFA

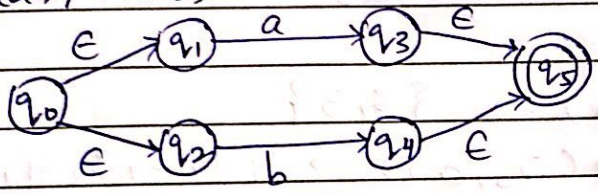
27/08/18

Thompson Construction

1. 

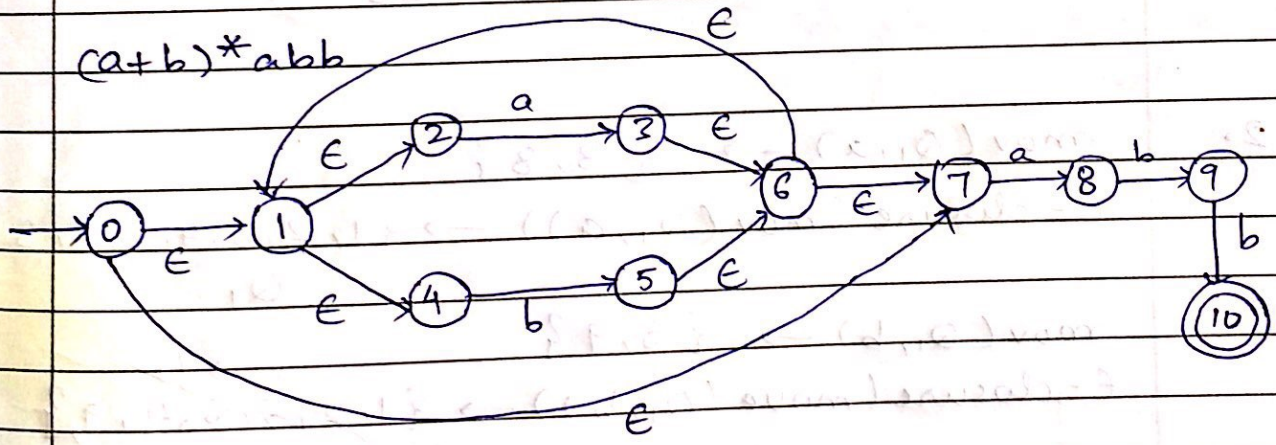
2. 

3. N(a) / N(b)



Construct ε - NFA for the following Reg Ex.

(a+b)*abb



- Use this ε-NFA to make a DFA (Subset construction method)

Regular Expression to DFA conversion
(Use the diagram on previous page)

Reg Ex^n: $(a+b)^*abb$

(A.) ϵ-closure $(q_0) \rightarrow \{0, 1, 2, 4, 7\}$
Q_0 (Initial state of DFA)

1. $mov(Q_0, a) \rightarrow \{3, 8\}$
ϵ-closure $(mov(Q_0, a)) \rightarrow \{1, 2, 3, 4, 6, 7, 8\}$
Q_1

$mov(Q_0, b) \rightarrow \{5\}$
ϵ-closure $(mov(Q_0, b)) \rightarrow \{1, 2, 5, 6, 7, 4\}$
Q_2

2. $mov(Q_1, a) \rightarrow \{3, 8\}$
ϵ-closure $(mov(Q_1, a)) \rightarrow \{1, 2, 3, 4, 6, 7, 8\}$
Q_1

$mov(Q_1, b) \rightarrow \{5, 9\}$
ϵ-closure $(move(Q_1, b)) \rightarrow \{1, 2, 5, 6, 7, 4, 9\}$
Q_3

3. $mov(Q_2, a) \rightarrow \{3, 8\}$
ϵ-closure $(mov(Q_2, a)) \rightarrow \{1, 2, 3, 4, 6, 7, 8\}$
Q_1

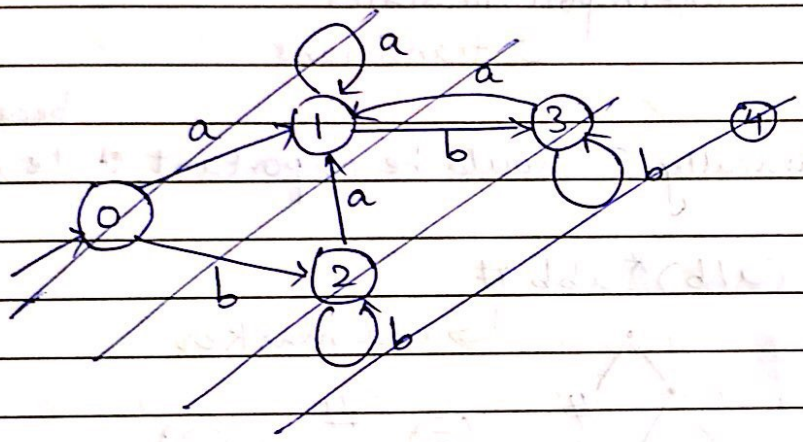
$mov(Q_2, b) \rightarrow \{5\}$
ϵ-closure $(mov(Q_2, b)) \rightarrow \{1, 2, 5, 6, 7, 4\}$
Q_2

4. mov(Q_3, a) → {3, 8}

ϵ-closure(mov(Q_3, a)) → {1, 2, 3, 4, 6, 7, 8} → Q_1

mov(Q_3, b) → {5, 10}

ϵ-closure(mov(Q_3, b)) → {1, 2, 5, 6, 7, 4, 10} → Q_4



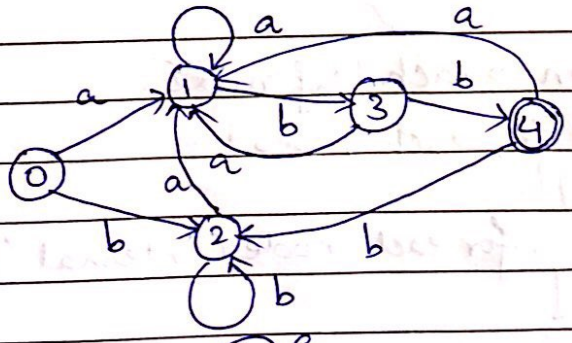
Final states: All those in which q_{10} occurs (or any other state through which we can reach q_{10} using ϵ - moves)

5. mov(Q_4, a) → {3, 8}

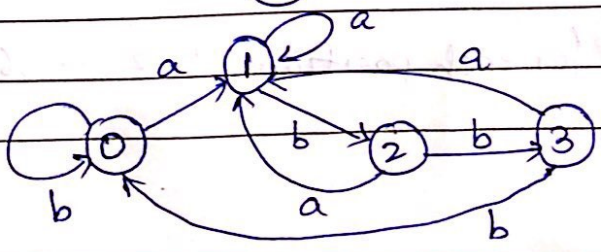
ϵ-closure(mov(Q_4, a)) → Q_1

mov(Q_4, b) → {5}

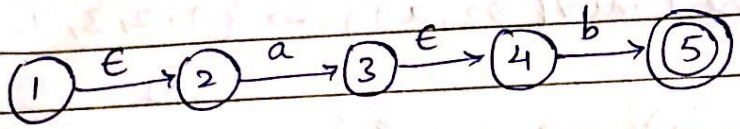
ϵ-closure(mov(Q_4, b)) → Q_2



Now minimize this DFA (0 & 2 combined)



28/08/18

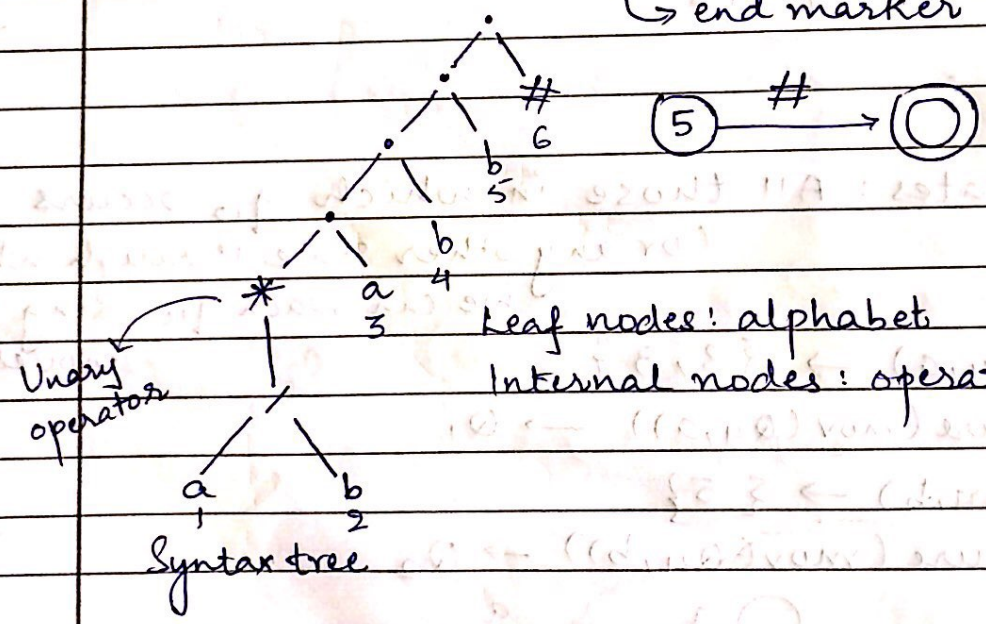


2, 4 → Important states
Non-ε transitions from these

1, 3, 5 → Un-important states
ε transitions

because it is
But technically 5 should be important state ∧ final state

(a|b)* abb #
↳ end marker



Leaf nodes: alphabet
Internal nodes: operators

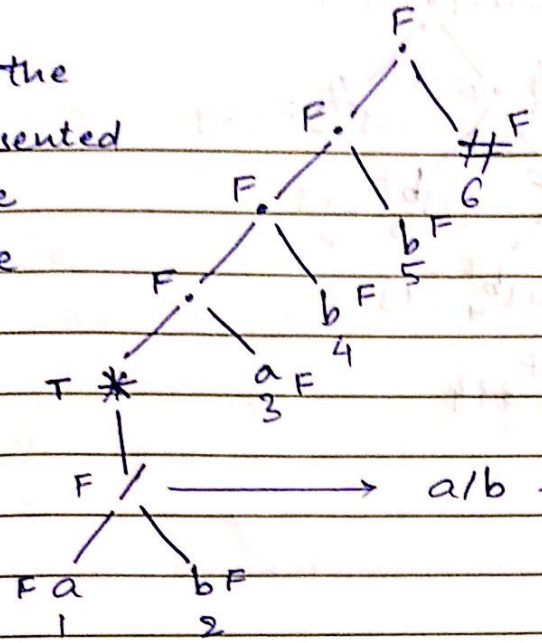
Syntax tree

1. Assign a position to each leaf node.
2. Calculate these for each node :-
   - nullable
   - firstpos
   - lastpos

   } for each node (internal / leaf)

   - followpos } for each position (1, 2 . . . , 6)

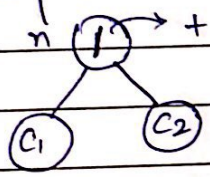
Nullable : Can the Reg Ex represented by that node generate a null (empty) string



a/b → Can it generate empty string? No

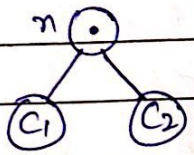
Reg Ex generated by a → a (No ϵ : hence nullable = F) (False)

	nullable(n)	~~first~~pos(n)
1. If n is a leaf labelled ϵ	T	ϕ
2. If n is a leaf labelled with position i (alphabet)	F	i

3. 

nullable(C_1) or nullable(C_2) →

if nullable(C_1)
then firstpos(C_1)
∨ firstpos(C_2)
else firstpos(C_1)

4. 

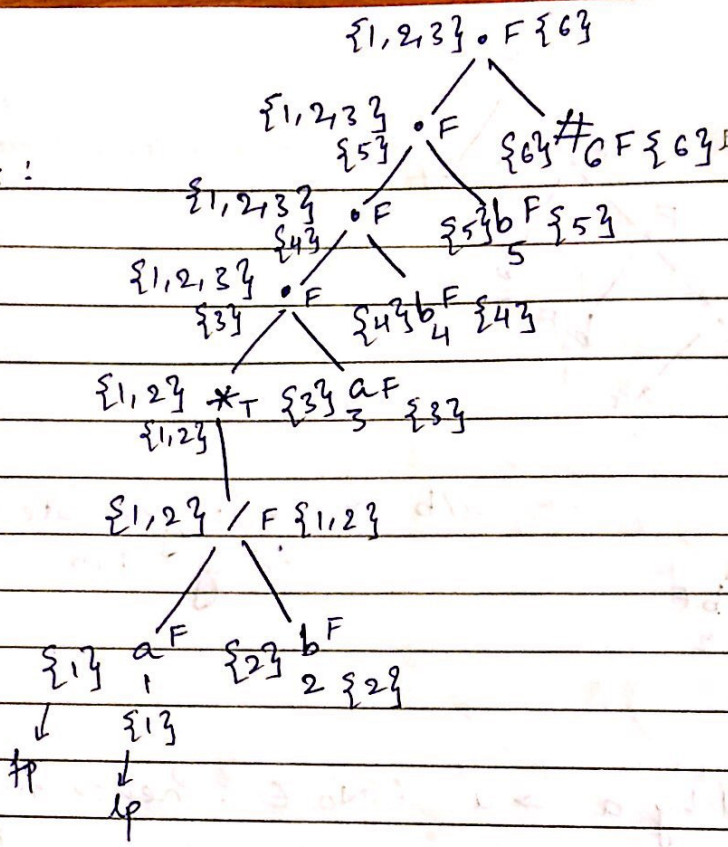
nullable(C_1) & nullable(C_2)

(poora box)

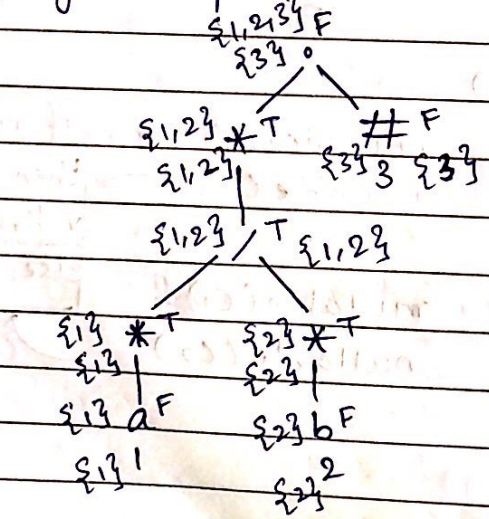
ab*c : firstpos → a (a hamesha aayega pehle)

a*b*c : firstpos →

firstpos :



Regular Expression : $(a^* / b^*)^*$



Follow → we find for each position

(a/b)*abb

position	follow (i)
{1}	{1, 2, 3}
{2}	{1, 2, 3}
{3}	{4}

29/08/18.

position	follow(i)
{4}	{5}
{5}	{6}
{6}	

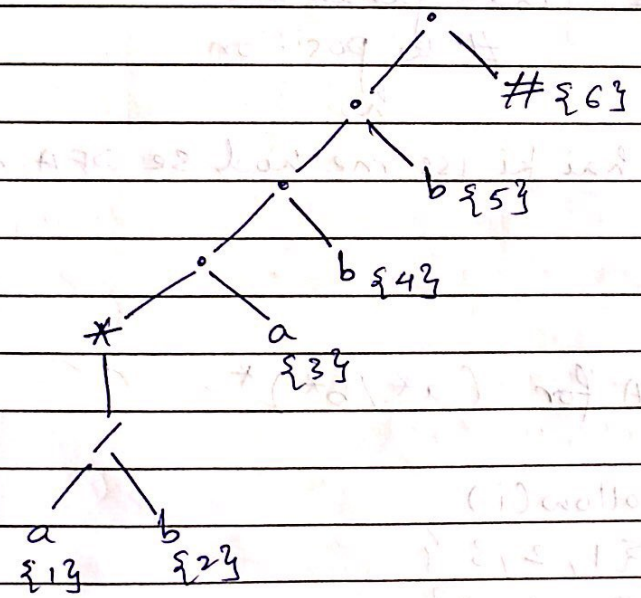
We find follow using · & * nodes

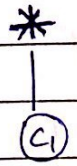
· abc(a|b)

↓

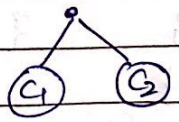
c follows b ⇒ b ka follow c

c ka follow ⇒ a|b





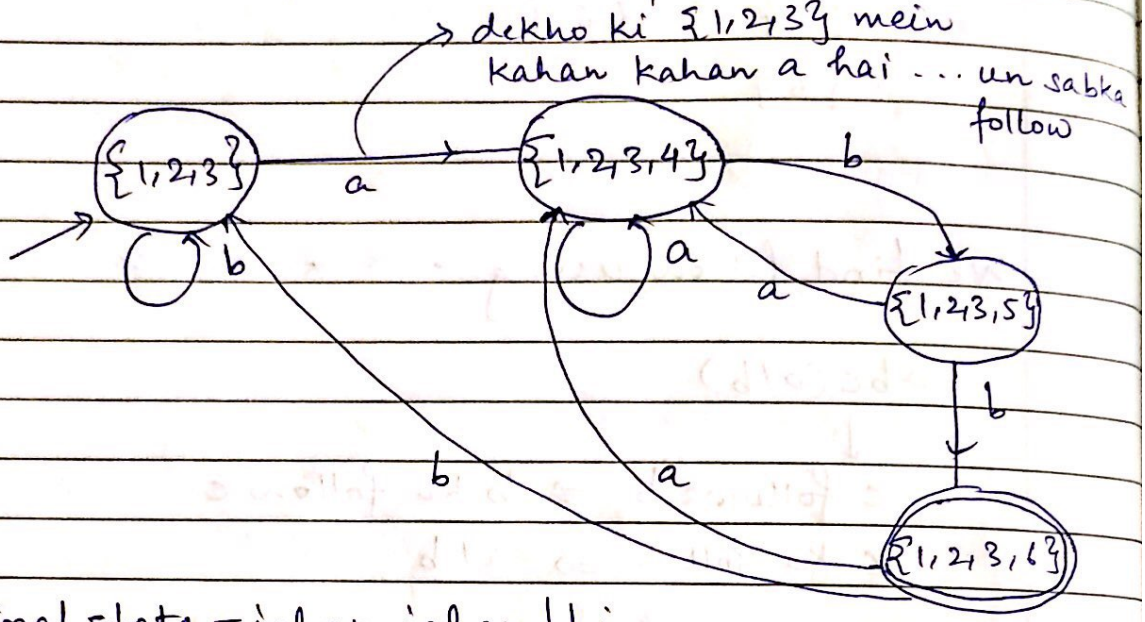
All the positions in first of c_1 are in follow of last positions of c_1



All the positions in first of c_2 are in follow of last positions of c_1

Making DFA using tree

Initial state of DFA → first position of root

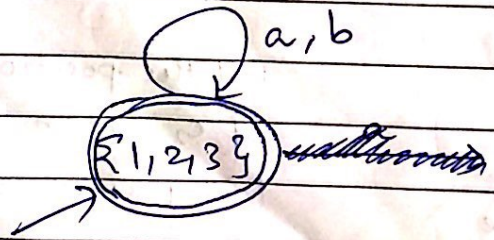


final state - jahan jahan bhi # ki position ho

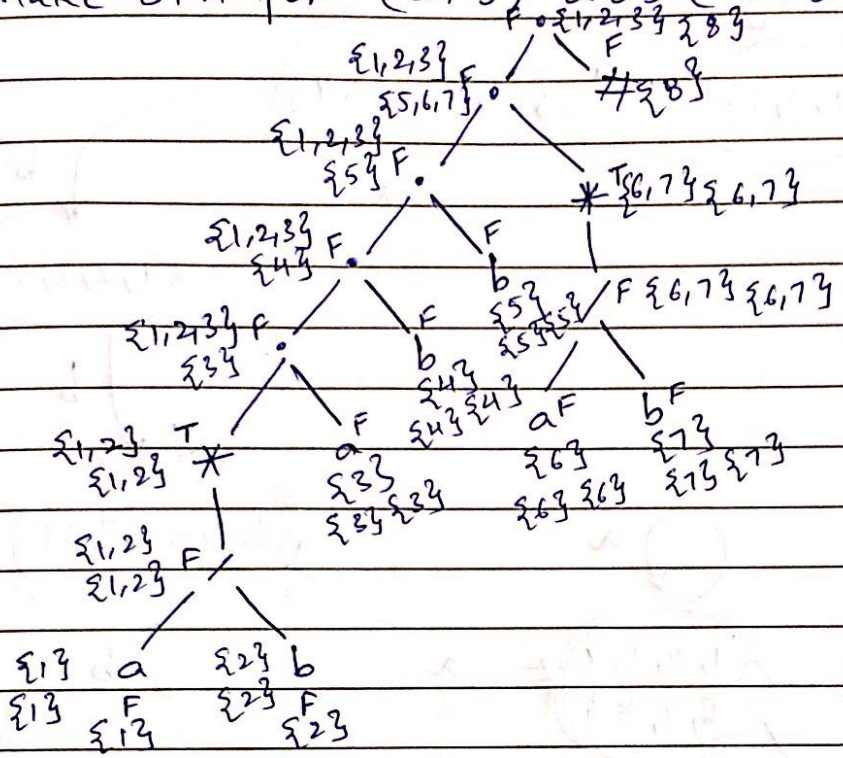
Zaruri ni hai ki iss method se DFA minimized bane

Q. Make DFA for $(a^*/b^*)^*$

position	Follow(i)
{1}	{1, 2, 3}
{2}	{2, 1, 3}
{3}	



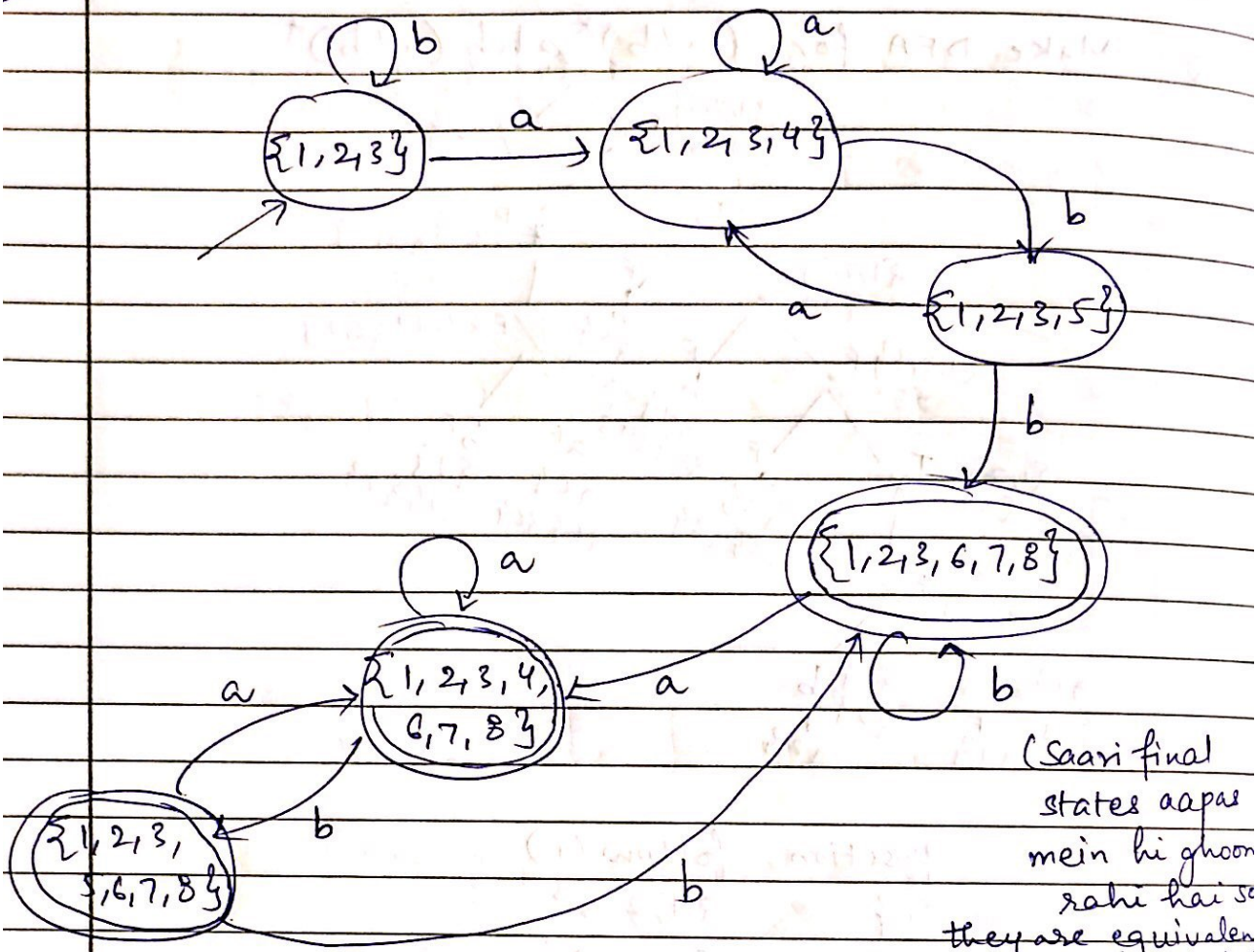
Q. Make DFA for $(a|b)^*abb(a|b)^*$



position	follow(i)
{1}	{1,2,3}
{2}	{1,2,3}
{3}	{4}
{4}	{5}
{5}	{6,7,8}
{6}	{8,6,7}
{7}	{8,6,7}
{8}	

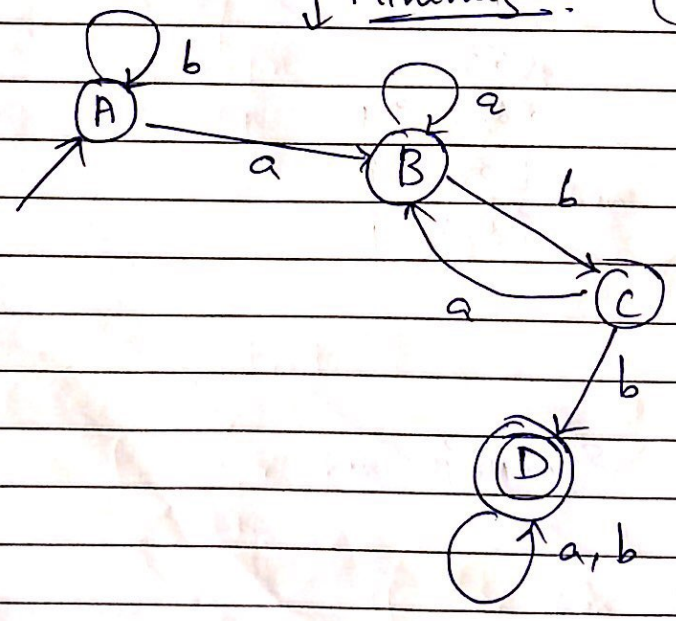
Initial state : {1,2,3}

08/18.

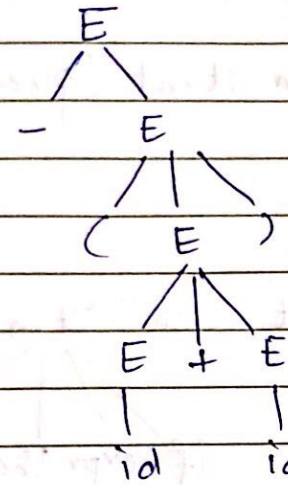


(Saari final states aapas mein hi ghoom rahi thi so they are equivalent to a single state)

↓ Minimize. (Exam mein do properly using metho



4/9/18 Leftmost, rightmost derivative



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

Grammar:
$E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$

Q. id + id * id
Make leftmost derivative

(1) $E \rightarrow E*E \rightarrow (E)*E$
$\hookrightarrow$ X (We can't use those symbols (bracket) which are not there in the initial expression)
$\rightarrow E+E*E$
$\rightarrow id+id*id$ } 3 steps

This is the correct one (acc. to precedence order)

(2) $E \rightarrow E+E$
$\rightarrow id+E$
$\rightarrow id+E*E$
$\rightarrow id+id*E$
$\rightarrow id+id*id$

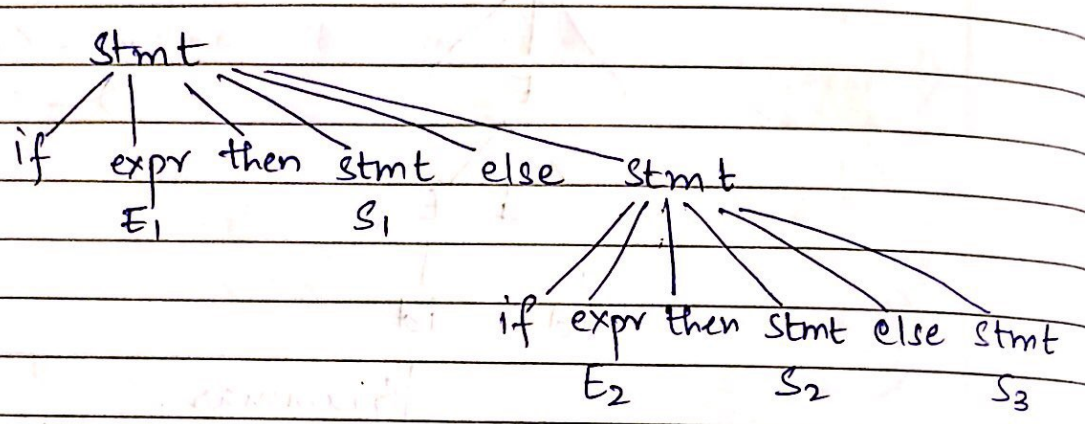
If we can make more than 1 leftmost derivative or more than 1 rightmost derivative for the given string $\Rightarrow$ grammar is ambiguous

stmt $\rightarrow$ if expr then stmt / if expr then stmt else stmt / others

4/9/18

Q. if E_1 then S_1, else if E_2 then S_2 else S_3

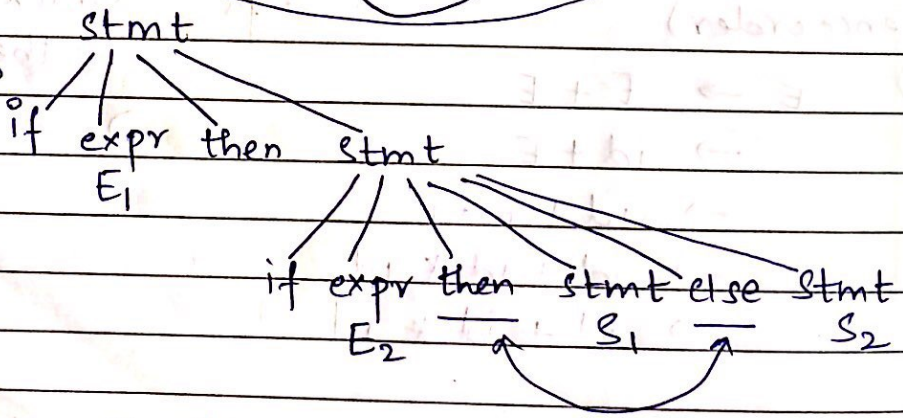
Derive this string from stmt (previous page)

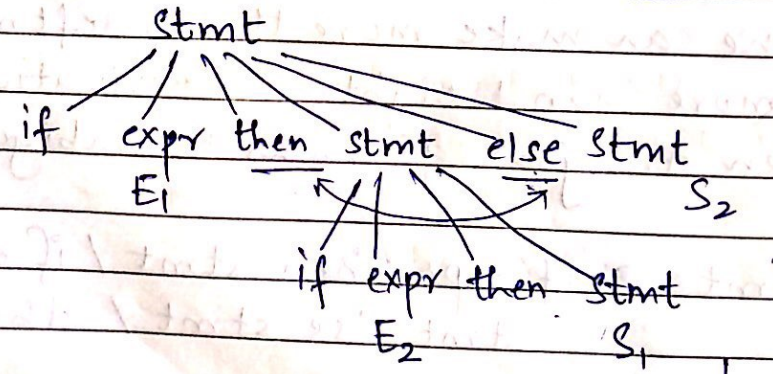


There is no ambiguity in this grammar. Ek hi leftmost derivative possible hai

Q. if E_1 then if E_2 then S_1 else S_2

This tree is correct →





In Pascal, else should be matched to the closest then

} Yeh part beech mein khaali reh sakta hai

aisa nahi hona chahiye. Agar is apna else hota toh theek th

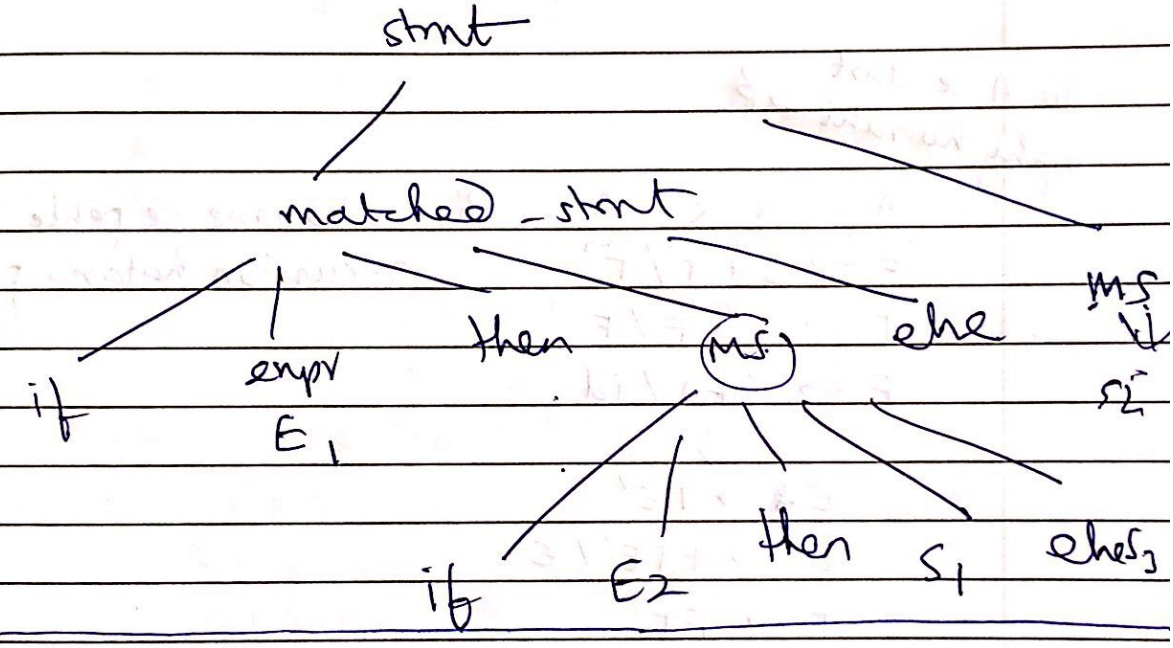
4/9/18

stmt → matched_stmt / unmatched_stmt

matched_stmt → if expr then matched_stmt
else matched_stmt / other

Example

if E_1 then if E_2 then S_1 else S_2 else S_3



unmatched_stmt → if expr then stmt / if expr then matched_stmt else unmatched_stmt

5/9/18

Left Recursion

$A \longrightarrow A\alpha / \beta$ → Left recursive grammar

How to remove left recursion

Eg (1) $A \longrightarrow A\alpha / \beta$

$\Downarrow$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \epsilon$

Eg (2) $A \rightarrow A\alpha_1 / A\alpha_2 / \beta_1 / \beta_2 / \beta_3$

$\Downarrow$

$A \rightarrow \beta_1 A' / \beta_2 A' / \beta_3 A'$

$A' \rightarrow \alpha_1 A' / \alpha_2 A' / \epsilon$

Jo A se start nahi ho rahi → β

$E \rightarrow E+T / T$ (A: E; α: +T; β: T)

Parser banane se pehle left recursion hatana padega

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

$\Downarrow$

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

$F \rightarrow (E) / id$

(1) $A \rightarrow A\alpha / \beta$ (Immediate left recursion)

↳ A directly derives A

(2) $S \rightarrow AB / c$

$A \rightarrow Ad / SAc / e$ → Immediate

$S \rightarrow AB \rightarrow SAcB$ (Indirect left recursion)

5/9/18

How to remove indirect left recursion

$S \to A_1 a / b$ ($A_1 \to A$)

$A \to A_2 c / S d / \epsilon$ ($A_2 \to A$)

$A_1 \to A_2 a / b$

$A_2 \to A_2 c / A_1 d / \epsilon$

~~$A_1 \to A_2 ca / A_1 da / \epsilon a / b$~~

~~$A_2 \to A_2 c$~~

$A_2 \to A_2 c / A_2 ad / bd / \epsilon$

(α_1 α_2 β_1 β_2)

$A_2 \to bd A' / A'$

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

$A_1 \to bdA'a / A'a / b$

Left factoring → if expr then stmt else stmt

→ if expr then stmt

$S \to ietses / iets$

Agar i mila toh kaise choose karenge ki ietses lena hai ya iets ... both start with i

One by one karenge toh backtrack karna padega

ietses iets

$S \to \alpha A'$ common part → α

$A' \to \beta_1 / \beta_2$ A' → remaining

In this case

$S \to iets A'$ Ab backtrack ni karna

$A' \to es / \epsilon$ padega

Imp° Parser ke liye remove left recursion & do left factoring