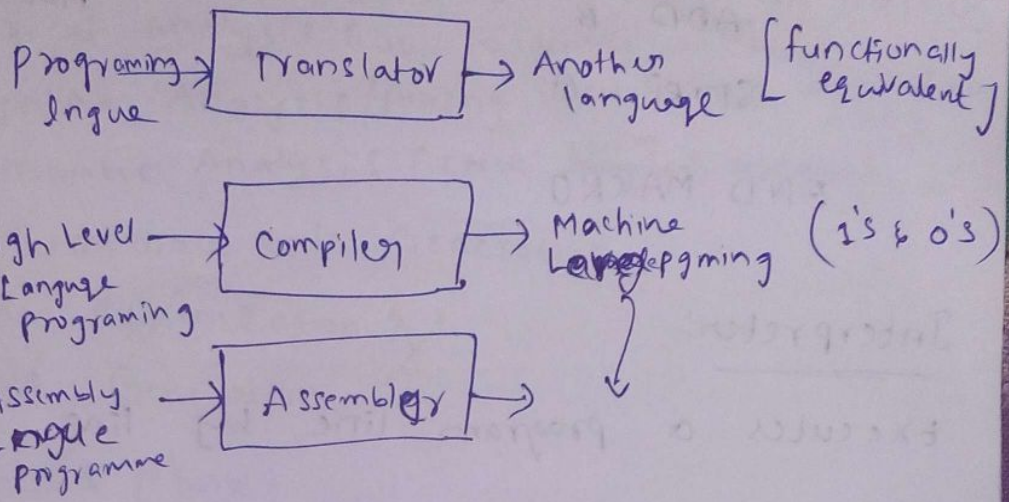
20/11/17

CD

Translator :-



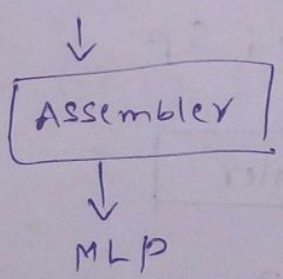
Assembly Language :

In this language we represent,

→ operations with Mnemonics

→ addresses with symbolic names

Eg:- Add A,B



→ It has MACRO Facility (predefined functions which supplies group of statements).

Eg: ADD 2, A, B

LOAD A

ADD B

STORE A

END MACRO

ADD 2 C, D

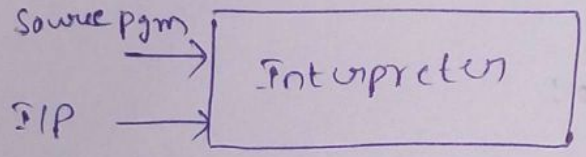
LOAD C

ADD D

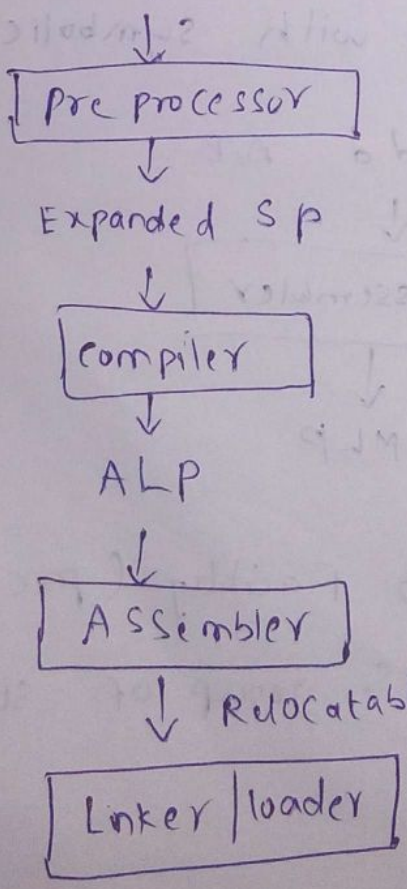
STORE C

Interpreter:

Executes a program line by line.



⇒



Analysis Phase:-

This Analysis the code and produces this code into intermediate code. and This phase depends on the language which we write on the compiler.

→ Lexical Analysis (used to identify words)
→ Syntax Analysis/Parsing (used to check valid sentences or not)
→ Semantic Analysis (check meaning of sentences)
→ Intermediate code Generation
→ code optimization
→ code Generation

Synthesis phase:-

This takes the intermediate code and produces it in a normal code (or) (single code)

→ code optimization
→ code Generation

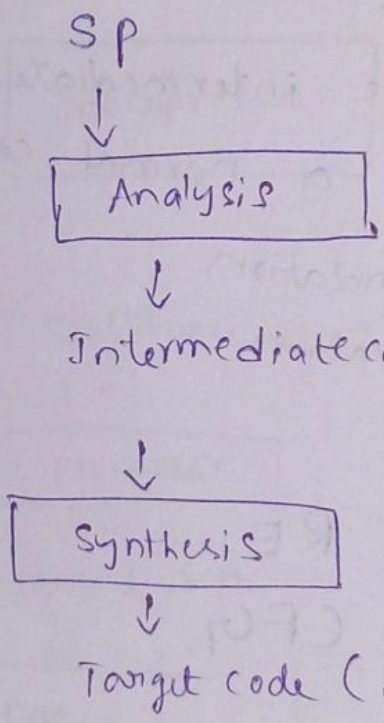
LA — RE
SA — CFG

21/11/17

Analysis and Synthesis model:-

→ Analysis model analyses the source program and generates the intermediate code.

→ Synthesis model take the intermediate code and generates a target code.



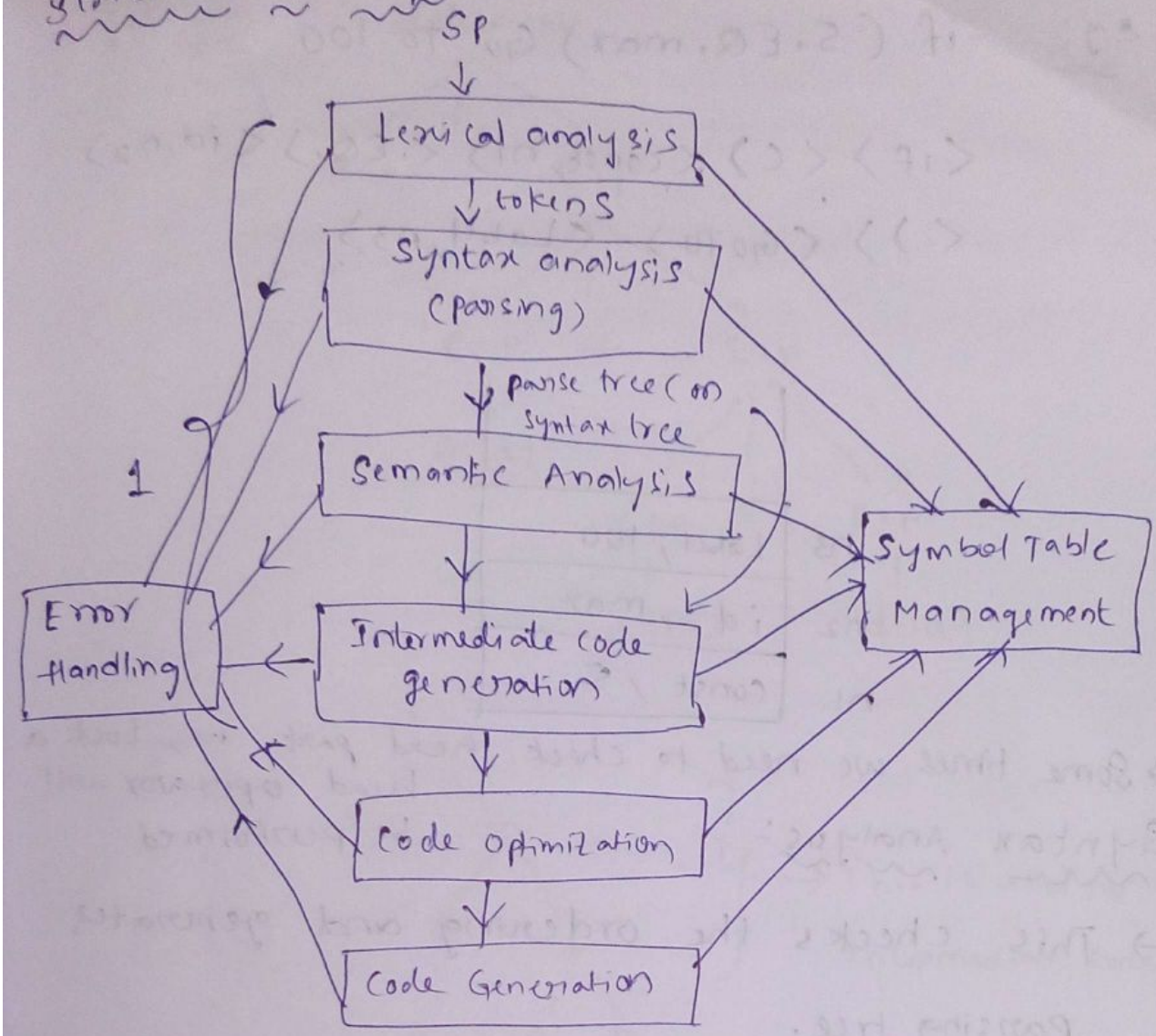
Analysis model:-

→ Lexical analysis
→ Syntax / Parsing analysis
→ semantic analysis
→ Intermediate code generation

Synthesis model:-

→ code optimization
→ Code Generation

Structure of Compiler:-



→ data structure store and retrieve information

Lexical Analysis :- It returns tokens and components

Token :- →It is a pair of components.

It may be user defined variable / operation / keyword / punctuation.

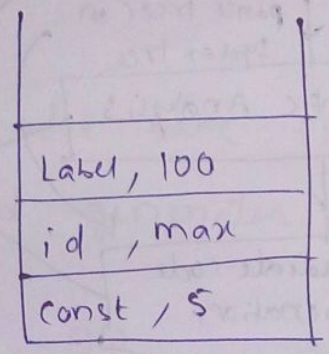
→ Lexical analysis defines the

* (type, value) pointer to symbol table

* Regular expression

Eg:- if (5.EQ.max) Go TO 100

<if> <(> <const, n1> <.EQ.> <id, n2>
<)> <GoTO> <Lab4, n3>



→ Some times we need to check head part i.e., Look a head operator will be performed

Syntax Analysis:-

→ This checks the ordering and generates parsing tree.

→ CFG is used in this syntax analysis

CFG

id is an exp

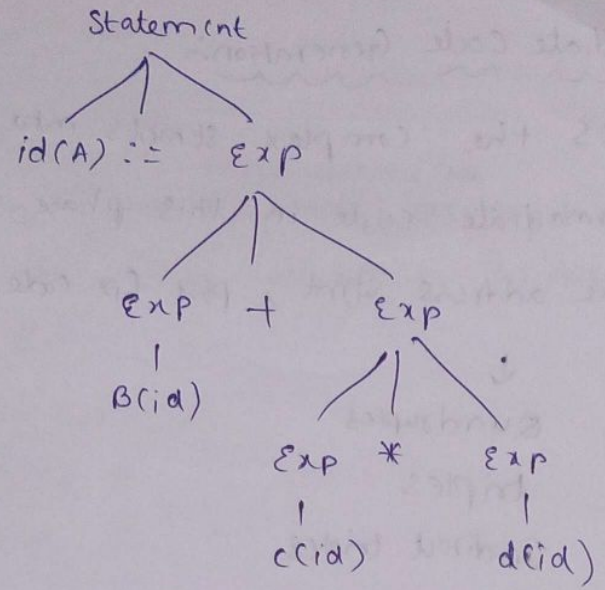
const is an exp

exp combined with an exp

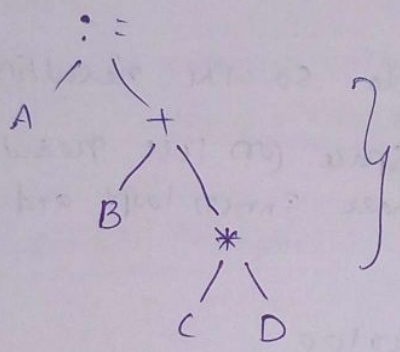
exp assigned to a id

Eg:- A = B + C * D

parse tree



Syntax tree



→ This is used for Intermediate codes.

→ syntax tree eliminates the redundant information.

Symantic Analysis:-

→ checks the meanings of the sentences.

→ Type checking is performed in this phase.

→ Mixed mode is used in this analysis.

int - Real

index - int

Real - int

Intermediate Code Generation:-

→ Breaks the complex stmts into simple stmts.

→ Intermediate code in this phase is

Three address stmt / post fix code / syntax tree

↓

Quadruples

triples

Indirect triples

Code optimization:-

→ optimizes the code. so the resulting program optimizes less space (or) less resulting time.

we use in this phase Inner loops and common sub exp elimination.

→ Inner Loops

```
For i:= 1 to 100
For j:= 1 to 100
    k:= 1 to 100
    - - -
    - - - -

    p = 1
    - - - - -
    - - - - -
```

→ common sub exp elimination

Eg: A := B + C + D

X = B + C + E

assign T = B + C

$\therefore$ A = T + ID

X = T + E

code Generation:

→ Takes the (optimized code) as input and (Three addressed code) generates the final code which is Assembly Language code.

A = B + C

Load B

Add C

Store A

22/11/17

structure of compiler with example:

(Q) position := initial + rate * 60

↓

RE → [Lexical Analysis]

↓

Symbol Table

4	Const	
3	rate	
2	initial	
1	position	

<id, 1> <:=> <id, 2> <+> <id, 3> <*>
<const, 4>

$id_1 := id_2 + id_3 * 60$

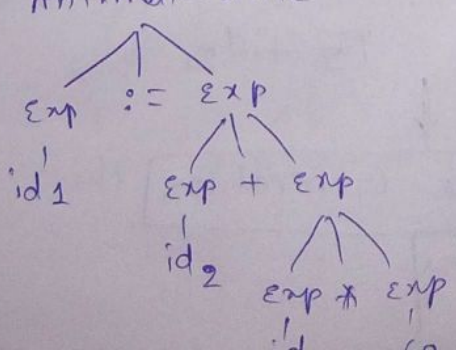
↓

CFG → [Syntax Analysis]

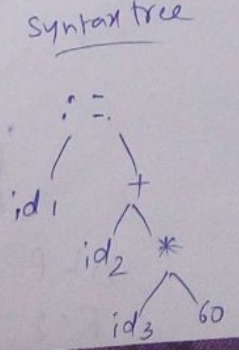
↓

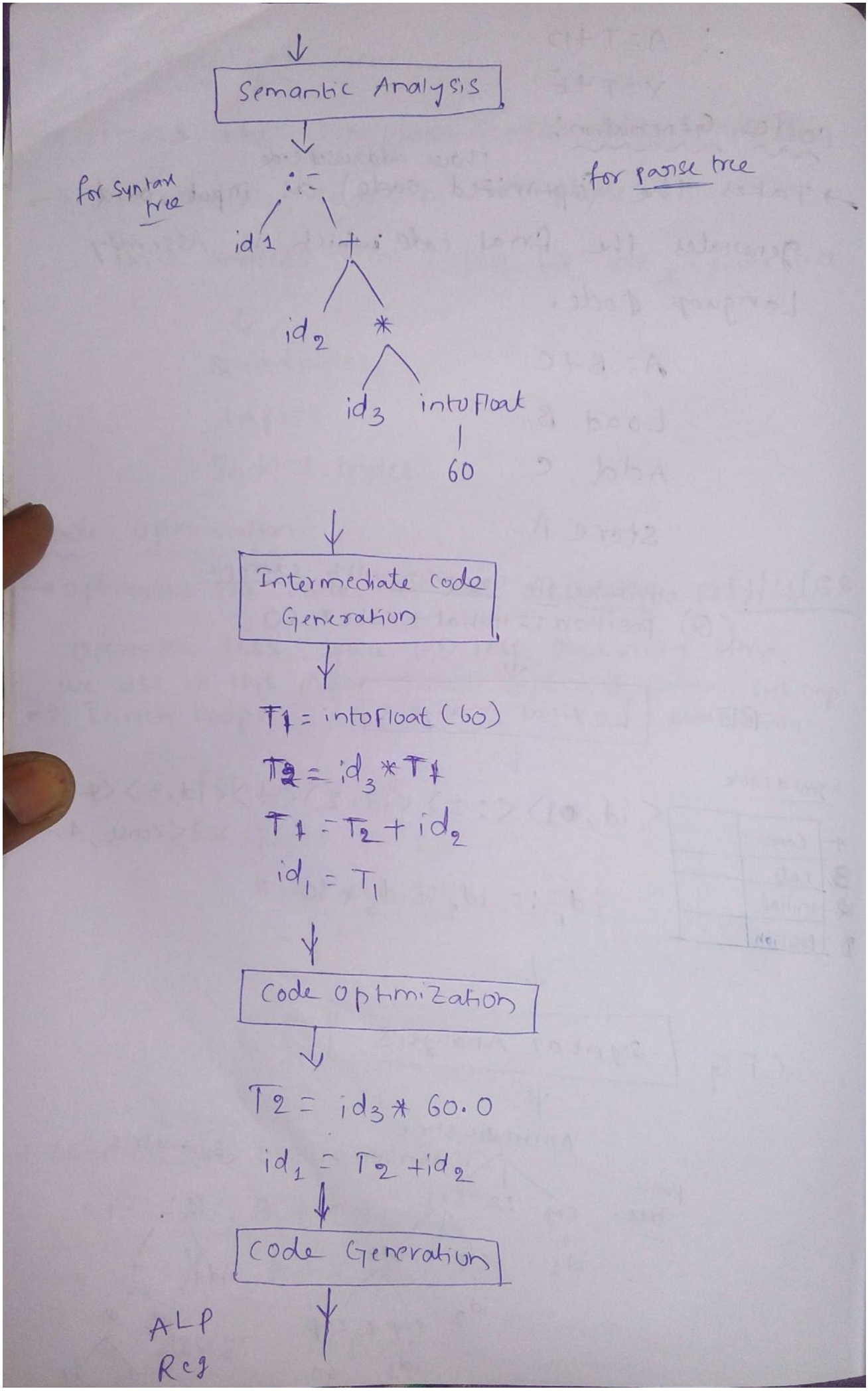
Arithmetic stmt

parse tree



Syntax tree

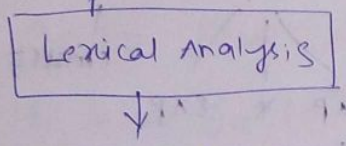




```
MOV  R1, id3   / write comment
MUL  R1, #60.0 (or) MUL R1, R1, #60.0 / comment
MOV  R2, id2   / comment
ADD  R1, R2    / comment
MOV  id1, R1   / comment
```

Generate Intermediate code for :

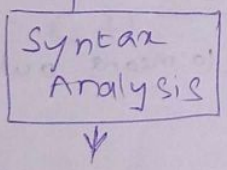
(Q) while (A > B & A <= 2 * B - 5) do A := A + B



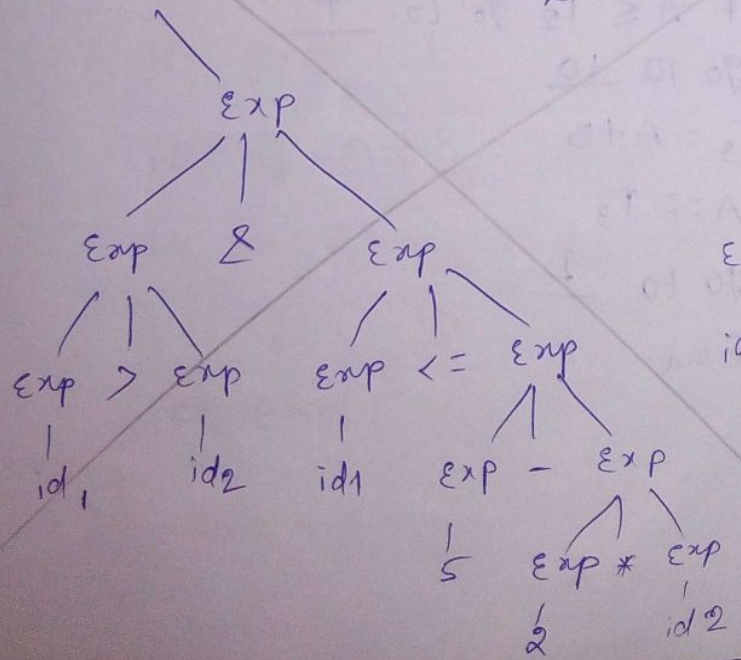
<while> <(> <id, 1> <>> <id, 2> <&> <id, 1>

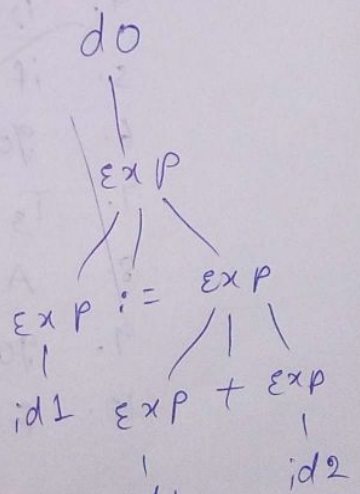
<<=> <const, 3> <*> <id, 2> <-> <const, 4>

<)> <do> <id, 1> <:=> <id, 1> <+> <id, 2>

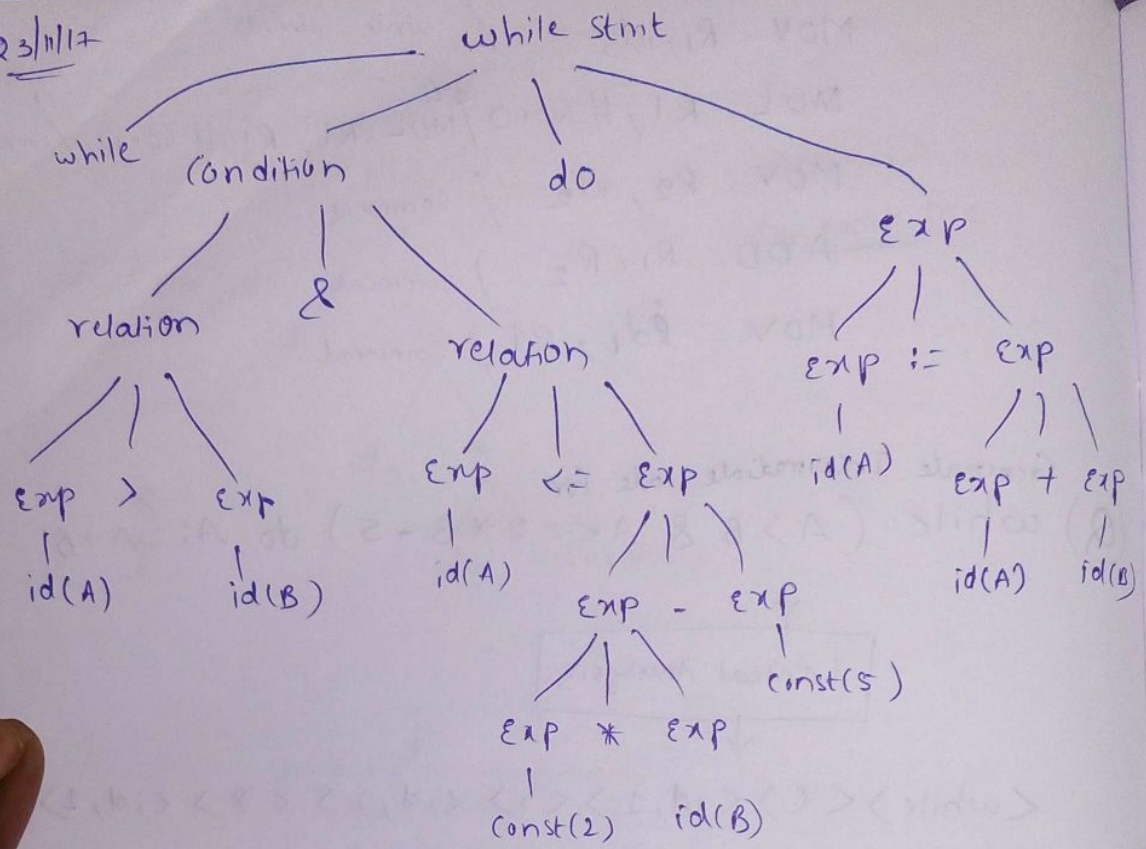


while → correct in next page





23/11/17

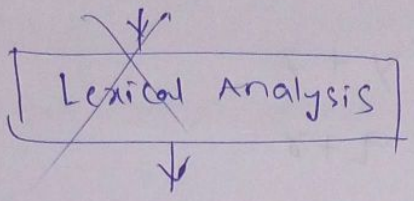


↓

Intermediate code Generation

1. IF A > B go to 3
2. go to 10 [10 means outside loop] fail condition
3. $T_1 = 2 * B$
4. $T_2 = T_1 - 5$
5. if $A \leq T_2$ go to 7
6. go TO 10
7. $T_3 = A + B$
8. $A := T_3$
9. go to 1
10. End.

(Q) write three address code for
while (a<c and b>d) do if a=1 then
c=c+1 else
while a<=d do a=a+3



<while>

Sol.

1: if a<c go to 3
2: go to End
3: if b>d go to 5
4: go to End
5: if a=1 then go to 7
6: go to 10
7: $T_1 = C+1$
8: $C = T_1$
9: Go To 1
10: if a<=d go to 12
11: go to 1
12: $T_1 = a+3$
13: $a = T_1$
14: go to 10
15: End

(Q) write the three address code for

```
switch(i+j)
{
  case 1: x = y + z
  case 2: p = q + r
  default: v = v + w
}
```

Sol:

```
1  T = i+j
2  if (T=1) go to 6
3  go to 4
4  if (T=2) go to 7
5  go to 8
6  x = y+z
7  p = q+r
8  v = v+w
```

Sol:

```
1. T1 = i+j
2. IF T1 == 1 go to 4
3. go to 6
4. x = y+z
5. go to 11
6. IF T1 == 2 go to 8
7. go to 10
```

8. p = q + r

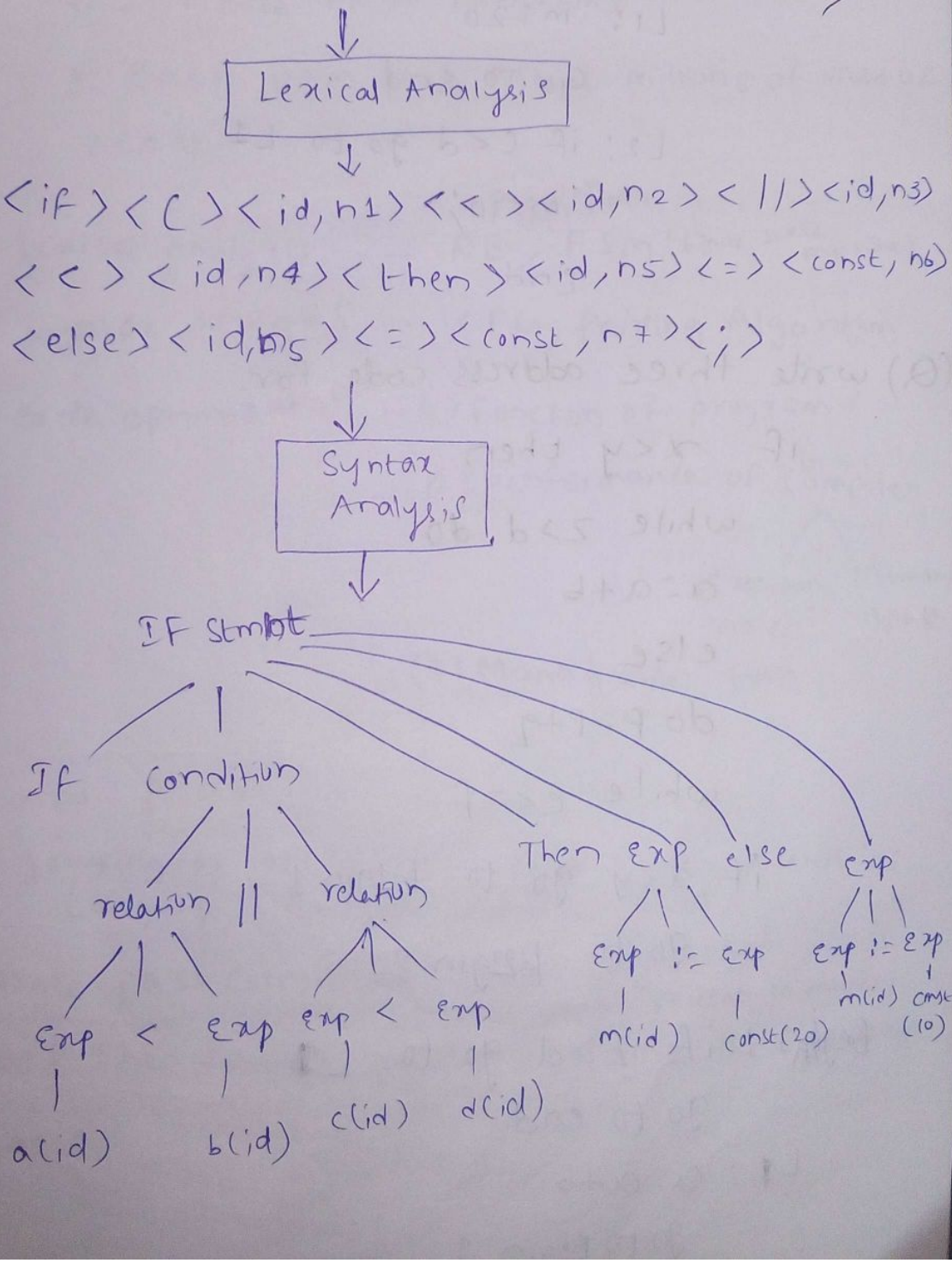
9. go to 11

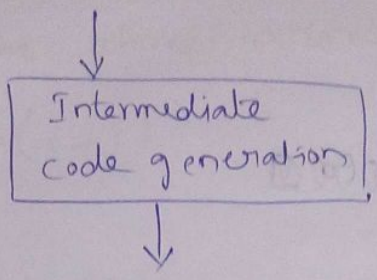
10. u = v + w

11. End

27/11/17

(Q) if (a<b || c<d) then m=20 else m=10;






```
if a<b go to L1
go to L2
L1: m=20
    Go TO end
L2: if c<d go to L1
    m=10;
    end
```

(Q) write three address code for

```
if x<y then
while z>d do
a=a+b
else
do p=p+q
while e<=f
```

```
if x<y go to Begin 1
    go to Begin 2
Begin 1 : if z>d go to L1
    go to end
L1 : a=a+b
    go to Begin 1
```

Begin 2: p = p+q

if e<=f go to begin 2

End;

30/11/17

Science of designing a compiler:-

1. Accept any program
2. Infinite no. of programs
3. Each pgm may accept millions of lines of code.

for Lexical Analysis — RE (Regular expression), FSM (Finite state machine)

Syntax Analysis — CFG, Parsing Algorithm

code optimization — (1) Function of program

(2) performance of compiler.

(execution time, Memory space)

(3) Manageable time.

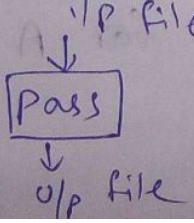
Pass and phase:-

→ grouping of one (or) more phases into a unit is Language, Machine called as "pass".

Single pass compiler:-

→ Total compiler should be present in memory

→ Reads the source pgm, performs the required phases.



Forward & Backward Jump:-

L1:

} backward bcoz L1 already known

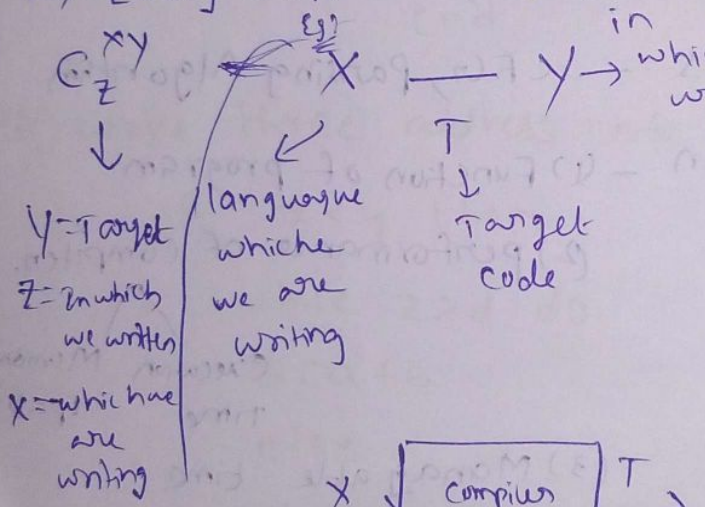
GO TO L1:

} forward bcoz L1 not known.

L1:

Boot strapping a compiler:-

→ writing the compiler in the same language
→ Every compiler contains three langues.



Eg:- (i) C^{LA}_{S} → C^{SA}_{A} → C^{LA}_{A}

In this, compiled on machine A and compiled for A itself.

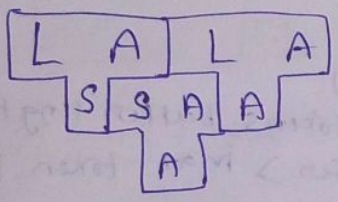
Eg:-(ii) C (L, LB) → [C (A, LA)] → C (A, LB)

The compiled for one machine and compiled on another machine is called "cross compiler"

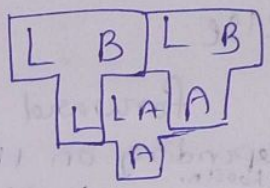
Eg:- (iii) C (L, LB) → [C (A, LB)] → C (B, LB) (as eg (i))

Simple diagram is based on above three eg's is

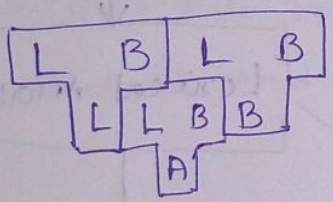
(i)



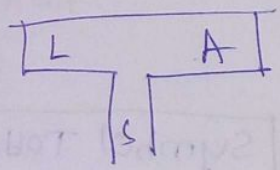
(ii)



(iii)



LA variable

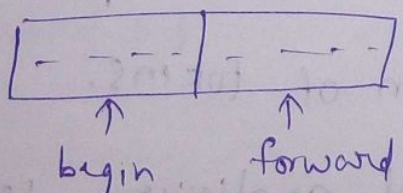


Lexical Analysis:-

(Q) IF A = 5.0

IF (A == B). X = X + 5

Look ahead operator



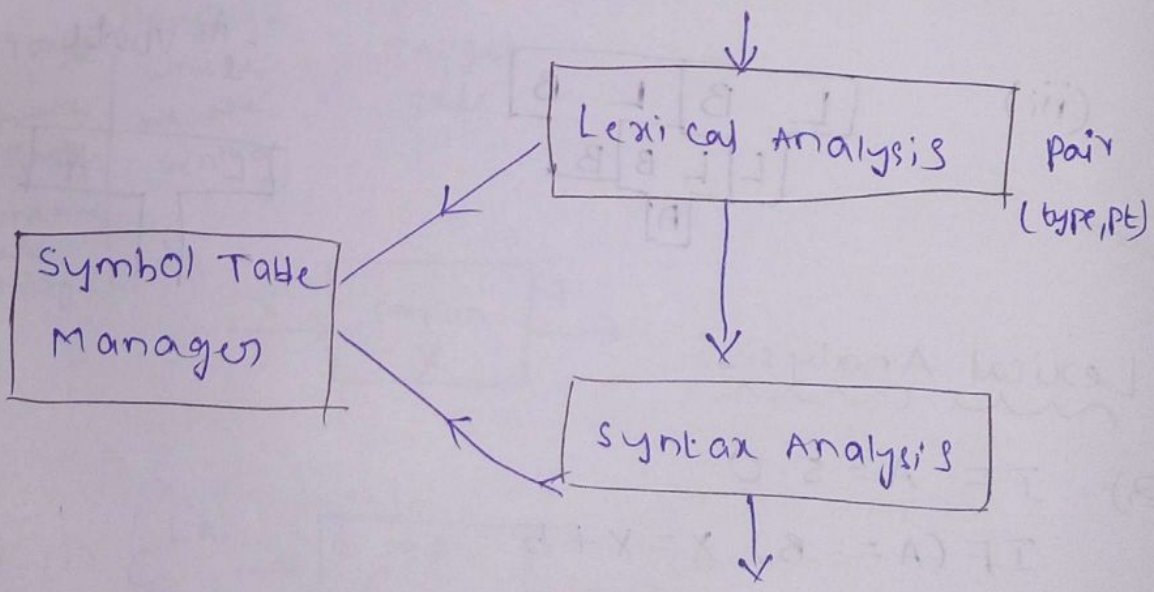
if forward at the end of first half begin

Next page continuation

Reload the second half
forward = forward+1
end
else
if forward at the end of the second half
begin
Reload the first half
move the forward to begining of first half
end
else
forward = forward+1

→ depending on the length of tokens, buffer length is to be choosen.
half length of buffer > max token length

1/12/17



If we use separate phases, it is easy for Specification of tokens.

The lexical analyser also does:-

→ Removing comment lines

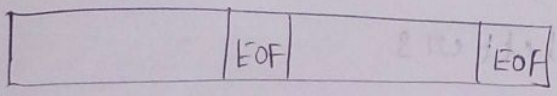
→ Blank spaces
→ values into symbols

TO design a Lexical analyser:-

step - 1:- define possible TOkens (R E)

step - 2:- Mechanism to identify them (F SM)

EOF → End of File (marker) used at the end of file.

*2m 

using EOF:- (Input Buffering)

Forward = Forward + 1;
if forward == EOF then
iff forward at end of 2nd half
begin
Reload the 1st half
move the forward to begining of 1st half
end
else if forward at the end of 1st half
Reload the 2nd half
forward = forward + 1
end
else
exit

Design of Lexical Analyser:

(1) RE

(2) FSM, Automata

identifier

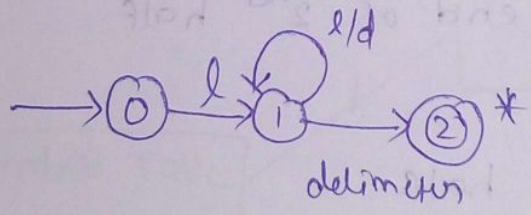
eg: $l(l/d)^*$

$*$ = 0 (or) more

l = letters

d = digits

delimeter → Appears after the identifier but not a part of identifier.



Pseudo code for above diagram:

State 0 : c := getchar();

if letter (c) go to state 1

else

fail();

State 1:-

```
c := getchar();
if letter(c) or digit(c) then goto
state 1
else
  if delimeter(c)
go to state 2
else
  fail.
```

State 2:-

Retract()
↳ moves the pointer in backward by 1 position.

return(id, install())
↳ returns pointer to the symbol table.

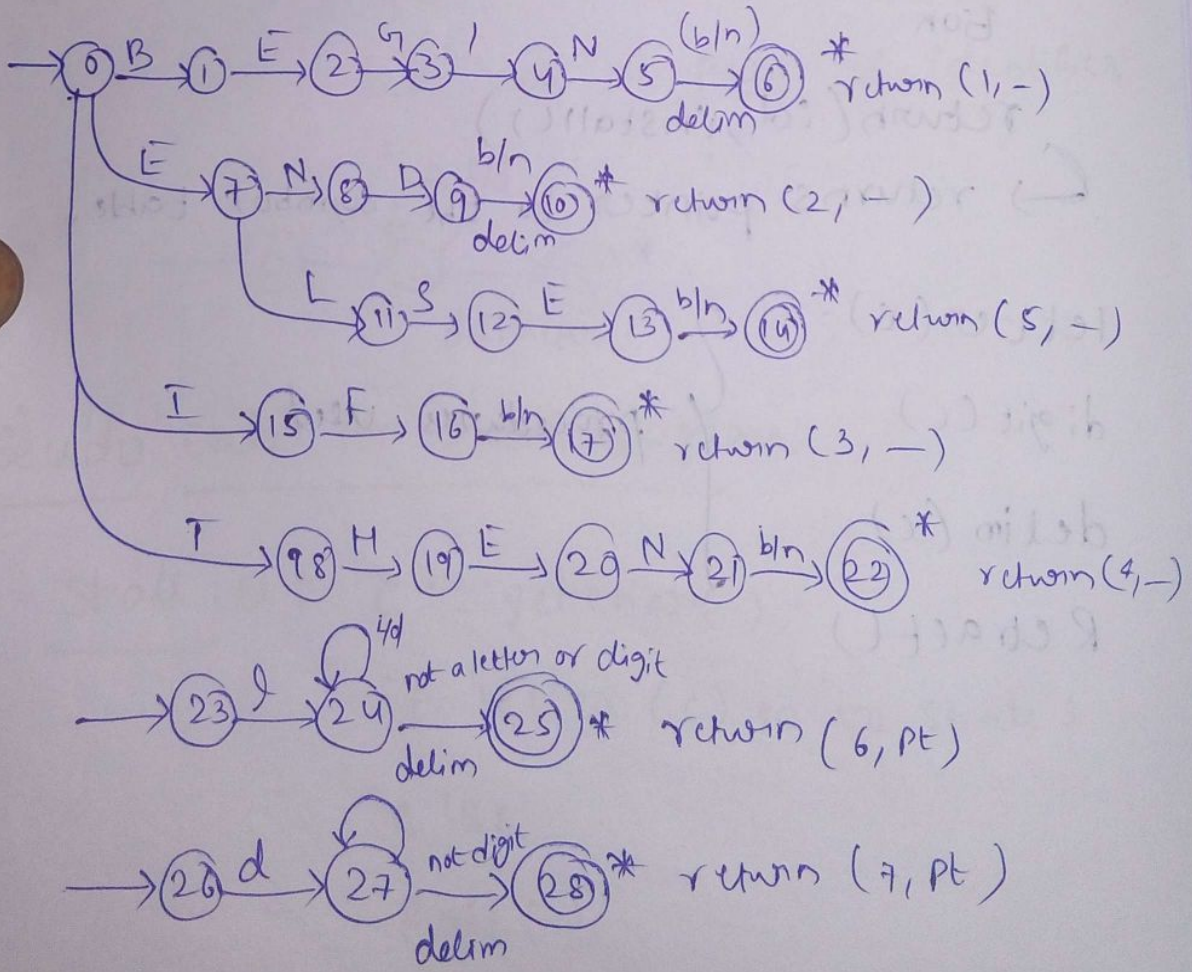
letter(c)
digit(c)
delim(c)
Retract()
} procedures used.

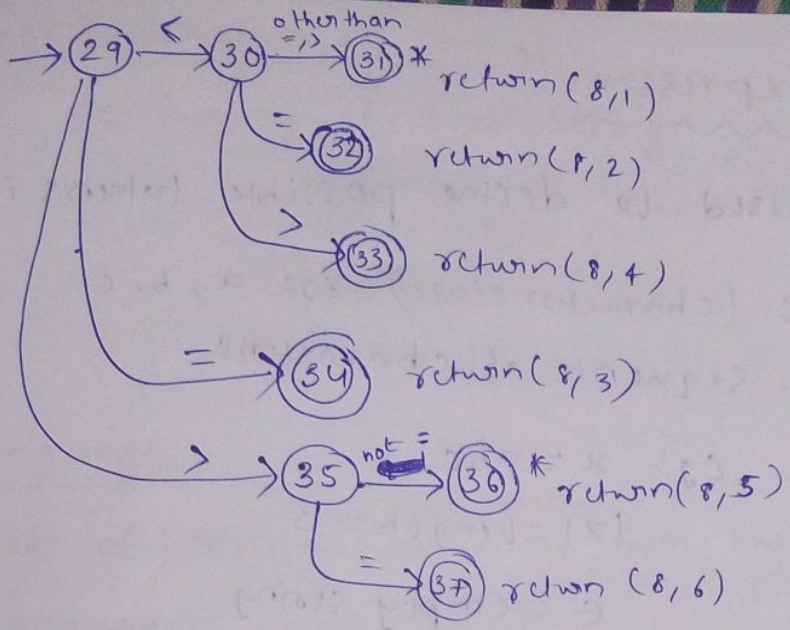
2/12/17

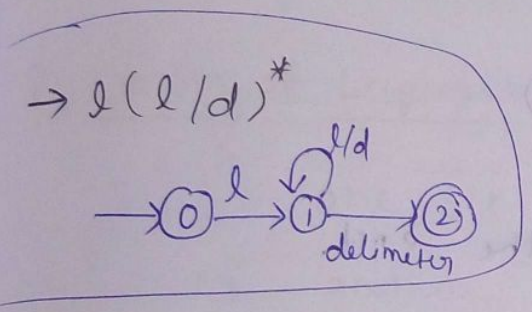
Design of LA:-

	code
Begin	1, —
end	2, —
if	3, —
then	4, —
else	5, —
identifier	6, pt
constant	7, pt

<	8, 1
<=	8, 2
=	8, 3
<>	8, 4
>	8, 5
>=	8, 6







Code for above states will be written as

(1) CASE) switch
Case 'B'
Case 'E'
- - - like this (OR)

(2) character to int
index into array of labels

(3) 2 (OR) 3 if not Req. character
Restart to begin id

Regular Expressions:

Notation used to define possible tokens is R.E.

1. Alphabet :- (character class) eg:- a, b, c
2. String :- sequence of characters

eg:- x = abc

|x| = Length = 3

ϵ = empty string

3. prefix : (trailing symbols)

eg:- x = abcde

string that obtained by deleting symbols from the end.

x = abcd

x = abc

x = ab

x = a

4. Suffix:- (leading symbols)

deleting symbols from the starting

x = abc

x = bc

x = c

5. Substring:-

Leading, trailing (or) both

proper prefix, suffix, substring

(one (or) more)

S is R.E

6: concadination of strings:

combining of two strings

$x = ab$

$y = cde$

$xy = abcde$

7. Language:

set of strings obtained from the specific language.

8. Union of Lnguge:

L - one set of strings from one language

M - another " " " another language

Union: $L \cup M = \{ x / x \text{ is in } L \text{ (or) } M \}$

Concadenation: $LM = \{ xy / x \text{ is in } L \text{ and } y \text{ is in } M \}$

→ $\phi = \{ \}$ $(\{\epsilon\})$

→ $L \cup \phi = L$

→ $L\phi = \phi L = \phi$

→ $L \cdot \{\epsilon\} = L$

Symbols used in RE:

| , • , * closure

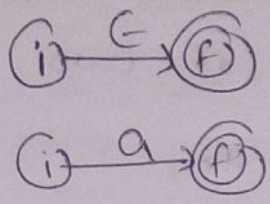
or ← • → concadination

priority

$* > \cdot > |$

(1) ϵ is a RE $\{\epsilon\}$

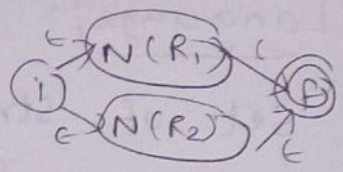
(2) a is in the Σ is RE $\{a\}$



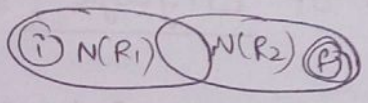
(3) R_1, R_2 are RE

$L(R_1)$, $L(R_2)$ are languages of R_1 & R_2

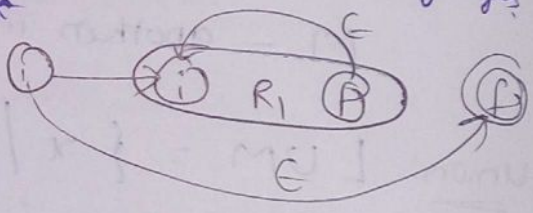
→ $R_1 | R_2$ is RE then language is $L(R_1) \cup L(R_2)$



→ $R_1 R_2$ denotes the concadination of RE then language is $L(R_1)\,L(R_2)$



→ R_1^* also denotes the concadination of languages then $(L(R_1))^*$



4/12/17

Laws

(1) $R_1 | R_2 = R_2 | R_1$ → Commutative

(2) $R_1 | (R_2 | R_3) = (R_1 | R_2) | R_3$

$(R_1 R_2) R_3 = R_1 (R_2 R_3)$ } Associative

(3) $R_1 (R_2 | R_3) = R_1 R_2 | R_1 R_3$

$(R_1 | R_2) R_3 = R_1 R_3 | R_2 R_3$ } Distributive

(4) $R_1 \epsilon = \epsilon R_1 = R_1$

(5) $(R_1^*)^* = R_1^*$

eg: (1) $a\,(a/b)\,(a/b)^*$

min length = 2

(2) $\epsilon/a/b\,(a/b)\,(a/b)\,(a/b)\,(a/b)^*$

strings accepted by this are

→ ϵ, a, b

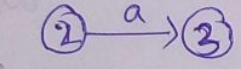
→ all strings a's and b's whose length is (not equal to 2) (or) length ≥ 3.

(3) RE → NFA → DFA → Minimization

Minimization → two process. (1) subset construction (2) partition (or) Minimization of DFA

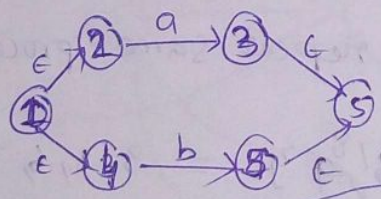
(3) $(a/b)^*abb$

a →

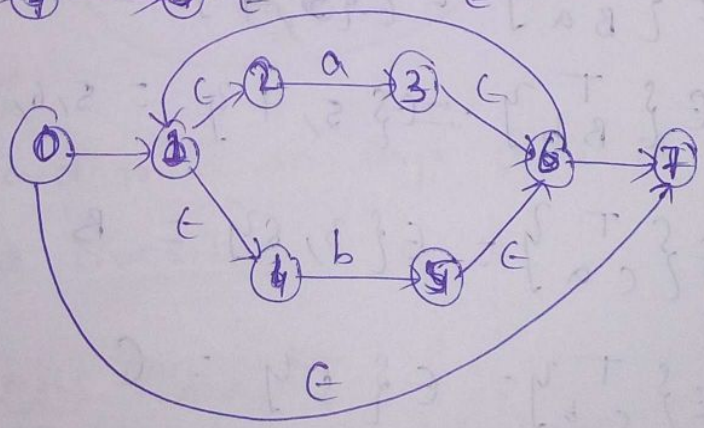


b →

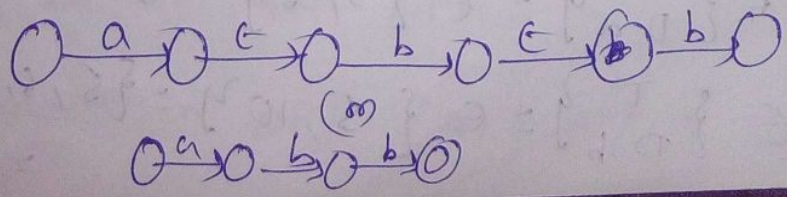
a/b →



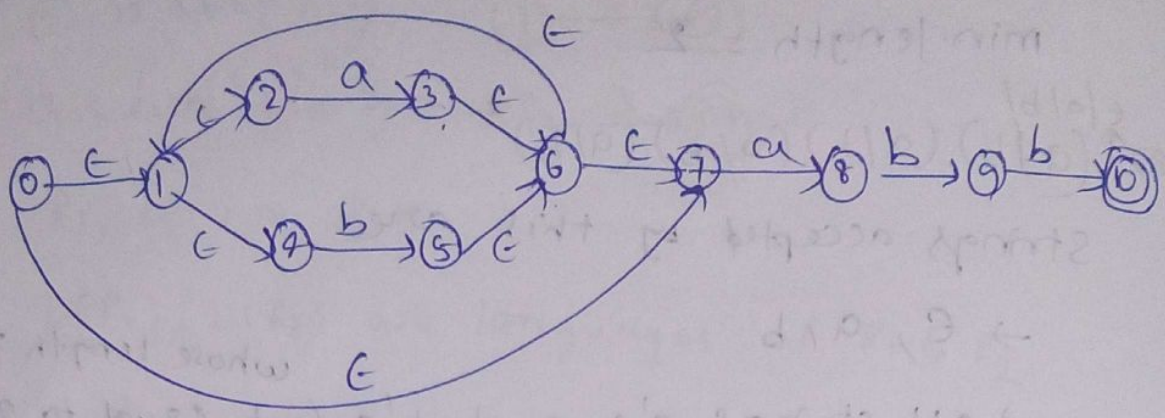
$(a/b)^*$ →



$(a/b)^*abb$ →



$(a|b)^* abb \longrightarrow$



ϵ- closure(0) = {0, 1, 2, 4, 7}

denote as A

$\epsilon\{T(A, a)\} = \epsilon\{3, 8\} = 3, 8, 6, 7, 1, 2, 4$

as B

$\epsilon\{T(A, b)\} = \epsilon\{5, \cdots\} = 5, 6, 7, 1, 2, 4,$

as C

Take B and C repeat same process

$\epsilon\{T_{B\,a}\} = \epsilon\{3, 8\} = 3, 8, 6, 7, 1, 2, 4 = B$

$\epsilon\{T_{B\,b}\} = \epsilon\{5, 9\} = 5, 6, 7, 1, 2, 4, 9 = D$

$\epsilon\{T_{C\,a}\} = \epsilon\{3, 8\} = B$

$\epsilon\{T_{C\,b}\} = \epsilon\{5\} = C$

$\epsilon\{T_{D\,a}\} = \epsilon\{3, 8\} = B$

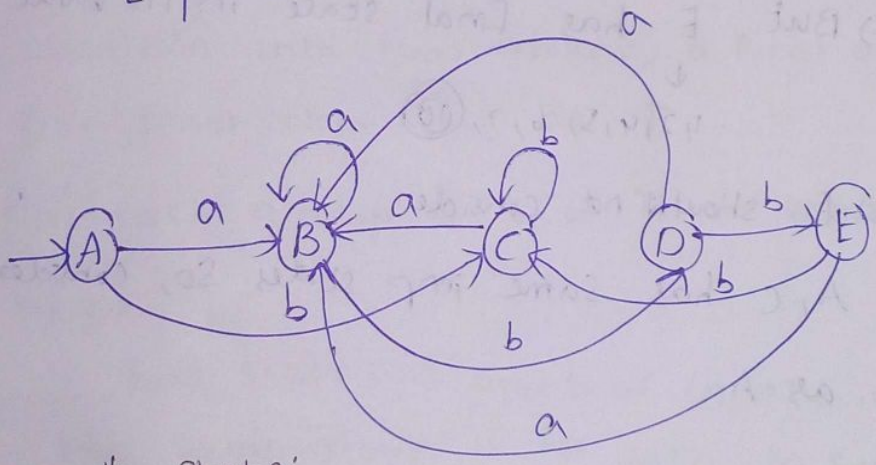
$\epsilon\{T_{D\,b}\} = \epsilon\{5, 10\} = \{5, 6, 7, 1, 2, 4, 10\}$

$= E$ call as E

$\epsilon\{ {}^{T}_{E}{}_{a} \} = \epsilon\{3,8\} = B$

$\epsilon\{ {}_{E}{}^{T}{}_{b} \} = \epsilon\{5\} = C$

	a	b
A	B	C
B	B	D
C	B	C.
D.	B	E
E	B	C



Important State:-

A state is said to be an imp state if it has non ϵ out transition

5/12/17

(i) method Subset construction:-

→ minimization will be done by identifying the imp state in the NFA;

∴ Imp states in NFA = 2, 4, 7, 8, 9

imp states

A(0,1,2,4,7) : 2,4,7

B(1,2,3,4,6,7,8) : 2,4,7,8

C(1,2,4,5,6,7) : 2,4,7

D(1,2,4,5,6,7,9) : 2,4,7,9

E(1,2,4,5,6,7,10) : 2,4,7

The states which contain same imp states will consider as single state.

→ Here A, C, and E has same imp states

→ But E has final state in its value

↓

1,2,4,5,6,7,(10)

→ E should not consider.

→ A, C has same imp states so, consider C as A

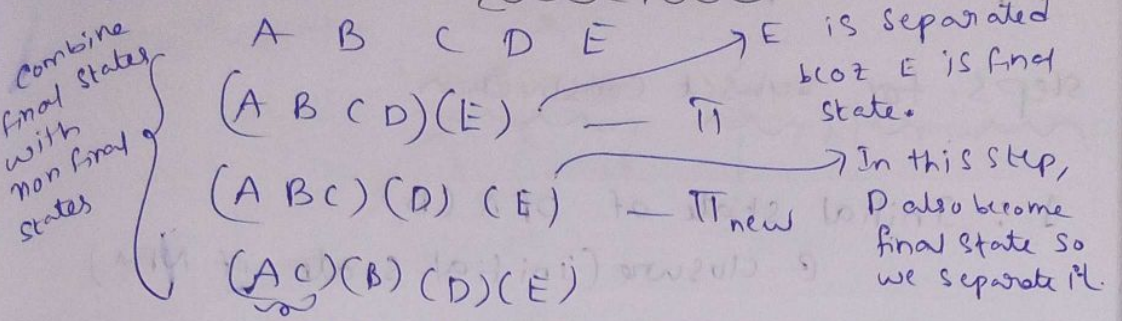
→ C will be discarded

→ In the place of C put A.

	a	b	
A	B	C	A
B	B	D	
~~C~~	~~B~~	~~C~~	
D	B	E	
E	B	C	A

Idi chudu

(ii) Partition method: Partition Method:



Steps: (Partition Algorithm (or) Minimize states of DFA)

1: If transition is not defined add dead state d, and will have transition to itself on all input symbols.

2: partition into two groups, a final and non final Name this as Π.

3: For each group of G of Π do ~~begin~~

begin

two states s and t of G are in the same group if for each i/p symbol a, s and t have transitions to the same sub group of Π. Name this as Π_{new}

4: If $\Pi_{new} \neq \Pi$ then $\Pi = \Pi_{new}$ and repeat step 3.

5: Remove all the dead states.

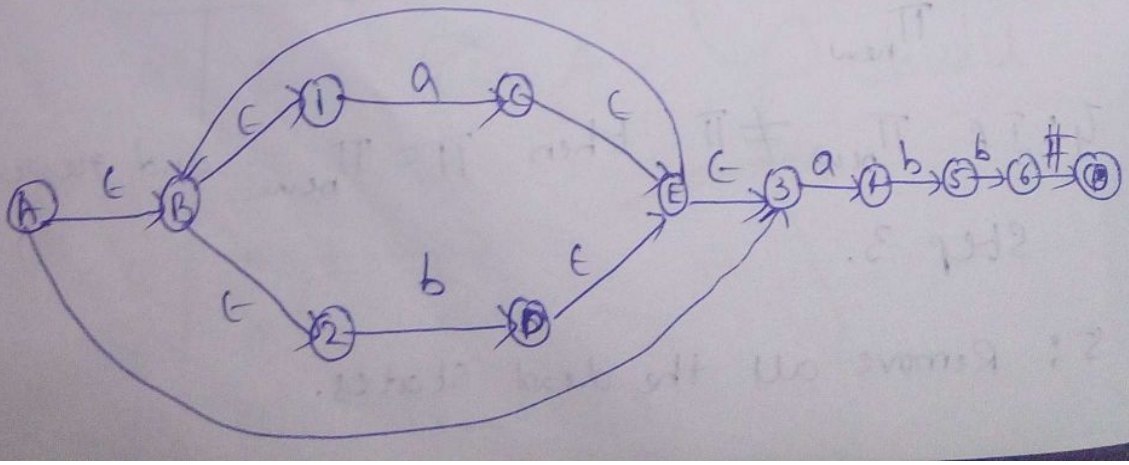
6: pick a representative from each group.

steps for subset construction:

1: Initial state of DFA (D)
ε-closure (Initial state of NFA)
unmark this state.

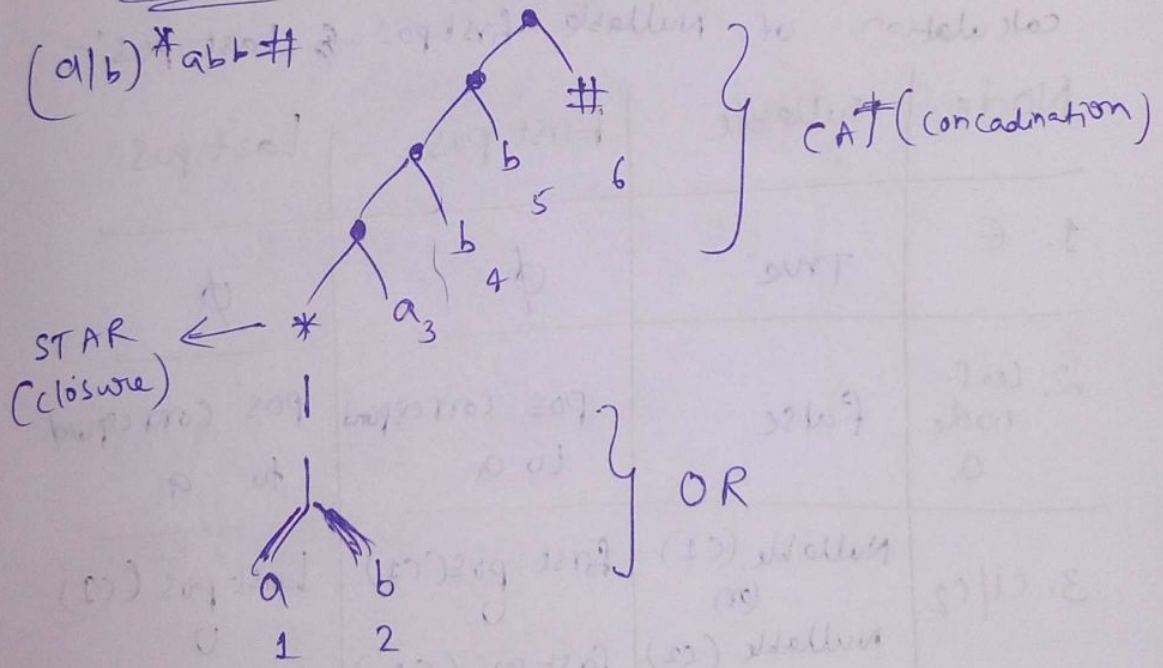
2: while there is an unmarked state x in D do
begin mark x
for each input symbol a
find the transition from X.
Let T be the transition from X on a.
$y = \epsilon\text{-closure}(T)$ $\{y = \epsilon(T_A a)\}$
if y is not empty and not present in the states of DFA, add y and unmark y.
end

Optimization of DFA based pattern:
RE → DFA → Minimization [NO - NFA]



Syntax tree

$(a|b)^*abb\#$



Represent leaf nodes with integers

First pos (match first symbols of strings - RE)

last pos (Set of positions that are corresponds to last node)

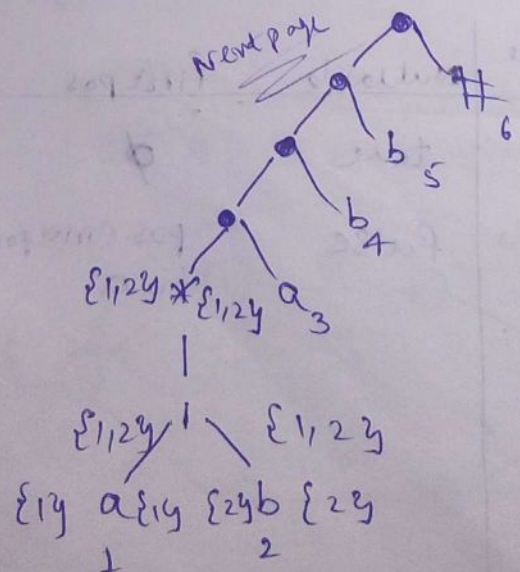
follow pos

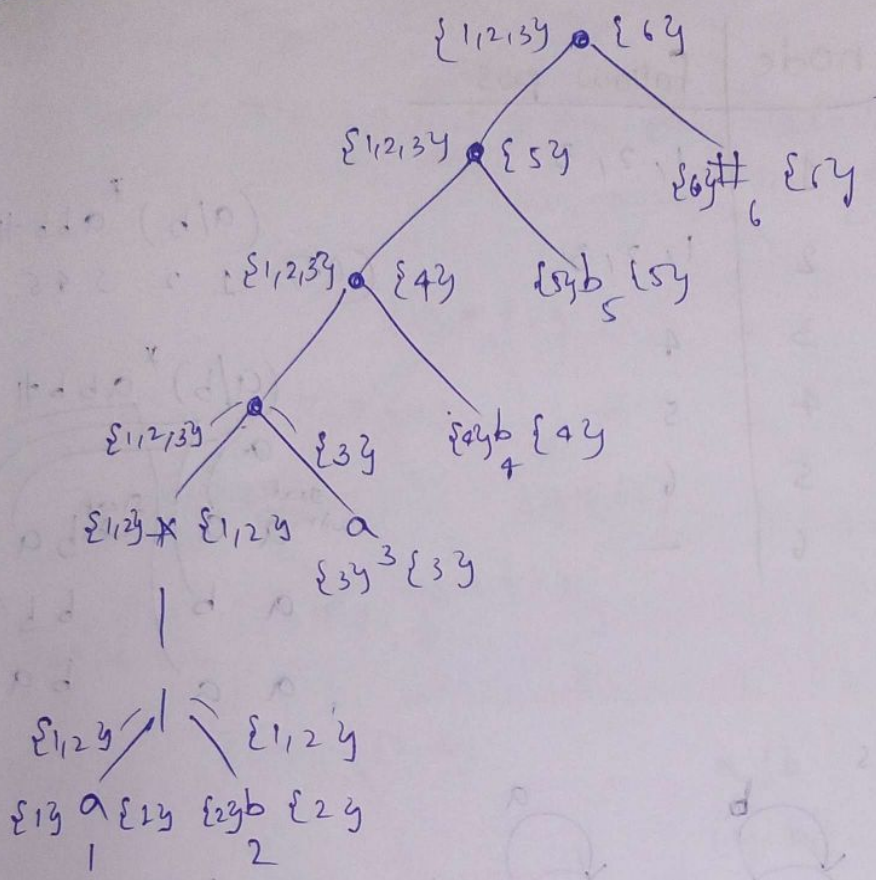
Nullable. (If a node is said to be nullable if the R.E generates ϵ)

6/12/17

calculation of Nullable, firstpos & lastpos:-

Node	Nullable	First pos	Last pos
1. ϵ	True	ϕ	ϕ
2. Leaf node a	false	pos correspond to a	pos correspond to a
3. c1/c2	Nullable (c1) (or) Nullable (c2)	first pos(c1) ∪ first pos (c2)	Last pos (c1) ∪ Last pos (c2)
4. CAT c1c2	Nullable(c1) and Nullable (c2)	if not nullable (c1) then first (c1) else first(c1) ∪ first (c2)	if not nullable(c2) then Last (c2) else last (c1) ∪ last (c2)
5. * node	True	First pos (c1)	last pos (c1)





Calculation of followpos(i) :-

1. $C_1 C_2$ is a CAT node

   if i is in lastpos(C_1)

   followpos(i) = firstpos(C_2)

2. If C_1 is a * node and i is in lastpos(C_1)

   then

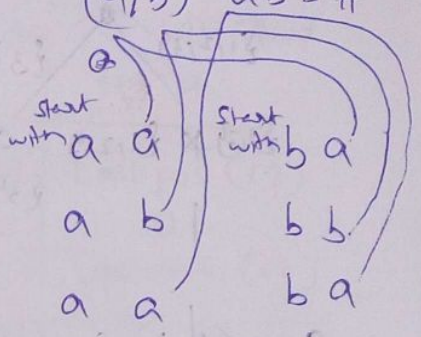
   followpos(i) = firstpos(C_1)

node	follow pos
1	1, 2, 3
2	1, 2, 3
3	4
4	5
5	6
6	—

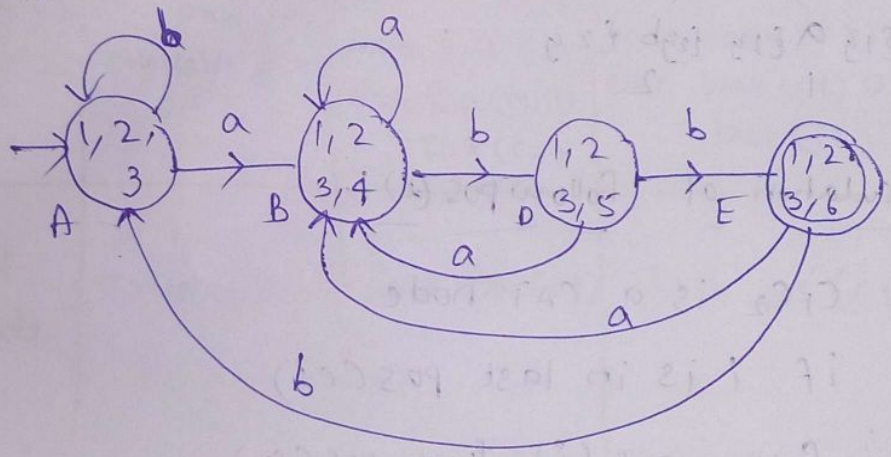
$(a/b)^* abb\#$

(or) 1 2 3 4 5 6

$(a/b)^* abb\#$



DFA is



	a	b
→ A	B	A
B	B	D
D	B	E
(E)	B	A

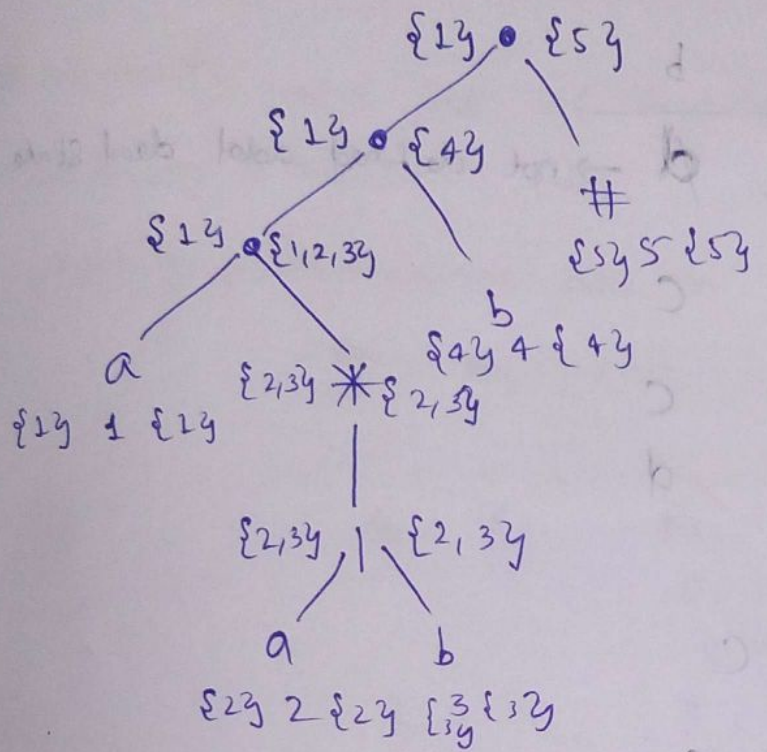
The state which includes #(6) is final state

Idli Chudu

3/12/19

(Q) $a(a|b)^* b$

Syntax tree → first & last pos → follow pos → DFA



a | b

(a, b)

→ Draw syntax tree
→ Represent leaf nodes with integers
→ write first and last pos of nodes.
→ write follow position in the form of table.

follow pos

Node	Follow pos
1	2, 3, 4
2	2, 3, 4
3	2, 3, 4
4	5
5	—

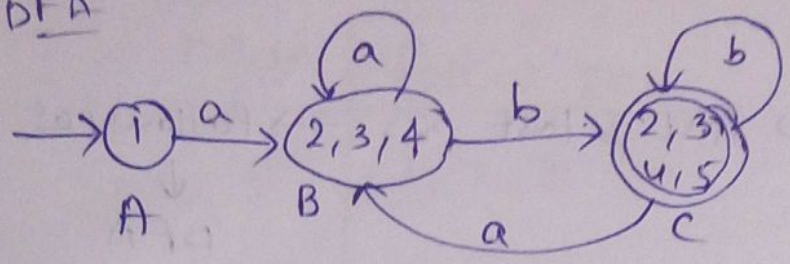
(or)

choose follow pos by following

$a(a|b)^* b\ \#$

1 2 3 4 5

DFA



C is final bcoz C contains # (5)

	a	b
A	B	d → not defined add dead state d
B	B	C
C	B	C
d	d	d

Minimization

A B C

⇒ (A B d) C

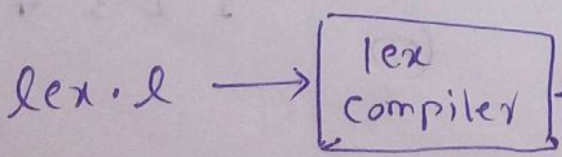
⇒ (A d) (B) (C)

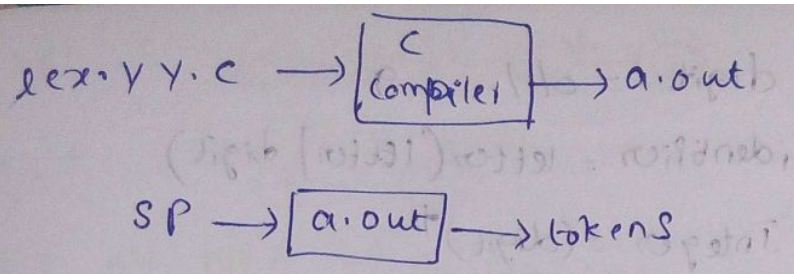
⇒ (A) (B) (C)

Lexical Analyser (lex process)

Lex → in C

not imp Flex } generates lexical analyser in C





Lexical Analyser contains set of auxilary definitions followed by translation rules.

Auxilary definitions are in the form

$D_1 = R_1$

$\vdots$

$D_i = R_i$ — R_i is a RE; D_i is distinguish name to the RE

$\vdots$

$D_n = R_n$ — Σ, previously define RE $R_i = \Sigma \cup D_1, D_2, \ldots D_{i-1}$

Translation rules are in the form

$P_1 \{A_1\}$

$\vdots$

$P_i \{A_i\}$ — P_i is a pattern or RE; A_i is action to be performed when particular RE is identified

$\vdots$

$P_m \{A_m\}$

belongs to Auxilary definitions! based on previously define RE

$\Sigma = A, \ldots Z, 0, \ldots 9, +, -, \bullet$

letter = A | B | ... Z

digit = 0/1/---9

identifier = letter(letter | digit)

Integer = (digit)$^+$

Sign = +/-/ϵ

Signed Integer = Sign Integer

decimal = signed Integer • Integer (+4.12) Eg:

= signed Integer • (+45.)

= Sign • Integer (+.23)

Exponential = (decimal / signed_Integer) E ↓ signed_Integer

Real number = decimal / exponential

Eg:- Σ a, b

Auxiliary definitions

{ }

Transition Rules

a { }

abb { }

$a^* b^+$ { }

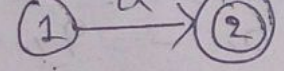
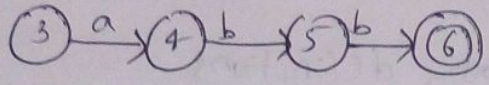
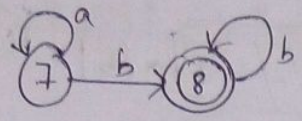
Sol: a = 

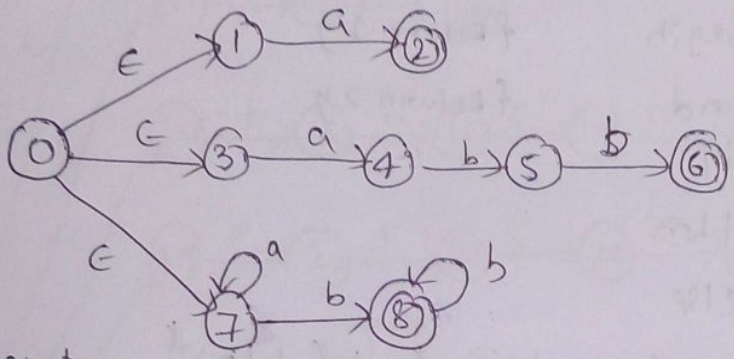
abb :



a^*b^+ :



Join all NFA with new state with ∈ to get final NFA



Initial state of DFA is '0' along with ∈ values

	a	b	token
0137	2,4,7	8	—
2,4,7	7	5,8	a
8	ϕ	8	a^*b^+
7	7	8	—
5,8	ϕ	6,8	a^*b^+
6,8	ϕ	8	abb

token is based upon the final states in the states. Eg: 0137 has no final & 247 has 2 final state.

6,8 → Two final so, write first one

Eg:-

Auxilary definitions

L ~~letter~~ A|B|---|Z|a|b|---z

D ~~digit~~ 0|1|---9

Transition Rules

begin	{Return 1}
end	{Return 2}
if	;
then	;
else	;

[letter (letter|digit)*] L(L|D)*

~~digit~~+ D+

<

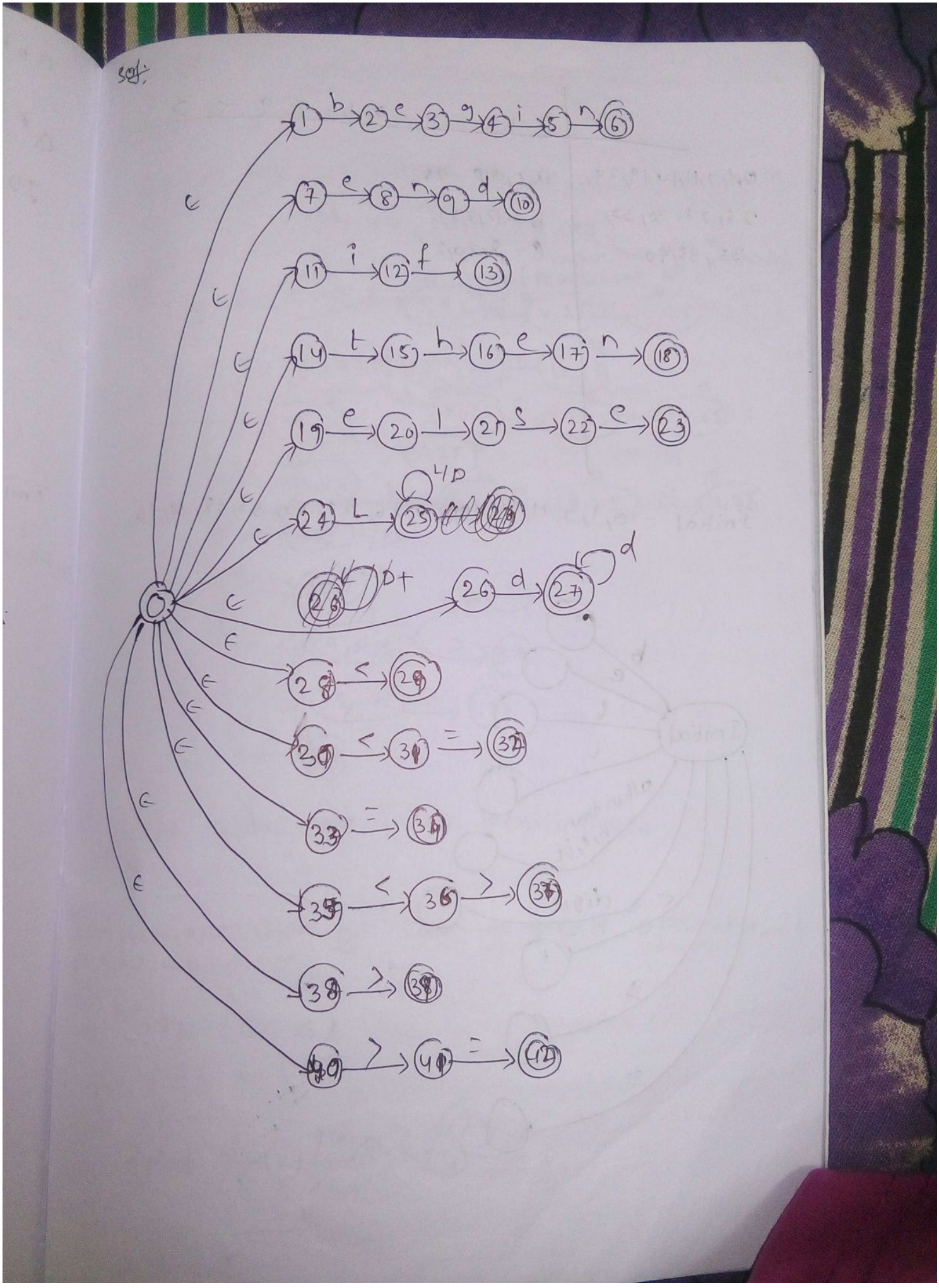
<=

=

<>

>

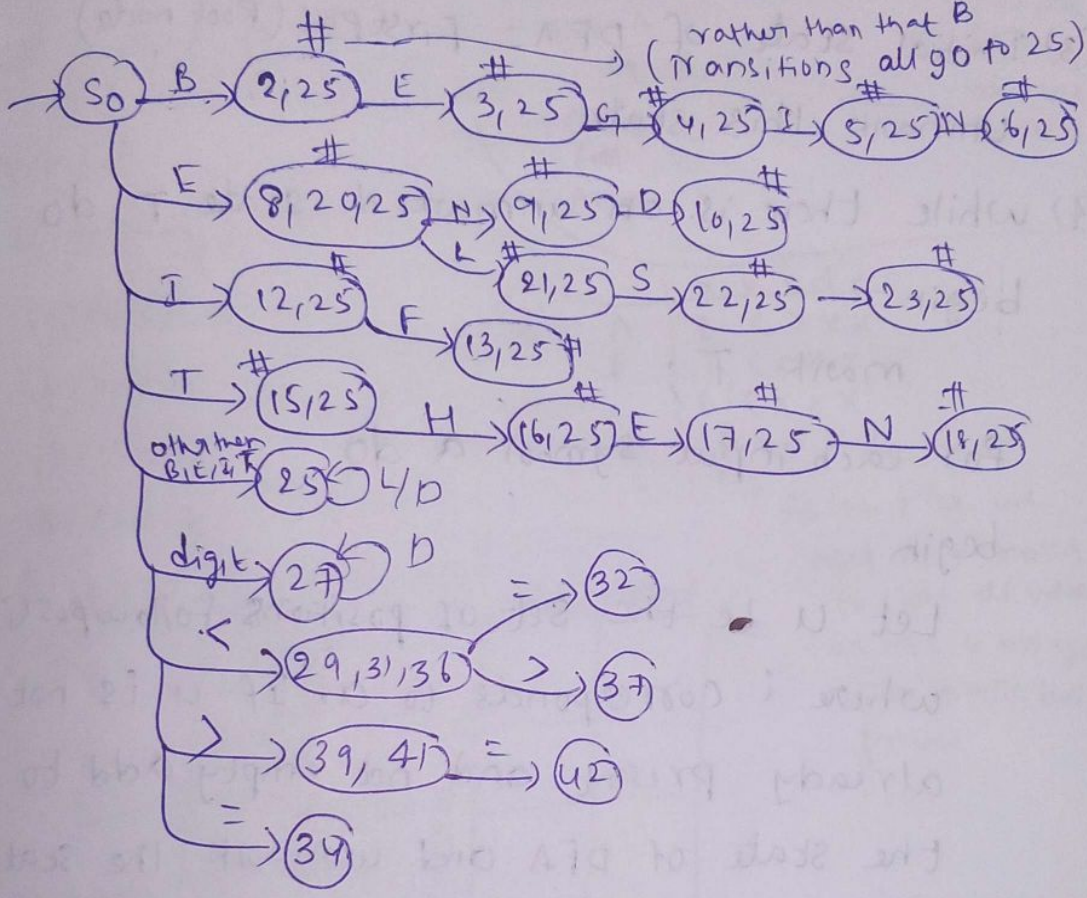
>=



11/12/17

DFA is

Initial are 0, 1, 7, 11, 14, 19, 24, 26, 28, 30, 33, 35, 38, 40 name as S_0

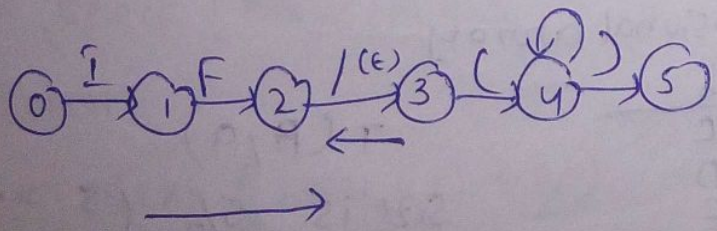


final states are (check longest prefix)

Implementation of Look a head operator:-

If (A = B)

IF A = S.O



Algorithm for optimization of DFA based Pattern

(1) Syntax tree

(2) First, last, followpos for leaf nodes.

(3) Initial state of DFA = Firstpos (Root node)
unmark this state

(4) while there is an unmarked state T do
begin
mark T;
for each input symbol a do
begin
Let U be the set of positions followpos(i)
where i corresponds to a. If U is not
already present and not empty add to
the state of DFA and unmark the state
Draw an edge for $T \xrightarrow{a} U$
end

12/12/17

Data Structures used to represent transition table :→ (1) Two dimensional array (2) Lists (3) Compaction

(1) Two dimensional array

$(a|b)^*abb$

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

T(A, a)

Size is 5/2 [5 = State, 2 = input symbol]

(2) lists

	a	b	c	d	e
A	B	C	C	C	C
B	B	D	D	D	D
C	B	C			
D	B	E			
E	B	C			

(since default entry is placed at the end)

Eg:- State A

a	B
b	C
c	C
d	C
e	C

All same so put default at input symbol b ←

State A

a	B
default (or) Any	C

b) Compaction:-

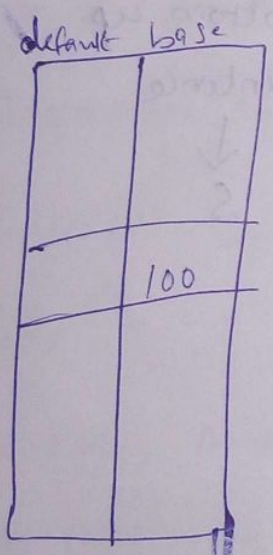
4 arrays

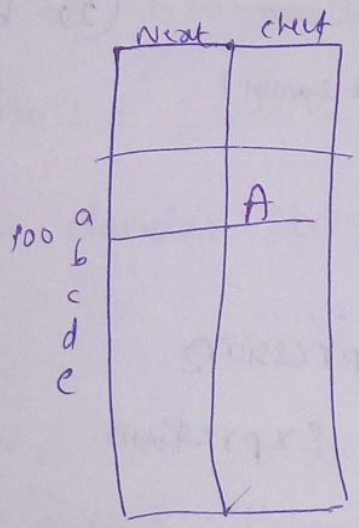
(1) default
(2) base
(3) Next
(4) check

Eg:-

	a	b	c	d	e
A	B	C	X	X	X
B	B	C	X	X	X
C	B	C	X	X	X
D	B	C	X	X	X
E	B	C			

for this eg we need 20 memories to store all values. we use 4 arrays to minimize this process.





~~Procedure~~ Algorithm for compaction:-

```
Procedure Nextstate(s, a)
if (check(base(s) + a) = s)
Return
    Next(base(s) + a)
```

else

Return (NextState (default (s), a))

end

UNIT-2

Syntax Analysis or parsing

→ we use CFG in this.

(N T P S)

(1) top down

S (Start symbol)

↓

Sentence

(2) bottom up

Sentence

↓

S

Eg:- E - Expression

id is an expression

Con " " "

Exp Con Exp

E → id

E → E+E

E → E*E

E → -E

E → (E)

1. NT (Non Terminals)

(a) Capital letters beggining of alphabet

(b) Lower case name expression, Statement

(c) S used as start symbol

(2) Terminals:-

(a) Small letters a, b, c, ---

(b) digits 0, 1, -- 9

(c) operators

d) punctuation symbols

(e) id (or) if

(3) X, Y, Z denote the Grammar symbol

Grammar symbols = Terminal / Non terminal

(4) u, v, w --- denotes strings of Terminals

(5) α, β, γ --- " " " grammar Symbols.

(6) $A \to \alpha_1$, $A \to \alpha_2$ ---- $A \to \alpha_1 / \alpha_2 / \ldots \alpha_n$

(7) The left hand side of first production is start symbol when start symbol S is not given, mention explicitly.

Ambiguous grammer:

The grammar contain more than one left most (or) right most derivations is called as "Ambiguous Grammar"

Eg:- $E \to E+E \mid E*E \mid (E) \mid id$

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

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

This is ambiguous.

Eg:-

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

$E \to E+T$
$\to T+T$
$\to F+T$
$\to F+F$
$\to id+F$
$\to id+id$

$E \to T$
$E \to F$
$\to id$

Not ambiguous.

14/12/17

(Q) $S \to (L) \mid a$
$L \to L,S \mid S$
derive $((a,a),a,(a))$

→ The combination of T and NT is called Sentencial form

→ The LMD is called left most Sentencial form.

→ The RMD – Right most Sentencial form

Left most derivation (LMD)

$S \to (L)$

$S \to (L,S)$

$S \to (S,S,S)$

$\to (S,S,S)$

$\to ((L),S,S)$

$\to ((LS),S,S)$

$\to ((S,S),S,S)$

$\to ((a,S),S,S)$

$\to ((a,a),S,S)$

$\to ((a,a),a,S)$

$\to ((a,a),a,(L))$

$\to ((a,a),a,(S))$

$\to ((a,a),a,(a))$

Right most derivation (RMD)

$S \to (L)$

$S \to (L,S)$

$S \to (L,(L))$

$S \to (L,(S))$

$S \to (L,(a))$

$S \to (L,S,(a))$

$S \to (L,a,(a))$

$S \to (S,a,(a))$

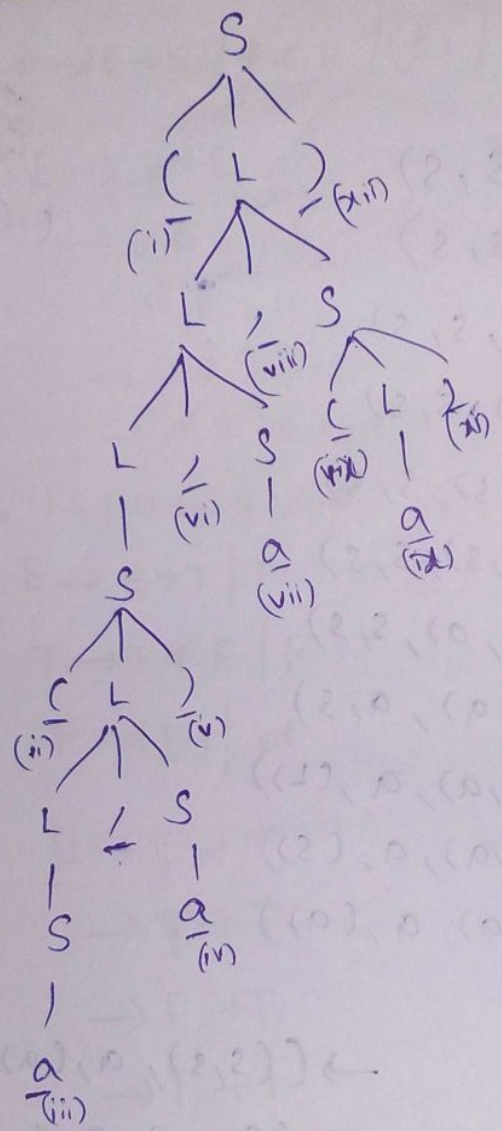
$S \to ((L),a,(a))$

$S \to ((L,S),a,(a))$

$\to ((S,S),a,(a))$

$\to ((a,S),a,(a))$

$\to ((a,a),a,(a))$



∴ ((a,a), a, (a))

The leaf nodes of the tree from left gives the yield (or) fountier of the tree

(Q) $S \to S+S \mid SS \mid (S) \mid S* \mid a$

derive $(a+a)*a$

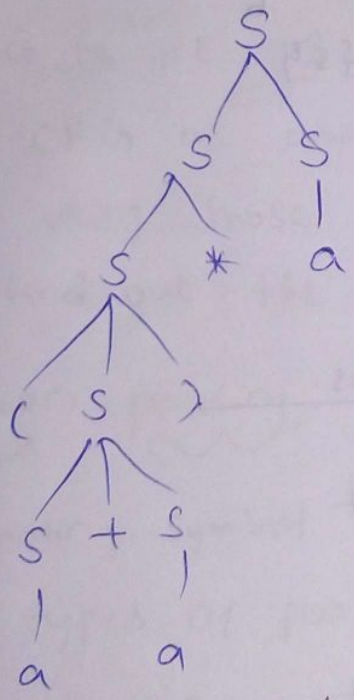
LMD

$S \to SS$
$\to S* S$
$\to (S)* S$
$\to (S+S)* S$
$\to (a+S)* S$
$\to (a+a)* S$
$\to (a+a)* a$

RMD

$S \to S S$
$\to Sa$
$\to S*a$
$\to (S)*a$
$\to (S+S)*a$
$\to (a+S)*a$
$\to (a+a)*a$

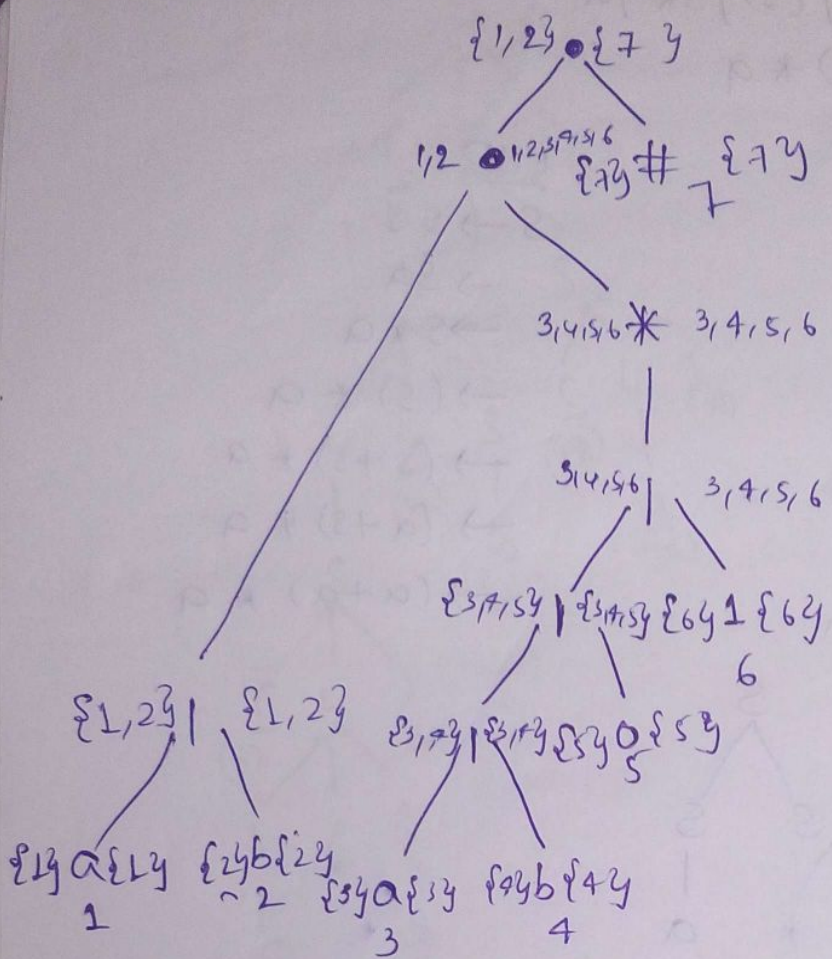
tree



(Q) $(a|b)(a|b|0|1)^*$

Regular expression to CFG [DFA - CFG process]

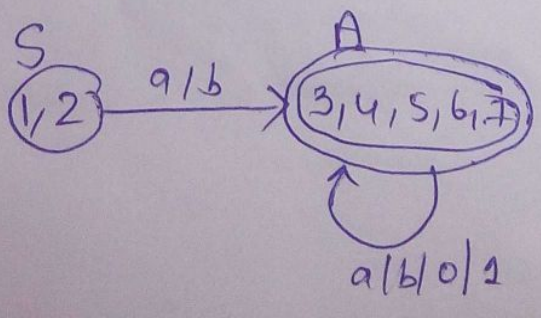
In next page



Follow pos

	Follow pos
1	3, 4, 5, 6, 7
2	3, 4, 5, 6, 7
3	3, 4, 5, 6, 7
4	3, 4, 5, 6, 7
5	3, 4, 5, 6, 7
6	3, 4, 5, 6, 7
7	~~3, 4, 5, 6, 7~~ —

DFA is



includes 7 is final state.

Start symbol — S

NT — S, A

T — a, b, 0, 1

P — S → aA

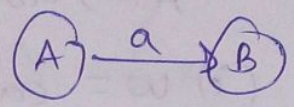
S → bA

A → aA/bA/

0A/1A

A → ∈

if



then

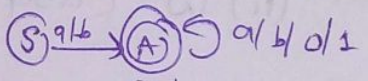
Production should be written as

A → aB

Reverse

CFG to RE is

Take CFG as above one, then we create a DFA using those productions and then we find out the regular expression.



(a/b) (a/b/0/1)*

TOP down parsing:

Starting symbol ⁼ to Sentence (we obtain sentence from start symbol).

Three types of parsing

(1) Brute force parsing — using backtracking.

(2) Recursive descent parsing

(3) Predictive parsing

(1) Brute force parsing:

disadvantages
→ Back tracking
→ Order of productions
→ Left Reccursion

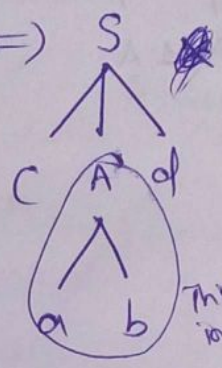
Eg:- $S \rightarrow cAd$

$A \rightarrow ab/a$

(i) $w = cad$

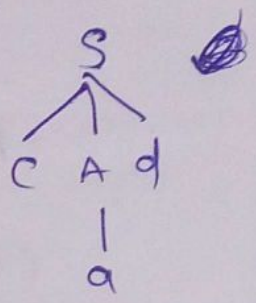
Leftmost
i.e.,
First take
ab in A
and then take a

$S \rightarrow cAd \Rightarrow$



This is incorrect

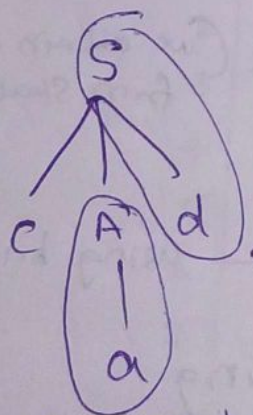
Mismatch
it gives
cabd ✗



Match
it gives
cad

(ii) $w = cabd$ $S \rightarrow cAd$

$A \rightarrow a/ab$



It mismatches

It gives a it crct

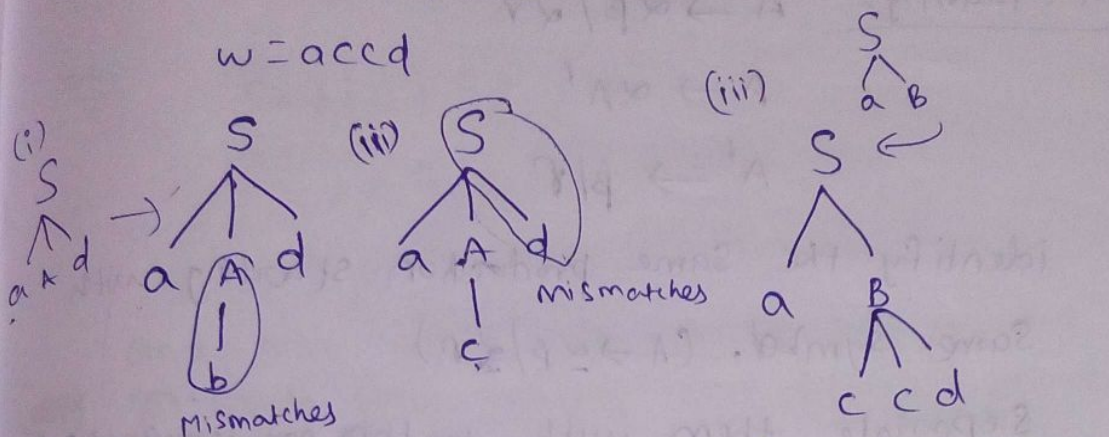
∴ Order of alternative is important in this.

Eg:- $S \to aAd \mid aB$

$A \to b \mid c$

$B \to ccd \mid ddc$

$w = accd$



gives crct answer

accd

→ If $A \to A\alpha$ occurs then it goes to infinite loop ↓ then it becomes disadvantage. (left recursion)

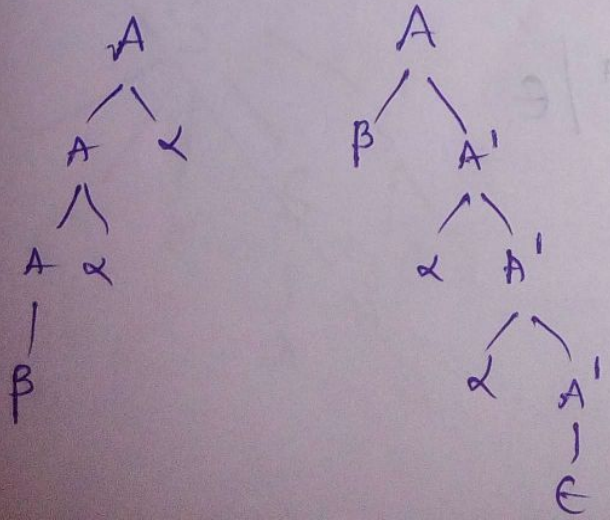
23/12/17

Left Recursion

$A \to A\alpha \mid \beta$

$A \to \beta A'$

$A' \to \alpha A' \mid \in$ } These two are same as $A \to A\alpha \mid \beta$



Left factoring:- $A \to \alpha\beta / \alpha\gamma$

$A \to \alpha A'$

$A' \to \beta / \gamma$

identify the same production starting with same symbol. ($A \to \underline{\alpha}\beta / \underline{\alpha}\gamma$)

Separate them with another non-terminal, (A')

$A \to A\alpha_1 / A\alpha_2 / \cdots A\alpha_n / \beta_1 / \beta_2 / \cdots \beta_k$

$A \to \beta_1 A' / \beta_2 A' / \cdots \beta_k A'$

$A' \to \alpha_1 A' / \alpha_2 A' / \cdots \alpha_n A' / \epsilon$

(Q) Left Recurse this problem

$S \to Aa / b$

$A \to Ac / Sd / e$ ⟶ $A \to \underset{\alpha_1}{Ac} / \underset{\alpha_2}{Aad} / \underset{\beta_1}{bd} / \underset{\beta_2}{e}$

Sol:

$S \to Aa / b$

$A \to bdA' / eA'$

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

(Q) $S \rightarrow iCtS / iCtSeS / a$

$C \rightarrow b$

left factor this problem.

Sol: $S \rightarrow iCtSS' / a$

$S' \rightarrow eS / \epsilon$

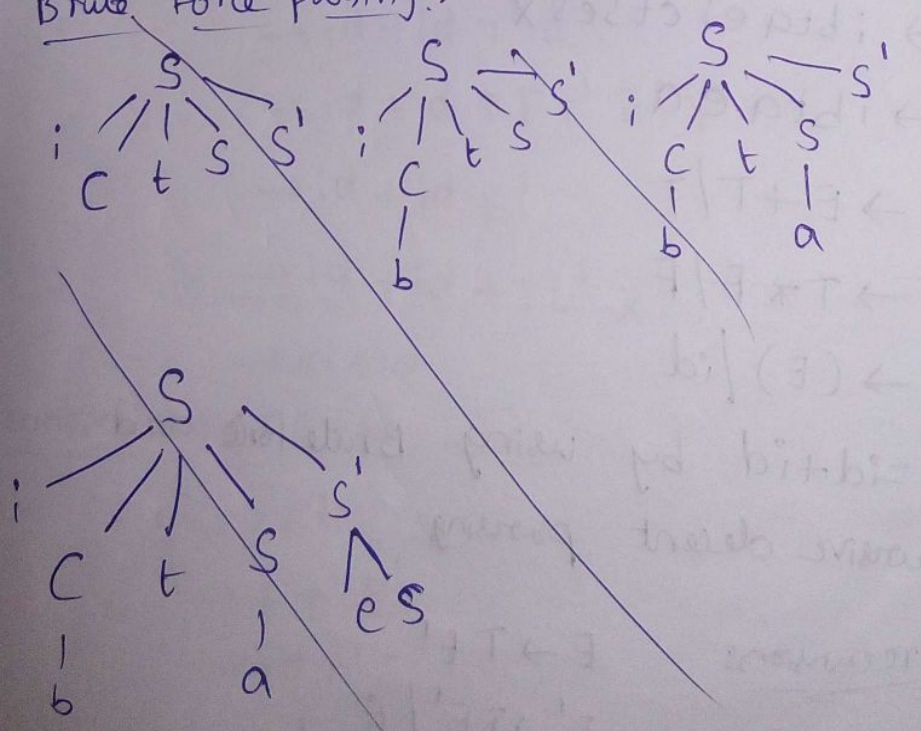
$C \rightarrow b$

and also check the string $w = ibtaea$ in brute force parsing and Recursive decent parsing.

$S \rightarrow iCtSS'$
$ibtSS'$
$ibtaS'$
$ibtaeS$
$ibtaea$

(2) Recursive decent parsing without back tracking

Brute force parsing:



$S \rightarrow ictS$

$\rightarrow ibtSX$

$\rightarrow ibticts X$

$\rightarrow ibtitses X$

$\rightarrow ibtax$

$S \rightarrow ictSeS$

$\rightarrow ibtSeS$

$\rightarrow ibticts eS \quad X$

$\rightarrow ibticts eses \quad X$

$\rightarrow ibtaeS$

$\rightarrow ibtaeictsX$

$\rightarrow ibtaeitsesX$

$\rightarrow ibtaea$

(Q) $E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id$

Check $w = id+id$ by using Brute force and Recursive decent parsing.

left recursions $E \rightarrow TE'$

$E' \rightarrow TE'/\epsilon$

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

Brute force :

$E \rightarrow TE'$

$\rightarrow FT'E'$

$\rightarrow (E)\, T'E'X$

$\rightarrow id\, T'E'$

$\rightarrow id * FT'\, E'\, X$

$\rightarrow id\, E'$

$\rightarrow id + TE'$

$\rightarrow id + FT'\, E'$

$\rightarrow id + (E)\, T'\, E'\, X$

$\rightarrow id + id\, T'E'$

$\rightarrow id + id * FT'\, E'\, X$

$\rightarrow id + id\, E'$

$\rightarrow id + id + TE'\, X$

$\rightarrow id + id$

Recursive decent parsing :

$E \rightarrow TE'$

$\rightarrow FT'E'$

$\rightarrow id\, T'E'$

$\rightarrow id\, E'$

→ id+TE'
→ id +FT'E'
→ id+id T'E'
→ id+id E'
→ id+id

(3) predictive parsing :-

Tabular implementation of Recursive decent parsing.

Explination using above example

Predictive parsing table :-

	id	+	*	(	)
E	TE'				
E'		+TE'			
T					
T'					
F					

TO construct the predictive parsing table, we have two components.

(1) First

(2) Follow

26/12/17

(1) First (X)

Set of terminals that begin the strings that that are derivable from X.

rules:- (1) if x is a terminal first(X) = X

(2) $X \to a\alpha$ then first $(X) = a$

(3) $X \to X_1 X_2 X_3 -- X_n$ where each X_i is a grammar symbol

if X_i does not contain $\in$

then

add all non $\in$ symbols from X_1 to X_{i-1} and First (X_i)

2) Follow(B) :-

Set of terminals that can appear after B.

rules:

1. $A \to \alpha B \beta$

   Follow(B) = First(β)

2. $A \to \alpha B$ (or) First(β) contains ϵ

   Follow(B) = Follow(A)

3. $ is in Follow(s) where S is start symbol.

(Q) $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

Calculate first and follow for each NT and draw predective parsing table

Sol: Left recursion:

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

[First left recursion then do left factoring and find first & follow, and at last find table]

	First	Follow
E	(, id	), $
E'	+, ϵ	), $
T	(, id	+,), $
T'	*, ϵ	+,), $
F	(, id	*, +,), $

Follow(E)
First choose E is in right side
$F \to (E)$
$\alpha B \beta$
)

Follow(E')
Choose E' in right side
$E \to TE'$ $E' \to +TE'$
$\alpha B \beta$ $\alpha B \beta$
)

Follow(T)
Choose T in right side
$E' \to +TE'$ $E \to TE'$
$\alpha B \beta$ $\alpha B \beta$

Follow(T')
Choose T' in right side
$T \to FT'$ $T' \to *FT'$
$\alpha B \beta$ $\alpha B \beta$

Follow(F)
Choose F in right side
$T \to FT'$ $T' \to *FT'$
$\alpha B \beta$ $\alpha B \beta$
$\$$

Rules for this table are in next page

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

Rules for filling table contains NT & T:

1. For each production of type $A \to \alpha$ do steps 2 and 3
2. For each terminal a in First(α) add $M[A, a] = A \to \alpha$
3. If First(α) contains ϵ then for each b in Follow(A), set the entry on $M[A, b] = A \to \alpha$

$A \to \alpha$

$E \to TE'$

First(TE') = (, id

$M[E, (] = E \to TE'$

$M[E, id] = E \to TE'$

$A \to \alpha$

$E' \to +TE'$

First(α) = +

$E' \to \epsilon$

$T \to FT'$

First(F)

$T' \to *FT'$

First(*)

$T' \to \epsilon$

$F \to (E)$

$F \to id$

→ All undefined entries in table are error entries.

→ If there are are no multiply defined entries, then the grammar is called LL(1) grammar.

id+id$

$E \to TE'$

$\to FT'E'$

$\to idT'E'$

$\to idE'$

$\to id+TE'$

$\to id+FT'E'$

$\to id+idT'E'$

$\to id+id\epsilon$

$\to id+id$

These steps are done by choosing predective parsing method table.

b is in $A \to \alpha$

28/12/17

Q) construct the predictive parsing table for the grammer.

$S \to iCtSeS \mid iCtS \mid a$

$C \to b$

Sol) ~~Left factoring~~ left factoring:

$S \to ictSS' \mid a$

$S' \to eS \mid \epsilon$

$C \to b$

	First	follow
S	i, a	e, $
S'	e, ϵ	e, $
C	b	t

	i	t	e	a	b	$
S	$S \to ictss'$			$S \to a$		
S'			$S' \to eS$ $S' \to \epsilon$			$S' \to \epsilon$
C					$C \to b$	

The grammar is not LL1 grammer.

Second method for checking LL1 grammer:

$A \to \alpha / \beta$

2. At most one of α or β can derive ϵ

3. if $\alpha \to \epsilon$ (α derives ϵ)

$First(\beta) \neq Follow(A)$

↳ is not equals to

1. $First(\alpha) \neq First(\beta)$

If the above rules are not satisfied, then Grammar is not LL1 grammer

Sol: For above problem

$S \to ictss' / a$ → 1st condition satisfied

This S doesnot have derive ϵ so no need to check rule 2 and 3

$S' \to es / \epsilon$

↳ This is not satisfied.

So, the grammar is not LL1 grammar.

(a) $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

using 2nd method.

sol: Left recursion:

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

→ draw first & follow table.

The above NT are satisfies the rules of 2nd method.

So, it is LL1 grammer.

(i) check string id+id

→ pop top of the stack and push elements in reverse order.

$ E	id+id $
$ E'T	id+id $
$ E'T'F	id+id $
$ E'T'id	id+id $
$ E'T'	+id $
$ E'	+id $
$ E'T+	+id $

$ E'T	id $
$ E'T'F	id $
$ E'T'id	id $
$ E'T'	$
$ E'	$
$	$

Accepted i.e., it belongs to Language

30/12/17

(ii) Check the string id + * id $

stack	i/p
$ E	id + * id $
$ E'T	id + * id $
$ E'T'F	id + * id $
$ E'T'id	id + * id $
$ E'T'	+ * id $
$ E'	+ * id $
$ E'T+	+ * id $
$ E'T	* id $ Error

In this i/p, id + * id we have operators next to next so, it becomes an error

Algorithm for predictive parsing:

Let X be the top stack symbol and a be the current symbol

Repeat

if M[A,a] is a terminal (or) $ then

and if X = a then pop X, remove a

else

error()

else

if $M[A, a] = X \rightarrow X_1 X_2 \cdots X_K$ then pop X

push $X_K X_{K-1} \cdots X_1$ (in reverse order)

else

error()

unit X = a = $

Error recovery procedures for predictive parsing:

There are two error recovery procedures. They are:-

(1) panic mode of error recovery

(2) Error routines method

(1) panic mode of error recovery:

(Q) $E \to E+T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid id$

Sol: first write the (i) First & follow table and
(ii) predictive parsing table

	First	Follow
E	(, id	), $
E'	+, ∈	), $
T	(, id	+,), $
T'	*, ∈	+,), $
F	(, id	*, +,), $

write sync in the pp table in the follow position operands.

pptable:

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	sync	sync
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	sync		$T \to FT'$	sync	sync
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$	sync	sync	$F \to (E)$	sync	sync

check) id * + id $

$ E	) id * + id $	(remove ')' because in pp table it has the token sync) :
$ E	id * + id $	
$ E' T	id * + id $	
$ E' T' F	id * + id $	
$ E' T' id	id * + id $	
$ E' T'	* + id $	
$ E' T' F *	* + id $	
$ E' T' F	+ id $	(F at + there is sync so eliminate +)
$ E' T' F	id $	
$ E' T' id	id $	
$ E' T'	$	
$ E'	$	
$	$	

(2) Error Routines method:-

	id	+	*	(	)	$
E	$E \to TE'$	e_1	e_1	$E \to TE'$	e_1	e_1
E'	$E' \to \epsilon$	$E' \to +TE'$	$E' \to \epsilon$	$E' \to \epsilon$	$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	e_1	e_1	$T \to FT'$	e_1	e_1
T'	$T' \to \epsilon$	$T' \to \epsilon$	$T' \to *FT'$	$T' \to \epsilon$	$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$	e_1	e_1	$F \to (E)$	e_1	e_1
id	pop	-	-	-	-	-
+	-	pop	-	-	-	-
*	-	-	pop	-	-	-
(	-	-	-	pop	-	-
)	e_2	e_2	e_2	e_2	pop	e_2
$	e_3	e_3	e_3	e_3	e_3	Accept

e_1 : missing operand insert id

e2 : missing right Paranthesis

e3 : extra symbols in the input

check for id +) $

$ E	id +) $
$ E' T	id +) $
$ E' T' F	id +) $
$ E' T' id	id +) $
$ E' T'	+) $
$ E'	+) $
$ E' T +	+) $
$ E' T	id) $
$ E' T' F	id) $
$ E' T' id	id) $
$ E' T'	) $
$ E'	) $
$	) $
$	$

[e1: at T) is e1 so insert id before]

[e3: extra symbol in input so eliminate it]

Q) Find the predictive parsing for

$r\text{expr} \rightarrow r\text{expr} + r\text{term} \mid r\text{term}$

$r\text{term} \rightarrow r\text{term}\ r\text{factor} \mid r\text{factor}$

$r\text{factor} \rightarrow r\text{factor}\ * \mid r\text{primary}$

$r\text{primary} \rightarrow a \mid b$