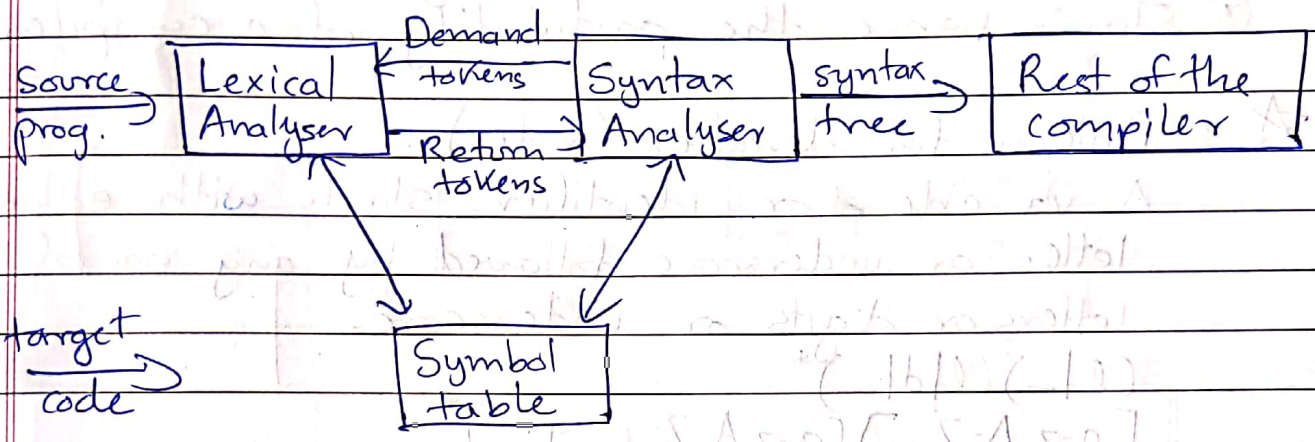
Lexical Analysis:-

Role of LA:-



Functions of LA:-

1. It produces stream of tokens.
2. It removes white space characters
3. It generates symbol table which stores the information about identifiers, constants, encountered in the input.
4. It keeps track of line numbers.
5. It reports the error encountered while generating the tokens.
6. It provides error messages with corresponding line numbers & column numbers.

Scanned with CamScanner

l -> letters
d -> digits
_ -> underscore

Seperation of lexical from Syntax Analyzer

1. To simplify the design of a compiler
2. To increase the efficiency of a compiler
3. To enhance the portability of a compiler

* Pattern for identifier (rules)
A variable or identifier starts with either letter or underscore followed by any no. of letters or digits or underscore.
(l|_)(l|d|_)*
[a-z A-Z _][a-z A-Z 0-9 _]*

* Tokens :- <name, attributes>
name -> token name
attributes -> info of token (pointer pointed to symbol table)

for E = m * C ** 2 ;

Lexeme	Token
E	<id, 1> or id1
=	<=>
m	<id, 2> or id2
*	<*>
C	<id, 3> or id3
**	<**>
2	<2>
;	<;>

identifier.
pointer

Symbol table

1	id	.....
2	id	.....
3	id	.....
	;	...
	.	

info

Scanned with CamScanner

S.T -> symbol table
ptr -> pointer

Q Consider Some encoding of tokens as follows:-

Token	Code	Value
if	1	–
else	2	–
while	3	–
for	4	–
identifier	5	Ptr to S.T
constant	6	Ptr to S.T
<	7	1
<=	7	2
>	7	3
>=	7	4
!=	7	5
(	8	1
)	8	2
+	9	1
–	9	2
=	10	–

S.T for identifier & constant.

location counter	Type	Value
100	identifier	a
⋮	⋮	
105	constant	10
⋮	⋮	
107	identifier	i
⋮	⋮	
110	constant	2

Consider the program codes as below & generate token stream:-

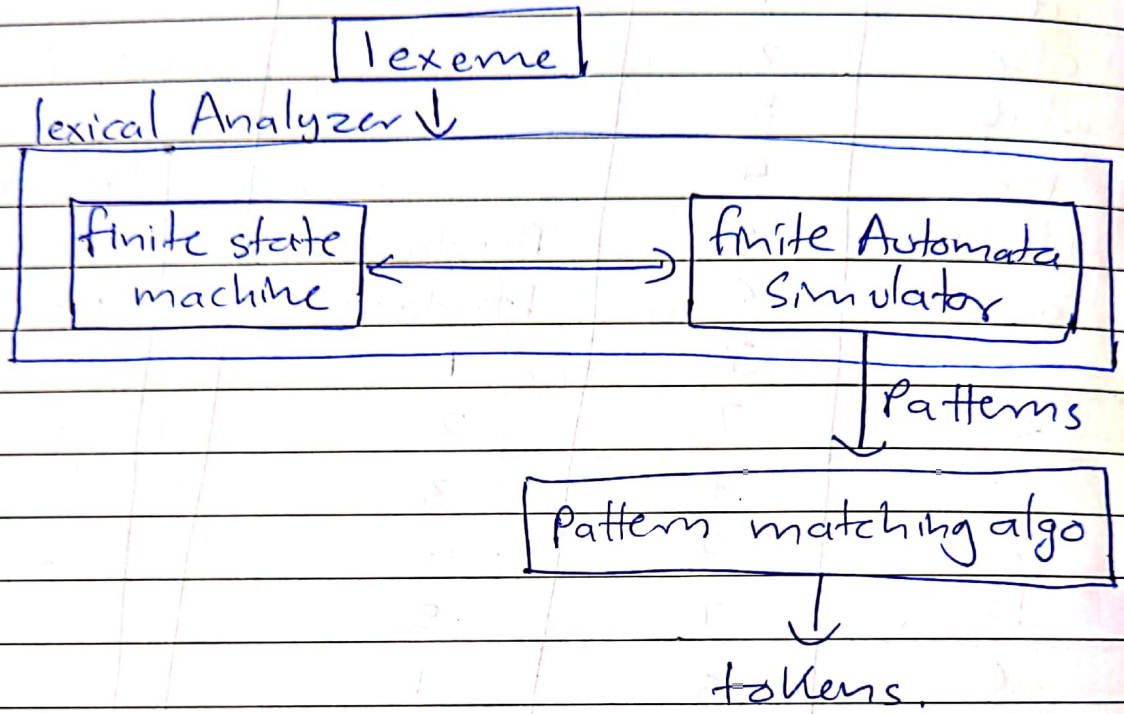
```
if (a < 10)
    i = i+2;
else
    i = i-2;
```

→ 1, (8,1), (5,100), (7,1), (6,105), (8,2), (5,107), 10, (5,107), (9,1)
(6,110), 2, (5,107), 10, (5,107), (9,2), (6,110)

for if ↑ for '(' ↑ for 'i' (code, value)

Scanned with CamScanner

Block Schematic of LA



- LA stores i/p in a buffer.
- builds R.E for tokens
- R.E → F.A is build.
- Lexeme matches pattern generated by F.A - token is recognized.

Scanned with CamScanner

$0^*1.(0^*+0^*10^*)$

H.W

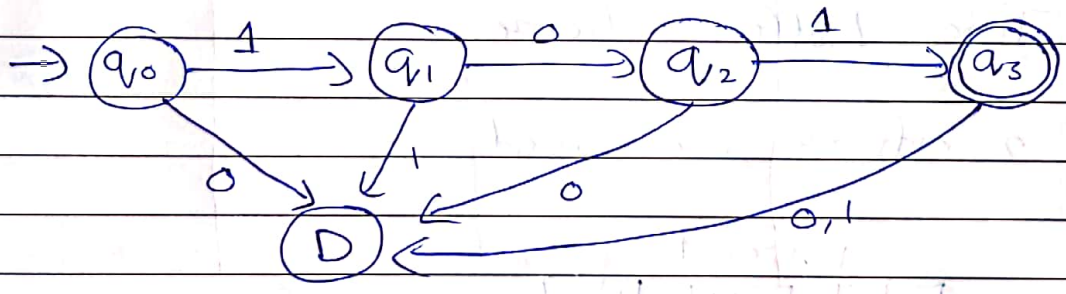
Q1) Design a finite Automata which accepts the only input 101 over the input set $\Sigma = \{0,1\}$

Q2) Design a fininte Automata which checks whether given binary number is even.

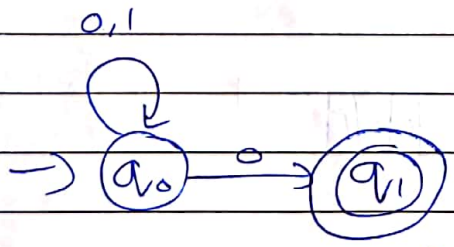
Q3) b ——— 11 ——— which accepts only those string which start with 1 & ends with 0.

Q4) ——— 11 ——— which accepts odd no. of 1's & any no. of zeroes.

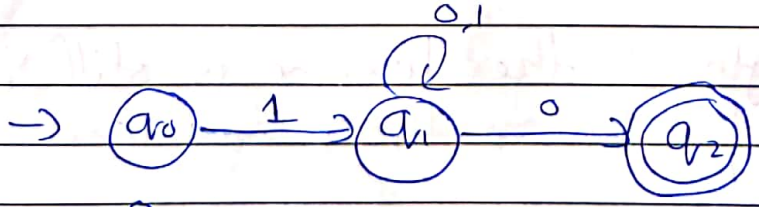
Q1) →



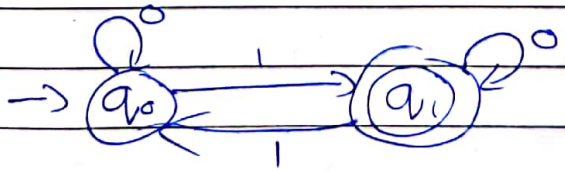
Q2) →



Q3) →

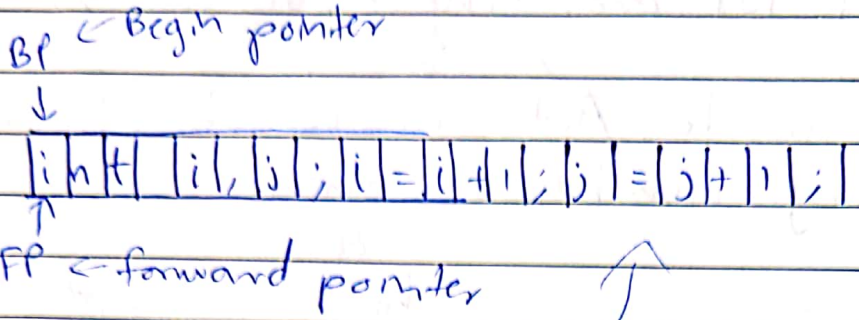


Q4) →



Scanned with CamScanner

Input Buffering

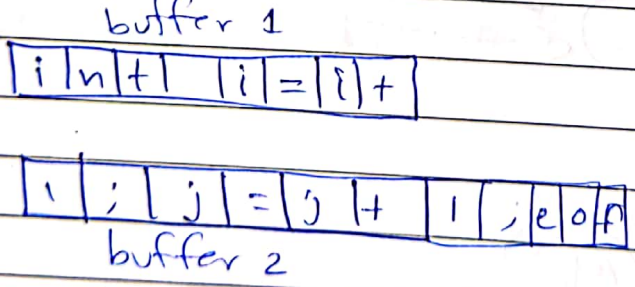


One Buffer Scheme

In this only one buffere is used, so we have to refill the buffere if the lexeme exceeds the buffer boundary.

Two Buffer scheme

2 buffers are used.



if the length of the lexeme is still bigger than both the buffers then ~~tame~~ lexeme cannot be scanned.

Scanned with CamScanner

Code for input buffering

```
if (fp == eof(buff1)) /* encounters end of first buffer */
{
    /* Refill buffer 2 */
    fp++;
}
else if (fp == eof(buff2)) /* encounters end of 2nd buffer */
{
    /* Refill buffer 1 */
    fp++;
}
else if (fp == eof(input))
    return; /* terminate scanning */
else
    fp++; /* still remaining has to be scanned */
```

☆ Lexical errors

1. Spelling errors
2. unmatched string
3. Appearance of illegal characters
4. Exceeding length of identifier

☆ Mechanism to ~~avoid~~ rectify lexical errors. (error recovery)

1. Delete a single character from the remaining input.
2. Insert a missing character into the remaining input.
3. Replace one character by other character.
4. Transpose (swap) 2 adjacent characters.

Scanned with CamScanner

(5) Delete succesive characters from the remaining input untill lexical analyzer recognizes a well formed tokens. This approach is called panic mode error recovery.

Compiler Construction Tools

1) Scanner generator - LEX, RE
2) Parser generator - YACC, CFG
3) Syntax directed traslation engines - IR
4) Automatic code generator - IR -> ML
   Template matching algorithms
5) Data flow engines - code optimization.

YACC - yet another compiler compiler.

H.W

1) Design an automata to check if given decimal no. is divisible by 3.
2) ——— " ——— checks whether if given binary no. is divisible by 3.
3) ——— " ——— accepts even no of 0's & 1's.
4) ——— " ——— checks whether given unary no. is divisible by 3.

Scanned with CamScanner

☆ Design of Lexical Analyzer

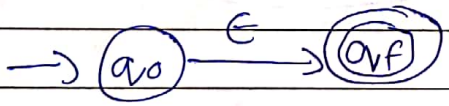
1. Representation of tokens by R.E
2. —"— R.E by Finite State Machine
3. Simulation of the behaviour of FSM

2 approaches.

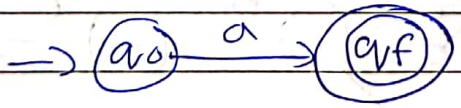
1. Pattern matching using NFA
2. Using DFA for LA.

Thompson's Rule (Construction)

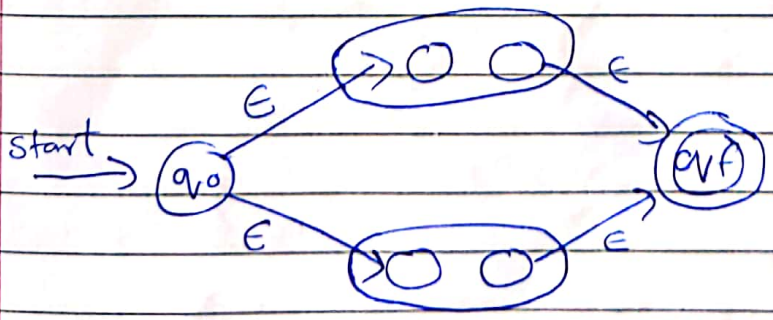
1) For $r = \epsilon$



2) $r = a$ for $\Sigma = \{a\}$ the NFA is

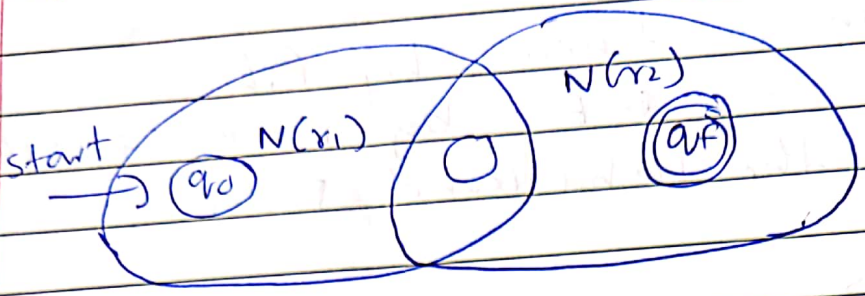


3) When $r = r_1 + r_2$ the NFA is

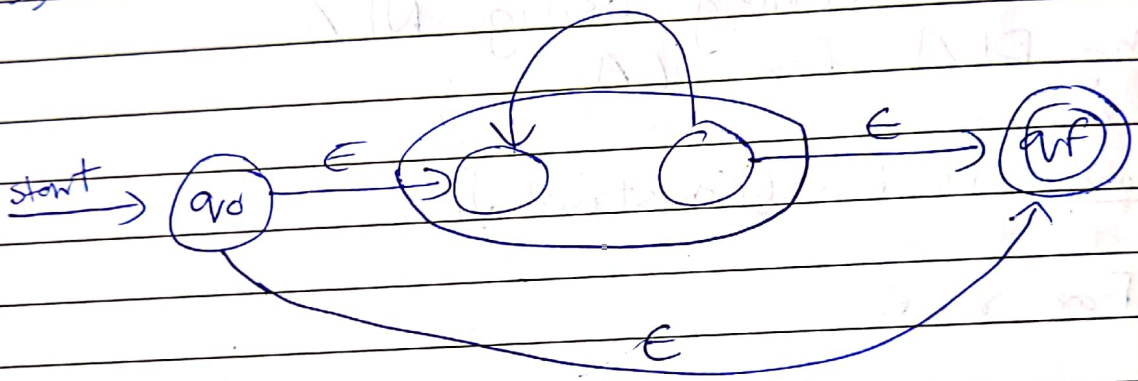


Scanned with CamScanner

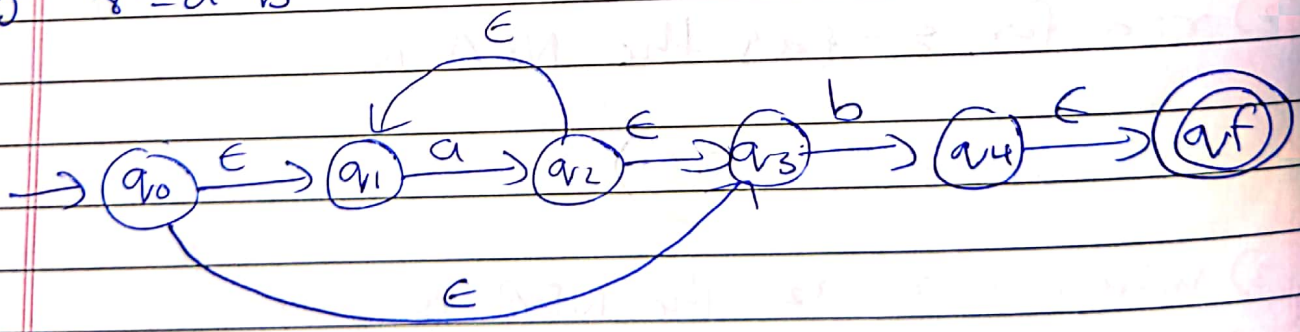
4) when $r = r_1 . r_2$



5) when $r = (r_1)^*$

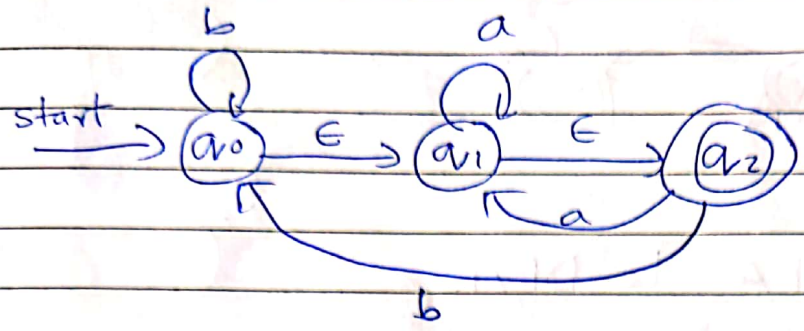


eg- $r = a^* b$



Scanned with CamScanner

Q Convert the following NFA with epsilon to equivalent DFA.



→ ε-closure $\{q_0\} = \{q_0, q_1, q_2\}$ ——— (A)
ε-closure $\{q_1\} = \{q_1, q_2\}$
ε-closure $\{q_2\} = \{q_2\}$

find the transitions in A with every i/p symbol.

$\delta'(A,a) = \epsilon\text{-closure}(\delta(A,a))$
$= \epsilon\text{-closure}(\delta(q_0,q_1,q_2),a)$
$= \epsilon\text{-closure}\{\delta(q_0,a) \cup \delta(q_1,a) \cup \delta(q_2,a)\}$
$= \epsilon\text{-closure}\{q_1\}$
$= \{q_1, q_2\}$ ——— (B)

$\delta'(A,b) = \wedge(\delta(A,b))$
$= \wedge(\delta(q_0),b)$
$= \wedge\{q_0\}$
$= \{q_0, q_1, q_2\}$
$= A$

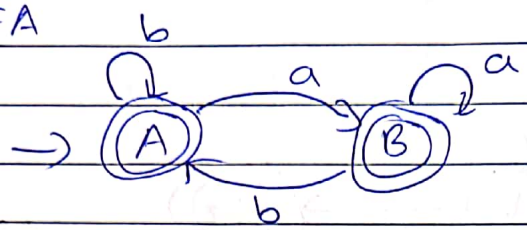
$\delta'(B,a) = \wedge(\delta(B,a))$
$= \wedge(\delta(q_1))$
$= \{q_1, q_2\} = B$

Scanned with CamScanner

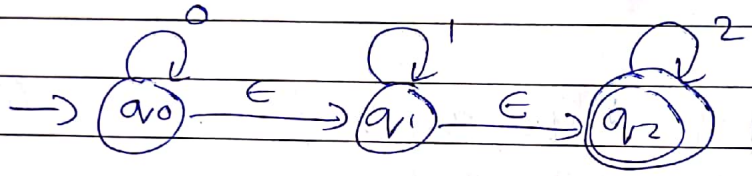
λ → closure $\quad \phi$

$\delta'(B, b) = A$

$\therefore$ DFA



Q Convert NFA to DFA.



~~$\lambda(q_0)$~~ $\lambda\{q_0\} = \{q_0, q_1, q_2\}$ —— (A)

$\lambda\{q_1\} = \{q_1, q_2\}$

$\lambda\{q_2\} = \{q_2\}$

$\delta'(A, 0) = \lambda\{\delta(A, 0)\}$

$= \lambda\{q_0\}$

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

$= A$

$\delta'(A, 1) = \lambda\{\delta(A, 1)\}$

$= B\{q_1, q_2\}$ —— (B)

$\delta'(B, \delta'(A, 2) = \lambda\{\delta(A, 2)\}$

$= \{q_2\}$ —— (C)

$\delta(B, 0) = \lambda\{\delta(B, 0)\} = \phi$

$\delta(B, 1) = \lambda\{\delta(B, 1)\} = \{q_1, q_2\} = B$

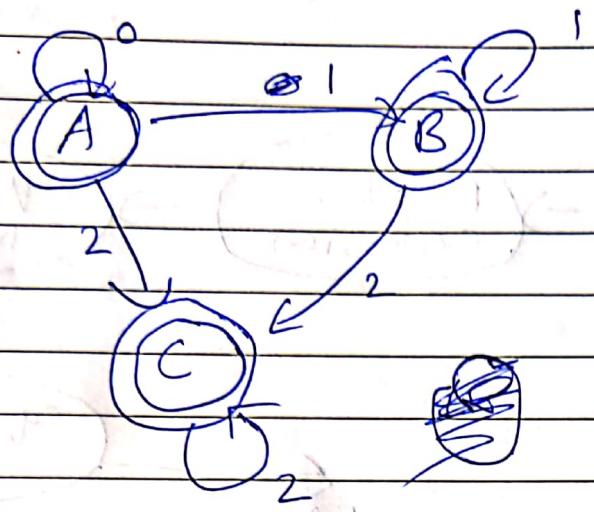
$\delta(B, 2) = \lambda\{q_2\} = \{q_2\} = C$

$\delta(C, 0) = \lambda\{\phi\} = \phi$

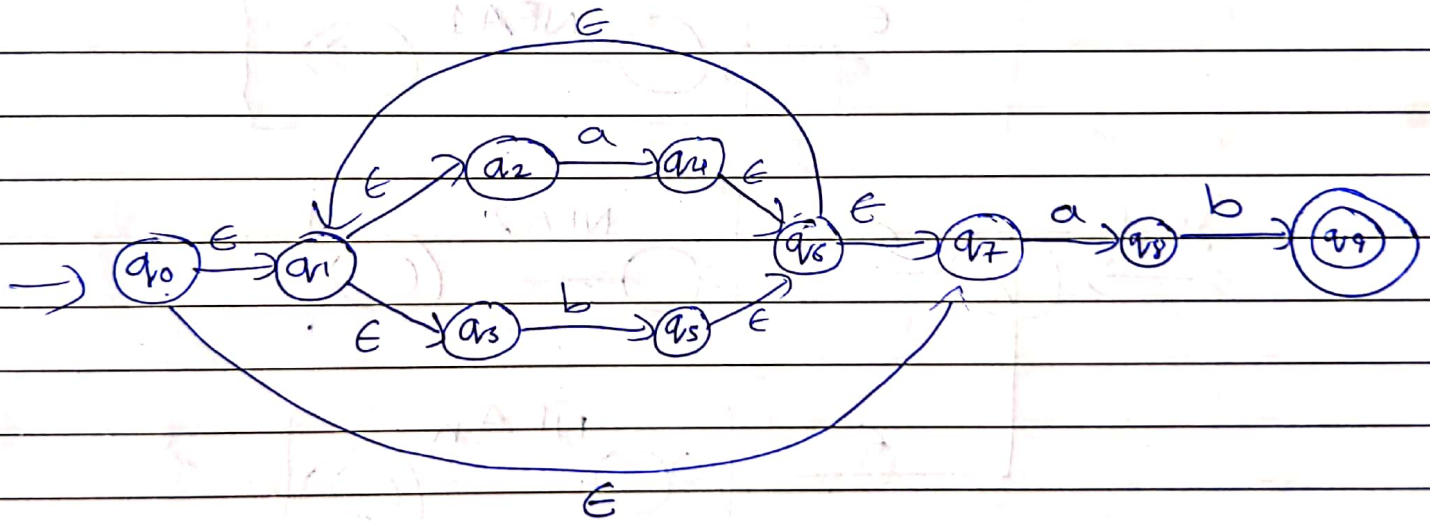
$\delta(C, 1) = \lambda\{\phi\} = \phi$

Scanned with CamScanner

$\delta(C, 2) = \Lambda\{q_{12}\} = C$



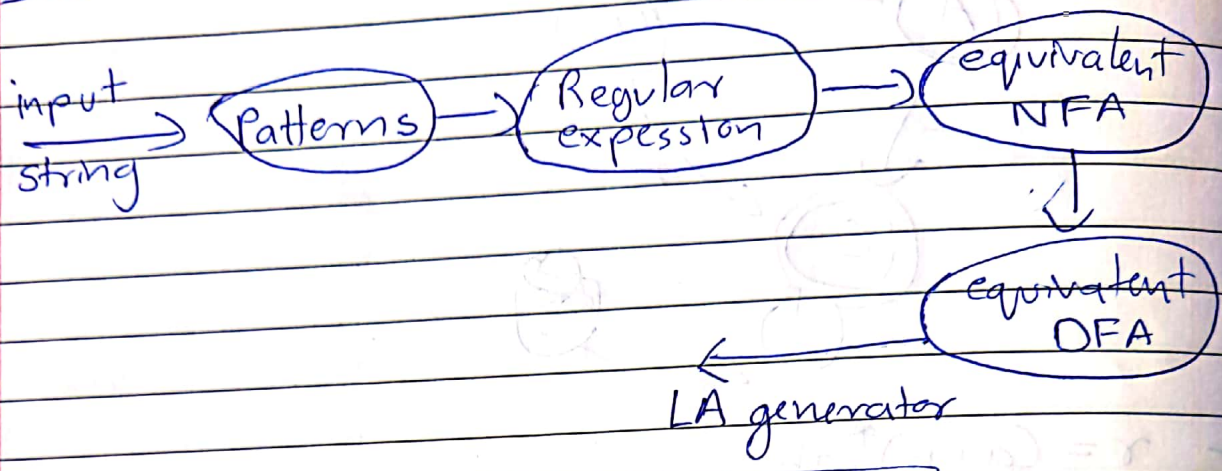
Q $r = (a+b)^* ab$

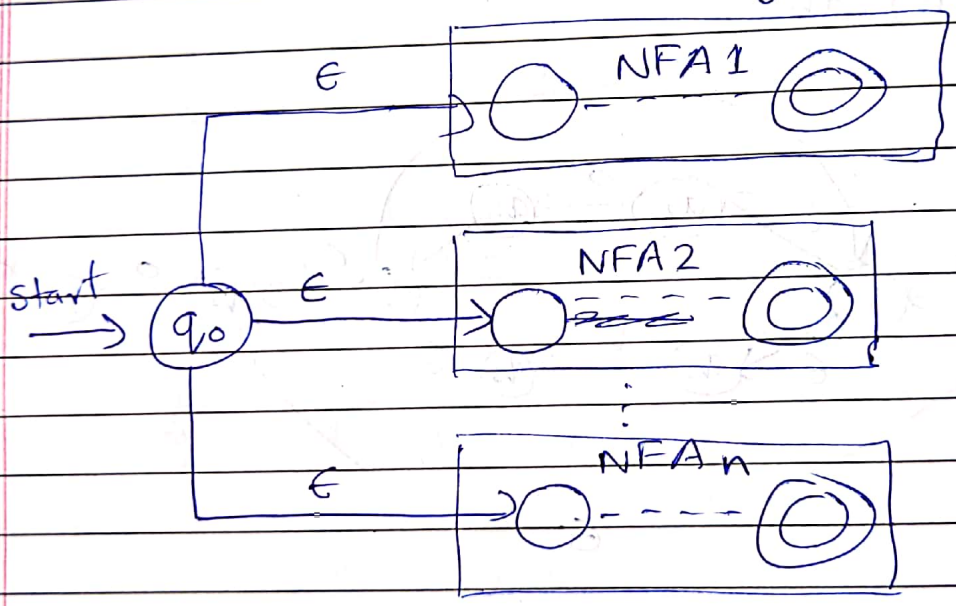


Q ~~Show~~

Scanned with CamScanner

LA Generator





- Combined NFA recognizes the longest prefix.
- Record the current i/p position & the pattern for corresponding R.E.

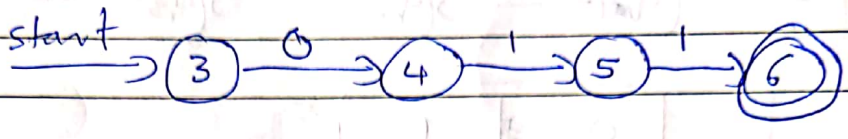
eg- 0 { } → Pattern 1
011 { } → pattern 2
0^+1^+ { } → pattern 3

String: 0011
↑

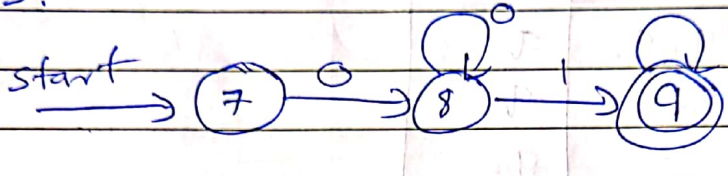
Scanned with CamScanner

pattern 1:

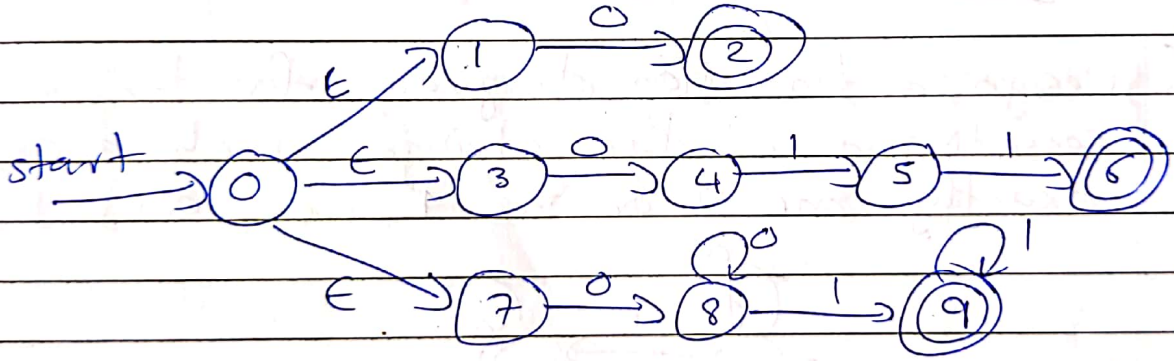
pattern 2:



pattern 3:

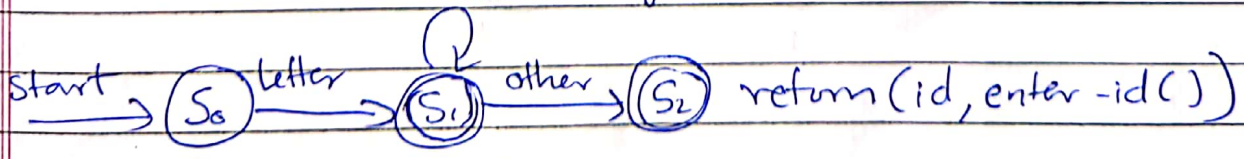


After combining;



Q Draw the transition diagram for ~~unsigned numbers~~ identifiers.

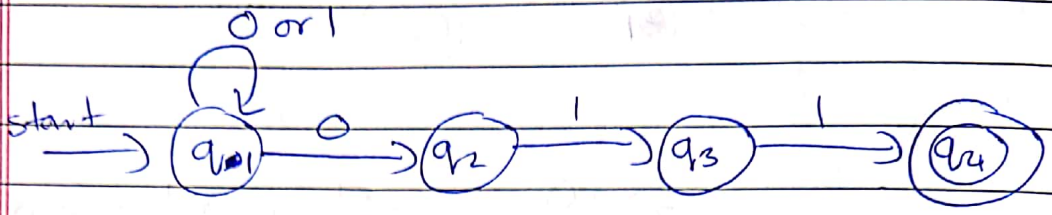
→ RE = letter (letter/digit)*



Q Design a transition diagram for the language consisting of all the strings ending with 011 {0,1}.

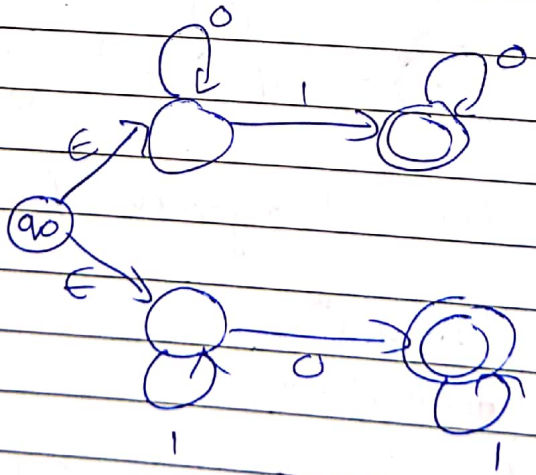
Scanned with CamScanner

→ RE = $(0|1)^*011$



	0	1
q_0	q_1, q_2	q_1
q_1	—	q_3
q_2	—	q_4
q_3	—	—
q_4		

Q Design a transition diagram for the language consisting of all the strings which accept exactly one 0 or one 1 over {0, 1}

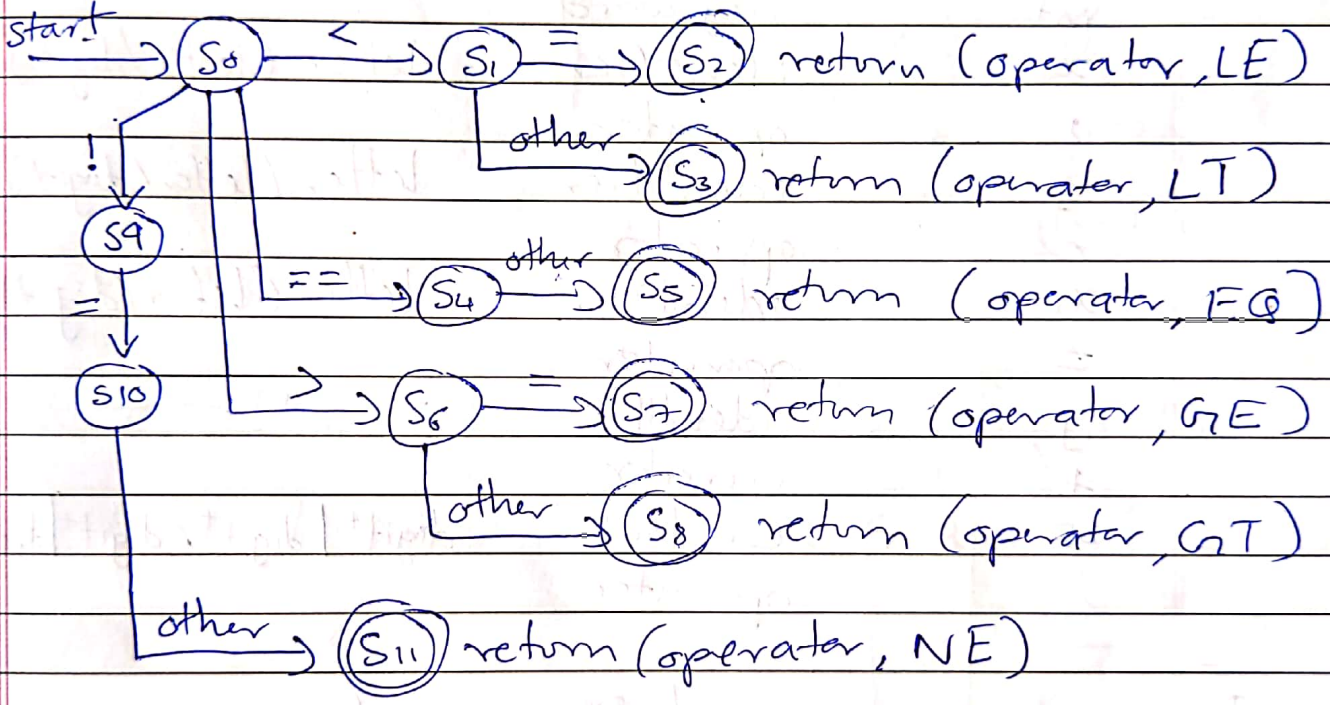


Scanned with CamScanner

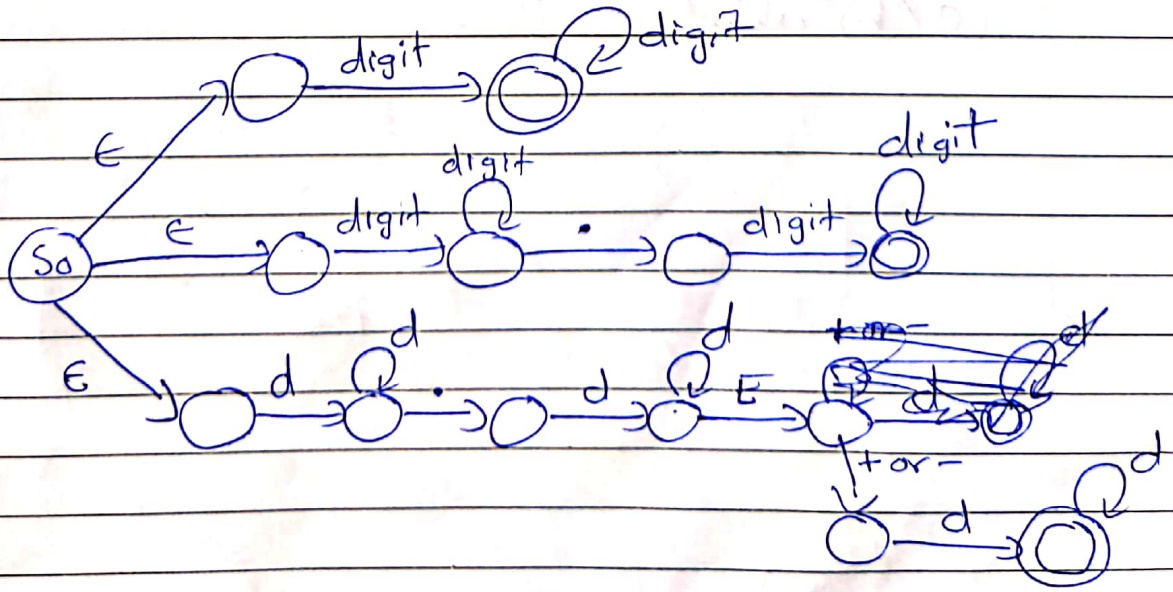
* R.E for constant

r.e = digit+ | (digit)+(.)(digit)+ | (digit)+(.)(digit)+ E(+/-)(digit+)

* Transition diagram for relational operator



* Transition diagram for constant



Scanned with CamScanner

Q Identify lexemes, token & patterns in C statement.

```
int x1, y;
x1 = y + 35;
```

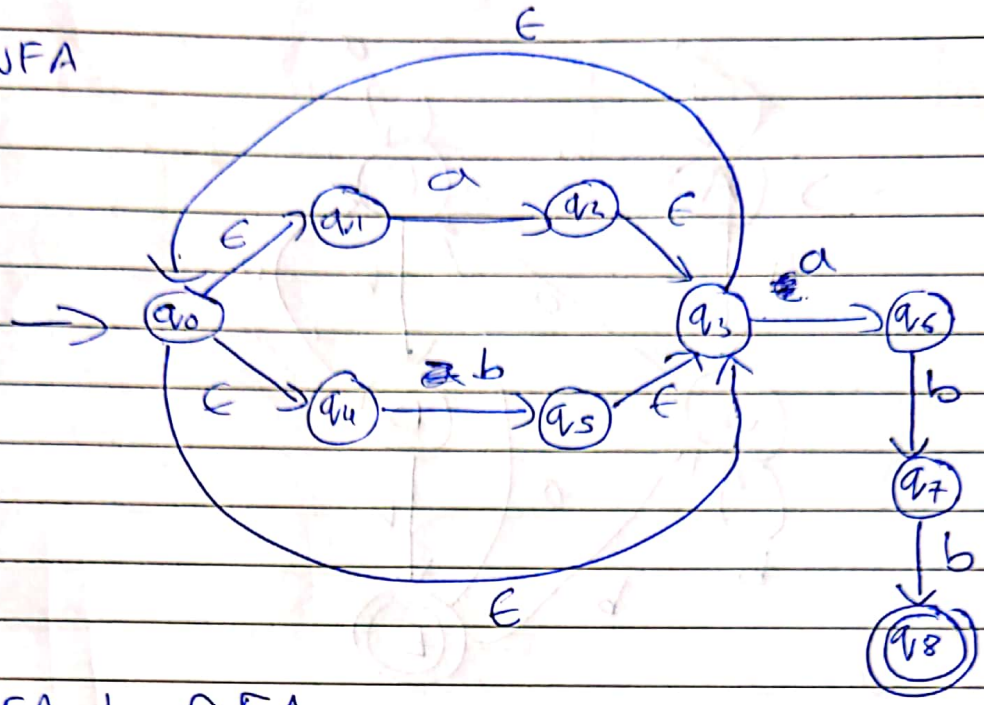
Lexemes	Token	Pattern
int	Key word	
x_1	identifier	letter (letter/digit)*
,	operator	
y	identifier	letter (letter/digit)*
;	operator	
x_1	identifier	letter (letter/digit)*
=	operator	
y	identifier	—— 11 ——
+	operator	
35	constant	digit+ | digit+ . digit+ | d.dE+/-d
;	operator	

Q Construct minimized DFA for following expressions

(a/b)* abb

Scanned with CamScanner

ϵ-NFA



ϵ-NFA to DFA

$\Lambda(q_0) = \{q_0, q_1, q_4, q_3\}$ —— (A)

$\delta'(A,a) = \Lambda\{q_2, q_6\} = \{q_2, q_3, q_0, q_1, q_4, q_6\}$ —— (B)

$\delta(A,b) = \Lambda\{q_5\} = \{q_5, q_3, q_0, q_1, q_4\}$ —— (C)

$\delta'(B,a) = \Lambda\{q_6, q_2\} = B$

$\delta'(B,b) = \Lambda\{q_5, q_7\} = \{q_5, q_3, q_0, q_1, q_4, q_7\}$ —— (D)

$\delta'(C,a) = \Lambda\{q_6, q_2\} = B$

$\delta'(C,b) = \Lambda\{q_5\} = C$

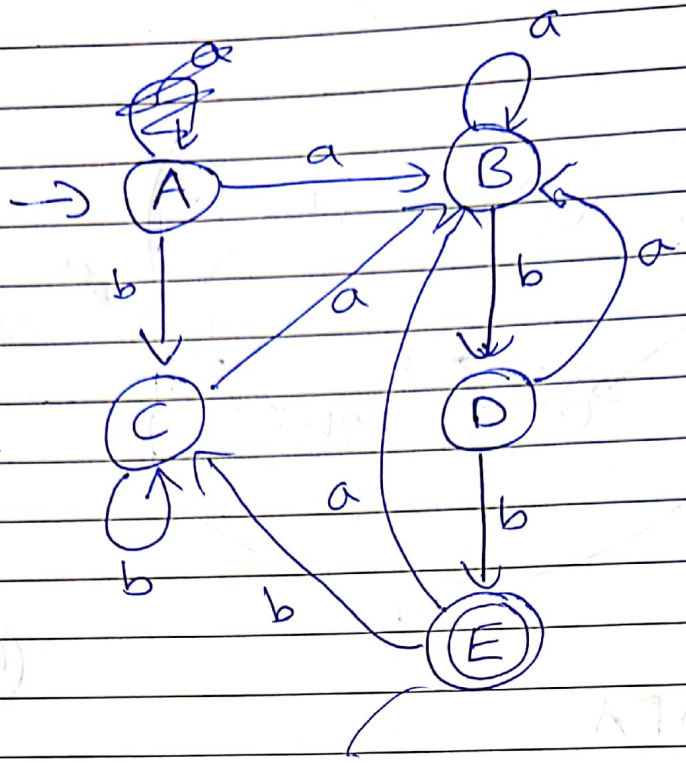
$\delta'(D,a) = \Lambda\{q_6, q_2\} = B$

$\delta'(D,b) = \Lambda\{q_5, q_8\} = \{q_5, q_3, q_0, q_1, q_4, q_8\}$ —— (E)

$\delta'(E,a) = \Lambda\{q_6, q_2\} = B$

$\delta'(E,b) = \Lambda\{q_5\} = C$

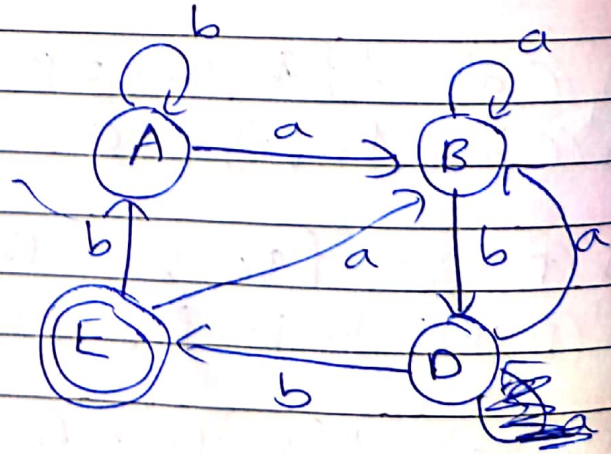
Scanned with CamScanner



	a	b
A	B	C
B	B	D
C	B	C
D	B	E
(E)	B	C

$A = \{A, C\}$

	a	b
A	B	A
B	B	D
D	B	E
(E)	B	A



Scanned with CamScanner

Q show the output of each stage of compilation to the following statement.

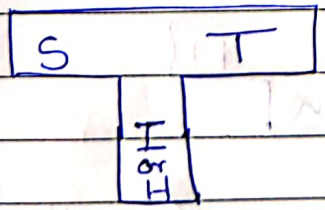
```
int t=0;
int i;
for (i=0; i<5; i++)
    t=t+5;
```

☆ Bootstrapping & Porting

Bootstrapping → is a means of developing a compiler in the target programming language which it is ~~inde~~ intended to compile.

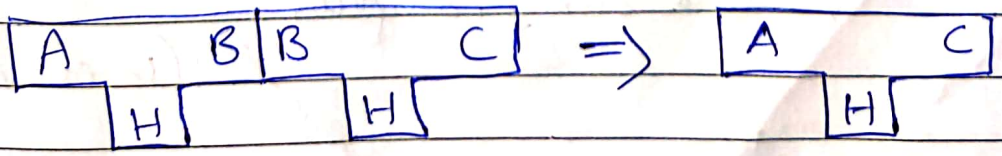
S - Source language, that it compiles
T - Target -11- that it generates code for.
H or I - Implementation language that it is written in (Host language)

T - Thomstone diagram



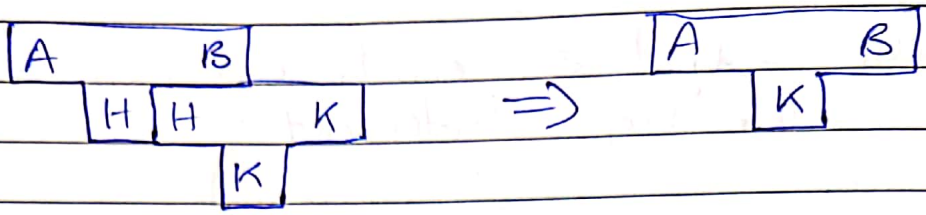
compiler written language H translates lang. S to lang. T.

Combination of T-diagram

① 

Scanned with CamScanner

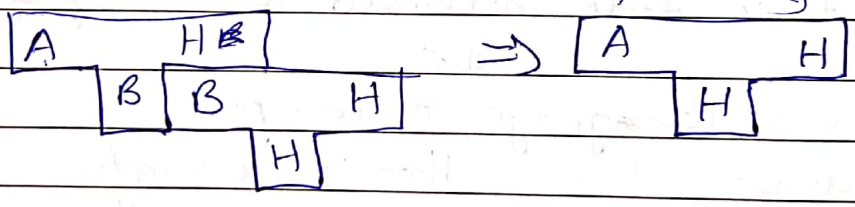
(2)



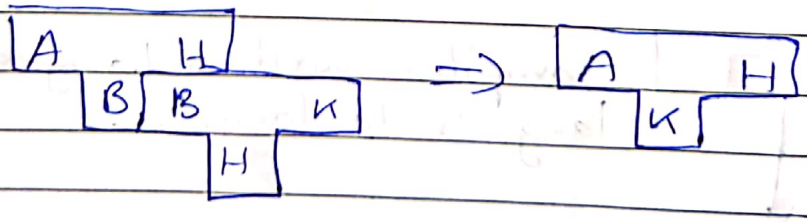
Use a compiler from machine H to machine K to translate the implementation lang. of another compiler from H to K.

Consider 2 scenarios

1st scenario :- Use an existing compiler for lang B on machine H to translate a compiler from lang A to H written in B.



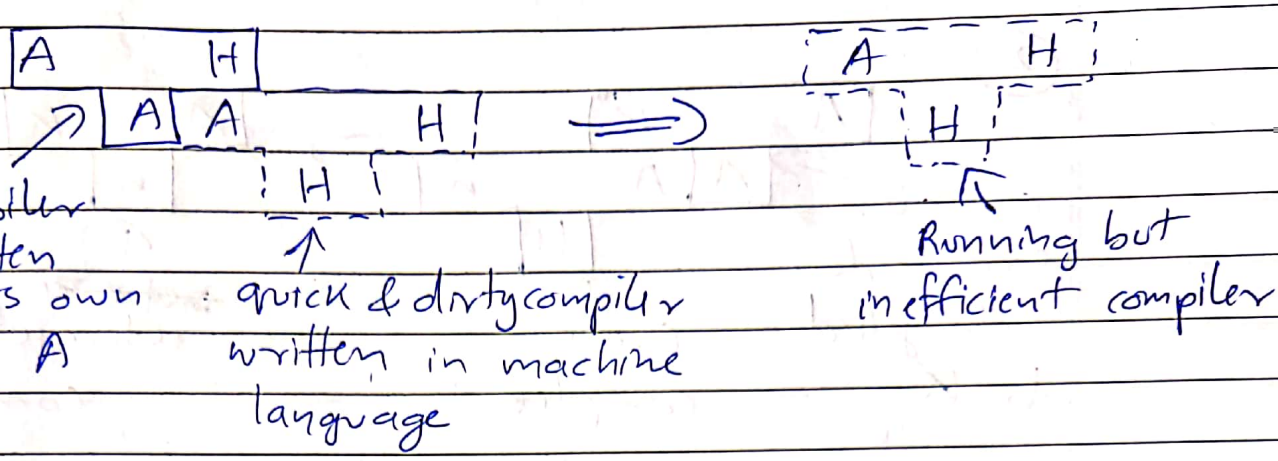
2nd scenario :- Compiler of lang. B generates code for (& runs on) a different machine, which results in a cross compiler for A.



Quick & dirty compiler ⇒ written in assembly lang. ⇒ (inefficient)

Scanned with CamScanner

step (1)

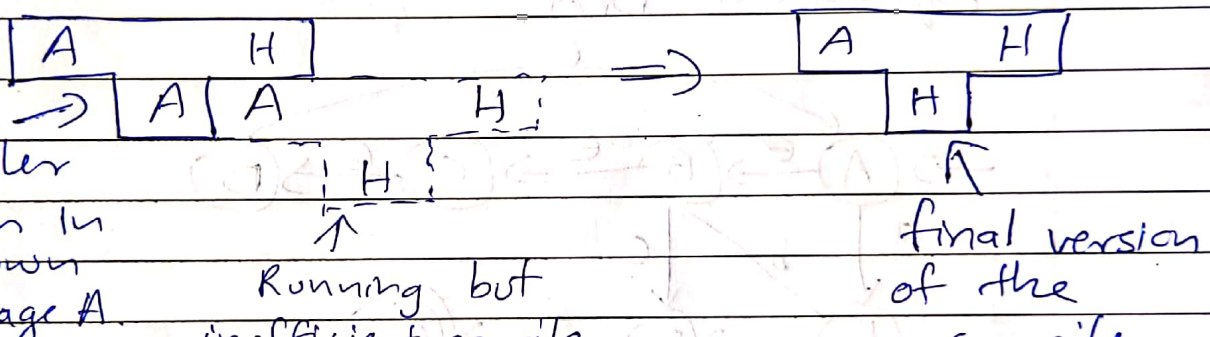


Compiler written in its own lang A

quick & dirty compiler written in machine language

Running but inefficient compiler

step (2)



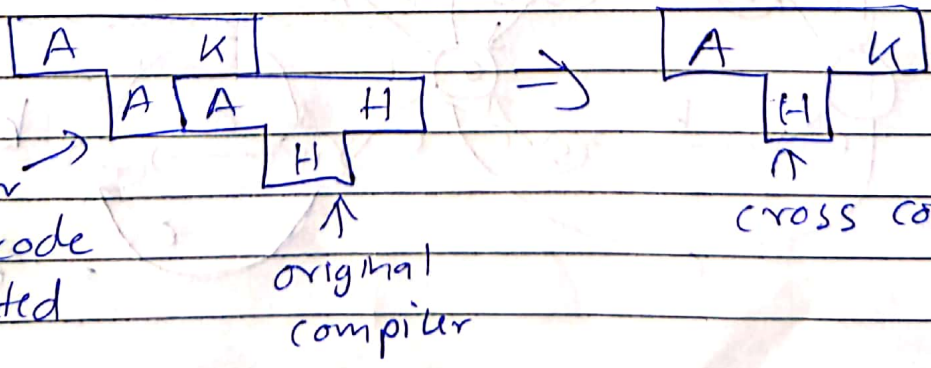
Compiler written in its own language A.

Running but inefficient compiler (from first step)

final version of the compiler

☆ Porting ⇒ only requires that the back end of the source code be rewritten to generate code for new machine.

(1)



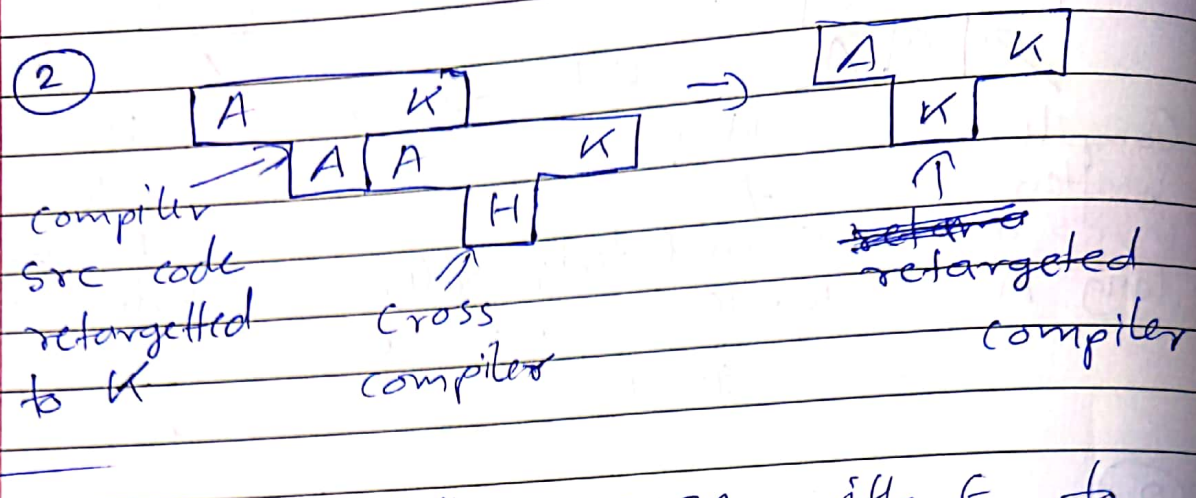
Compiler source code retargeted to K

original compiler

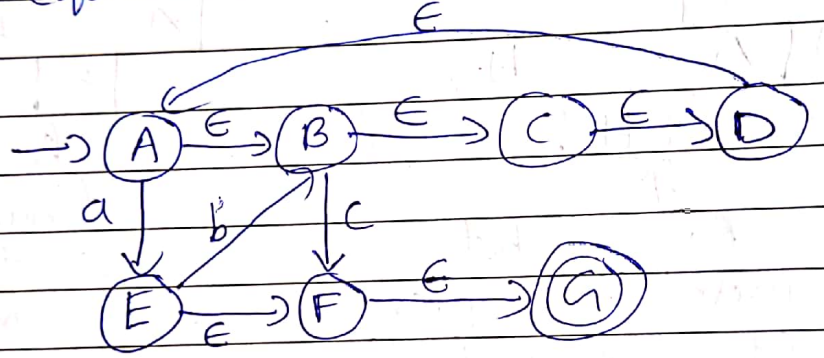
cross compiler

Scanned with CamScanner

(2)

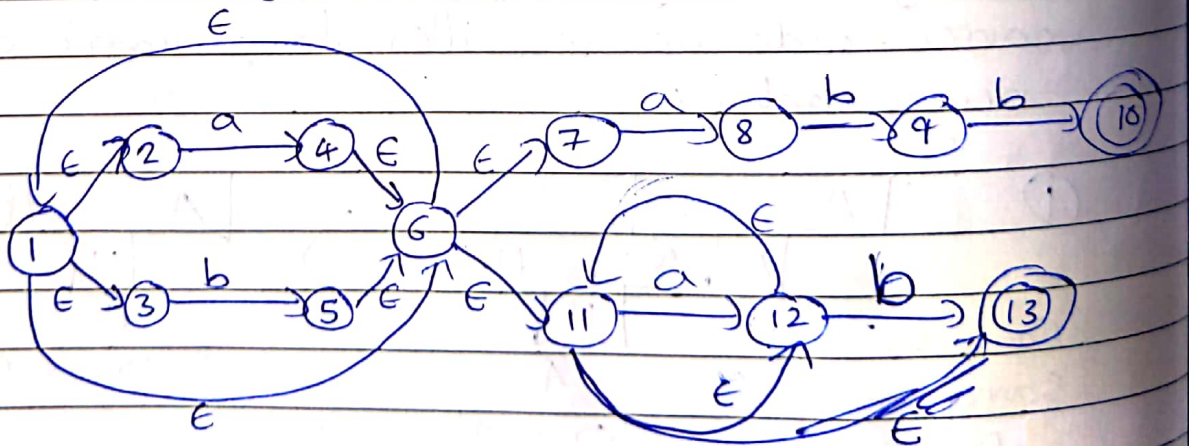


Q1) Convert the following NFA with ∈ to equivalent DFA.



Q2) Construct DFA for foll. ~~RE = (a+b)*(abb+a*b)~~

RE = (a+b)*(abb + a*b)



Scanned with CamScanner

(Q1) -> $\Lambda(A) = \{A\ B\ C\ D\}$ —— (1)

$\delta'(1,a) = \Lambda\{E\} = \{E\ F\ G\}$ —— (2)

$\delta'(1,b) = \Lambda\{\phi\} = \phi$

$\delta'(1,c) = \Lambda\{F\} = \{F\ G\}$ —— (3)

$\delta'(2,a) = \Lambda\{\phi\}$

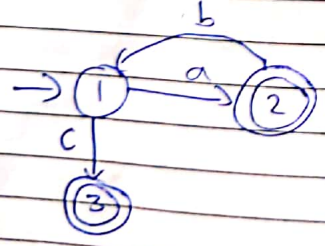
$\delta'(2,b) = \Lambda\{B\} = \{A\ B\ C\ D\} = 1$

$\delta'(2,c) = \{\phi\}$

$\delta'(3,a) = \Lambda\{\phi\}$

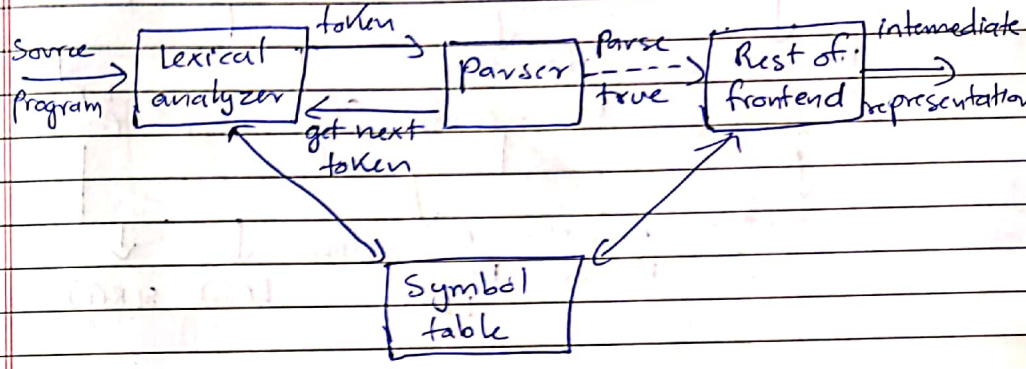
$\delta'(3,b) = \Lambda\{\phi\}$

$\delta'(3,c) = \Lambda\{\phi\}$



Syntax Analysis

Role of the Parser



Basic Issues in Parsing

i) Specification of syntax

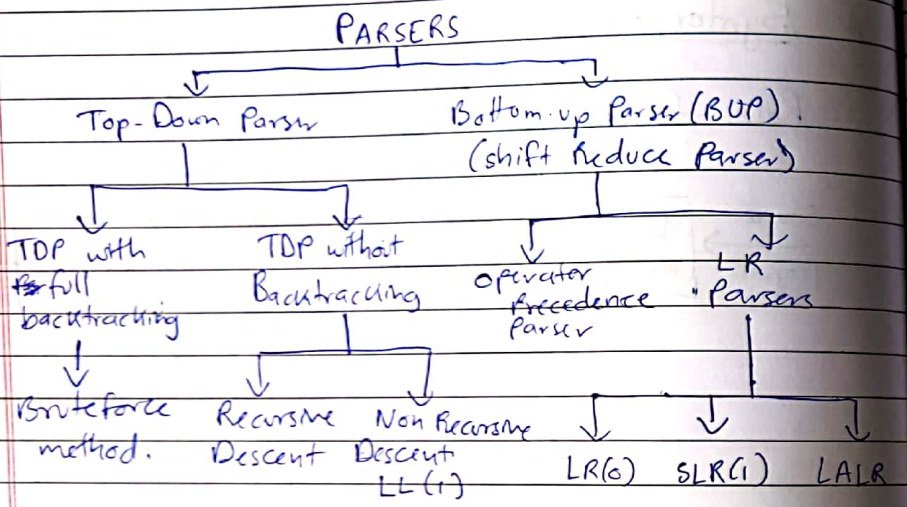
ii) Representation of input after passing.

Syntax → precise & unambiguous
→ in detail
→ compute.

parsing algo.

Scanned with CamScanner

LR -> Left to Right
SLR -> simple left Right
LALR -> look Ahead Left Right



Top-Down Parsing (LL(K))

Leftmost derivation (LMD)
LL(K)
↳ L -> left to right
↳ L -> leftmost derivation
↳ K -> K stands for no. of lock ahead tokens

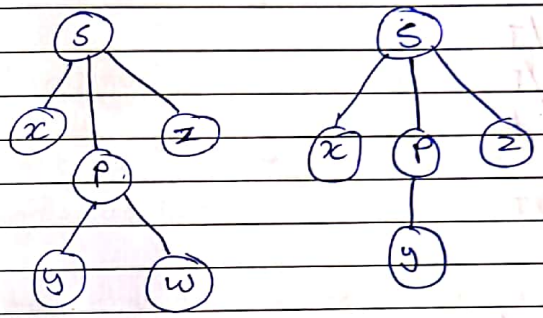
Normally use 1 -> scan one i/p at a time.

Problems with top-down Parsing

1) Backtracking
2) Left recursion
3) Left factioring
4) Ambiguity.

(1) Backtracking

eg- $S \to xPz$ i/p string :- xyz
$P \to yw/y$



↑ Backtracking ~~increases~~ overload.

(2) Left Recursion

$A^+ \to A\alpha$
$\alpha \to$ i/p string

- top-down parser can enter in infinite loop.
- expansion of A causes further expansion of A only.
i.e $A, A\alpha, A\alpha\alpha, \cdots$ i/p ptr. will not be sophesticated

$$A \to A\alpha / \beta$$
$$\Downarrow$$
$$A \to \beta A'$$
$$A' \to \alpha A' / \epsilon$$

$A \to A\alpha_1 / A\alpha_2 / A\alpha_3 / A\alpha_4 // A\alpha_m / \beta_1 / \beta_2 / \cdots / \beta_m$
where no. β_i begins with one A

Scanned with CamScanner

$A \to \beta_1 A' / \beta_2 A' / \dots / \beta_n A'$

$A' \to \alpha_1 A' / \alpha_2 A' / \dots / \alpha_m A' / \epsilon$

eg-

1) $E \to E+T / T$
2) $T \to T*F / F$
3) $F \to (E) / id$

① $E \to \overset{A}{E} \overset{\alpha}{+} \overset{\beta}{T} / T$ ×

$E \to TE'$ ✓
$E' \to +TE' / \epsilon$ ✓

② $T \to T*F / F$ ×

$T \to FT'$ ✓
$T' \to *FT' / \epsilon$ ✓

③ $F \to (E) / id$ ✓

(3) ~~F~~.

Left Factoring

$A \to \alpha\beta_1 / \alpha\beta_2$

$\Downarrow$

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

eg $S \to \overset{\alpha}{iEt}\ S / \overset{\alpha}{iEt}\ S \overset{\beta_1}{eS} / a$
$E \to b$

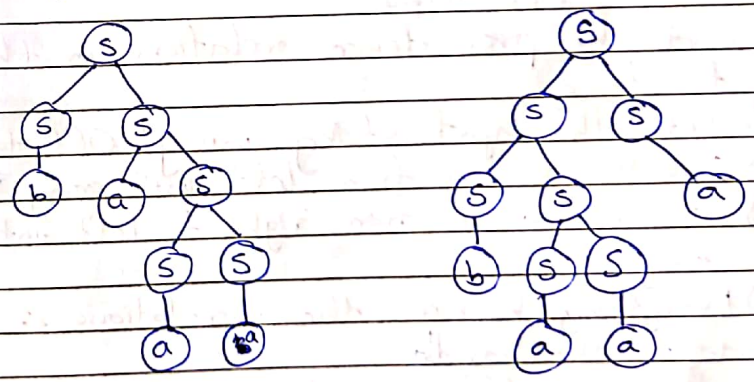
→ $S \to iEtSS' / a$
$S' \to eS / \epsilon$

(4) Ambiguity

left-associative operator $(+, -, *, /)$ => induce left recursi
right-associative " (exponential) => induce right recursion

Q Show that the grammar is ambiguous
$S \to SS / a / b$
string - baaa

→



Operator Precidence Parser

Operator precedence grammar → $N \to \underset{RHS}{\epsilon X}$

↓

RHS should not have ~~abst~~ ϵ & 2 non terminals

Scanned with CamScanner

Precedence relations <· ≐ ·>

p <· q p is given more precedence than q.
p ≐ q p has same " as q
p ·> q p takes precedence over q

~~Compute~~

Steps for OP parsing (OPP):-

1) Compute Leading & Trailing
2) " precedence relations
3) fill the OPP table
4) Insert the precedence relations in the input string
5) process the input string using OPP table
   i) Scan the string from left untill seeing ·>
   ii) Scan backwards from right to left until seeing <·
   iii) Everything between the 2 relations <· and ·> forms the handle
6) Compute precedence function graph
7) Find numerical value for f and g.

I) Computation of LEADING() and TRAILING()

LEADING(A)

'a' is in leading(A) [A → γ a δ] ; δ - either empty or a single NT

If 'a' is in leading(B) [A → Bα] then

a is in leading of A.

Trailing(A)

'a' is in trailing(A) [A → δ a δ] ; δ - either empty or single NT

If 'a' is in trailing(B) [A → Bα] then

a is in trailing of A.

γ a δ

eg① E → E+T
eg- E → T
T → T*F
T → F
F → (E)
F → id

.cu: E → T (rule 2)

→ leading(E) = {+, *, (, id}
leading(T) = {*, (, id}
leading(F) = {(, id}
trailing(E) = {+, *,), id}
trailing(T) = {*,), id}
trailing(F) = {), id}.

Scanned with CamScanner

Eg- S → L = R
S → R
L → * R
L → id
R → L

→ Leading(S) = {=, *, id}
Leading(L) = {*, id}
Leading(R) = {*, id}

trailing(S) = {=, *, id}
trailing(L) = {*, id}
trailing(R) = {*, id}

* Precedence Relation

1) $ <· leading(S)
   Trailing(S) ·> $
2) $T_1\ T_2 \xRightarrow{\text{set}} T_1 \doteq T_2$
3) $T_1\ NT\ T_2 \xRightarrow{\text{set}} T_1 \doteq T_2$
4) $T\ NT \xRightarrow{\text{set}} T <\cdot \text{leading}(NT)$
5) $NT\ T \xRightarrow{\text{set}} \text{Trailing}(NT) \cdot> T$

T → terminal

For eg①

LHS \ RHS	+	*	(	)	id	$
+	·>	<·	<·	·>	<·	·>
*	·>	·>	<·	·>	<·	·>
(	<·	<·	<·	≐	<·	
)	·>	·>		·>		·>
id	·>	·>		·>		·>
$	<·	<·	<·		<·	

E → E + T
(NT: E, T; T: +)
∴ Trail(E) ·> +
∴ * ·> +
+ ·> +
) ·> +
id ·> +

OP Parsing Algorithm

if a <· b or a ≐ b ⇒ push b onto the stack, advance i/p ptr.

else if a ·> b ⇒ pop stack until the top stack terminal is related by <· to the terminal most recently popped.

For eg①
string: id + id * id

Stack	Input String	Action
$	id + id * id $	$ <· id ⇒ push
$ id	+ id * id $	·> ⇒ pop
$	+ id * id $	<· ⇒ push
$ +	id * id $	<· ⇒ push
$ + id	* id $	·> ⇒ pop
$ +	* id $	<· ⇒ push
$ + *	id $	<· ⇒ push
$ + * id	$	·> ⇒ pop
$ + *	$	·> ⇒ pop
$ +	$	·> ⇒ pop
$	$	ACCEPT

* Precedence Functions

f(a) < g(b) whenever a <· b
f(a) = g(b) " a ≐ b
f(a) > g(b) " a ·> b

Scanned with CamScanner

step 1) Create Symbols fa and gb for each terminal a

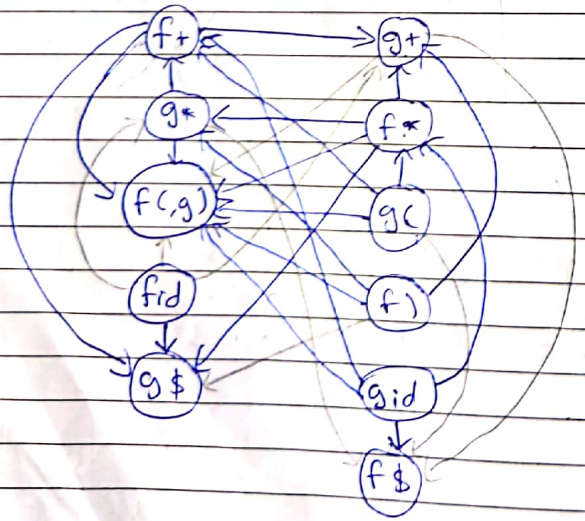
step 2) Partition the symbols into groups $a \doteq b$ fa and gb in same groups.

step 3) Create a directed graph
if $a <\cdot b$ place edge from gb to fa
if $a \cdot> b$ place edge from fa to gb.

step 4) if graph has cycle ⇒ no precedence function exist.
if no cycle ⇒ take the longest path of f(a), g(b).

Step①- f+, f*, f(, f), fid, f$
g+, g*, g(, g), gid, g$

(f(,g))



	+	*	id	(	)	$
f	2	4	4	0	4	0
g	1	3	5	5	0	0

(Q2) S → L=R
S → R
L → *R
L → id
R → L

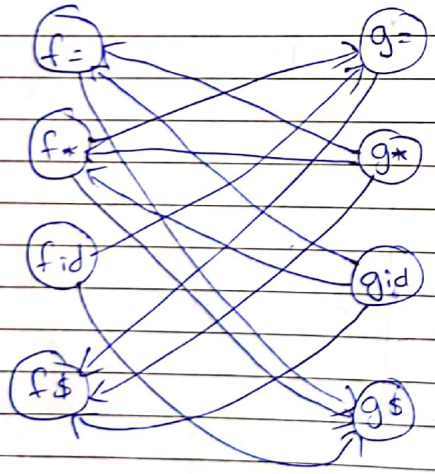
→ Leading (S) = {=, *, id}
Leading (R) = {*, id}
Leading (L) = {*, id}
Trailing (S) = {=, *, id}
Trailing (R) = {*, id}
Trailing (L) = {*, id}

	=	*	id	$
=		<·	<·	·>
*	·>	<·	<·	·>
id	·>			·>
$	<·	<·	<·	

String: * id = id $

Scanned with CamScanner

stack	input	Action
$	* id = id $	<·
$*	id = id $	<·
$* id	= id $	·>
$*	= id $	·>
$	= id $	<·
$ =	id $	<·
$ = id	$	·>
$ =	$	·>
$	$	ACCEPT



	=	*	id	$
f	1	2	2	0
g	1	3	3	0

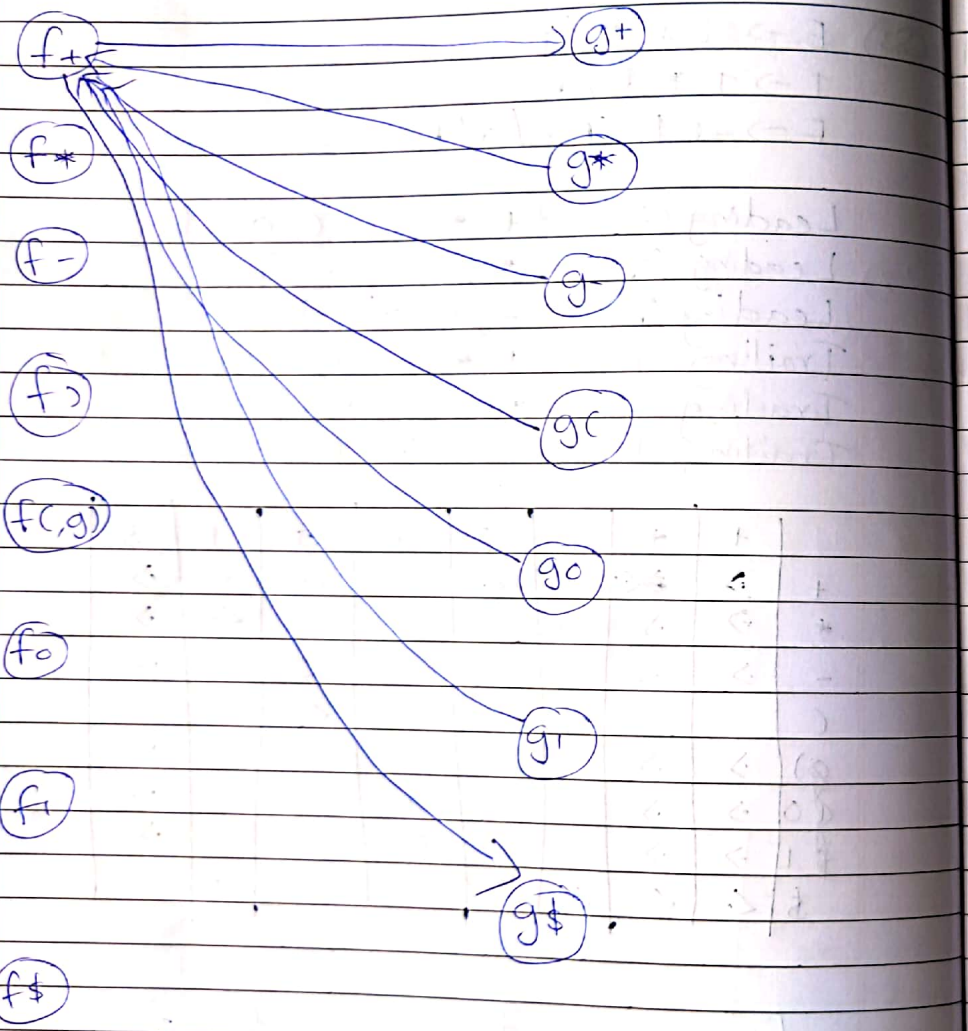
(Q3) E → E + T / T
T → T * F / F
F → - F / (F) / 0 / 1

Leading (E) = { +, *, -, (, 0, 1 }
Leading (T) = { *, -, (, 0, 1 }
Leading (F) = { -, (, 0, 1 }
Trailing (E) = { +, *, -,), 0, 1 }
Trailing (T) = { *, -,), 0, 1 }
Trailing (F) = { -, (, 0, 1 }

	+	*	-	(	)	0	1	$
+	·>	<·>	<·	<·		<·	<·	·>
*	·>	·>	<·	<·		<·	<·	·>
-	·>	·>	<·	<·	·>	<·	<·	·>
(			<·	<·	≐	<·	<·	
)	·>	·>			·>			·>
0	·>	·>			·>			·>
1	·>	·>			·>			·>
$	<·	<·	<·	<·		<·	<·	

f+, f*, f-, f), f0, f1, f$
g+, g*, g-, g(, g0, g1, g$, f(,g)

Scanned with CamScanner



☆ Shift Reduce Parser

S → aABe
A → Abc/b
B → d
Sentence abbcde reduced to S
abb cde
aAbcde
aAde
aABe
S

i.e $S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} aAbcde \underset{rm}{\Rightarrow} abbcde$

Handles

- Substring that matches RHS of a production to NT on LHS represents one step along reverse of rightmost derivation.

Handle

E → E+E
E → E*E I/P $id_1 + id_2 * id_3$
E → (E)
E → id

Scanned with CamScanner

Right-Sentential form	Handle	Reducing Production
$id_1 + id_2 * id_3$	id_1	$E \to id$
$E + id_2 * id_3$	id_2	$E \to id$
$E + E * id_3$	id_3	$E \to id$
$E + E * E$	$E * E$	$E \to E * E$
$E + E$	$E + E$	$E \to E + E$
E		

Stack Implementation of Shift Reduce Parsing

Stack	input buffer
$	w$

4 actions of S-R parser

1) shift
2) Reduce
3) Accept
4) Error

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

$w = id_1 + id_2 * id_3$

stack	i/p buffer	Parsing Action
$	$id_1 + id_2 * id_3$ $	shift
$id_1	$+id_2 * id_3$ $	Reduce by $E \to id$
$E	$+id_2 * id_3$ $	shift
$E+	$id_2 * id_3$ $	shift
$E+id_2	$* id_3$ $	reduce by $E \to id$
$E+E	$* id_3$ $	shift
$E+E*	id_3 $	shift
$E+E*id_3	$	Reduce by $E \to id$
$E+E*E	$	Reduce by $E \to E*E$
$E+E	$	Reduce by $E \to E+E$
$E	$	Accept.

For above table :-

rule 1 - if the incoming operator has more priority then in-stack operator, then perform shift.

rule 2 - if in-stack operator has same or less priority then the priority of incoming operator, then perform reduce

Q Consider the following grammar

$S \to TL;$
$T \to int/float$
$L \to L, id/id$
input string : int id, id;

Scanned with CamScanner

Stack	i/p buffer	Parsing Action
$	int id, id; $	shift
$int	id, id; $	reduce T → int
$T	id, id; $	reduce L → id
$TL	, id; $	shift
$TL,	id; $	shift
$TL, id	; $	reduce L → L, id
$TL	; $	~~reduce S →~~ shift
$TL;	$	reduce S → TL;
$S	$	ACCEPT

Q. S → CC
C → cC / d
i/p string w = cdd

Stack	i/p buffer	Parsing Action
$	cdd $	shift
$c	dd $	shift
$cd	d $	reduce C → d ~~shift~~
$cC	d $	reduce C → cC
$C	d $	shift
$Cd	$	reduce C → d
$CC	$	reduce S → CC
$S	$	ACCEPT

Q. S → (L) / a
L → L, S / S
i/p string: (a, (a, a))

Stack	i/p buffer	Parsing Action
$	(a, (a, a)) $	shift
$(	a, (a, a)) $	shift
$(a	, (a, a)) $	~~shift~~ reduce S → a
~~$(a,~~	~~(a, a)) $~~	~~shift~~
~~$(a,~~		
$(S	, (a, a)) $	~~S~~ L → S
$(L	, (a, a)) $	shift
$(L,	(a, a)) $	shift
$(L,(	a, a)) $	S → a
$(L,(S	, a)) $	L → S
$(L,(L	, a)) $	shift
$(L,(L,	a)) $	shift
$(L,(L,a	)) $	S → a
$(L,(L,S	)) $	~~$~~ L → ~~$~~ L, S
~~$(L,(L,L~~	~~)) $~~	~~shift~~
~~$(L,(L,L)~~		
$(L,(L	)) $	shift
$(L,(L)	) $	S → (L)
$(L,S	) $	L → L, S ~~shift~~
$(L ~~$~~	) $	shift
$(L)	$	~~$~~ S → (L)
$ ~~$~~ S	$	ACCEPT

Scanned with CamScanner

classmate
Date
Page

Conflicts During shift-Reduce Parsing

1) shift - Reduce Conflicts
2) Reduce - Reduce conflict

Scanned with CamScanner