LALR (1) parser

(Lookahead LR parser).

Algorithm: An easy, but space consuming LALR table construction.

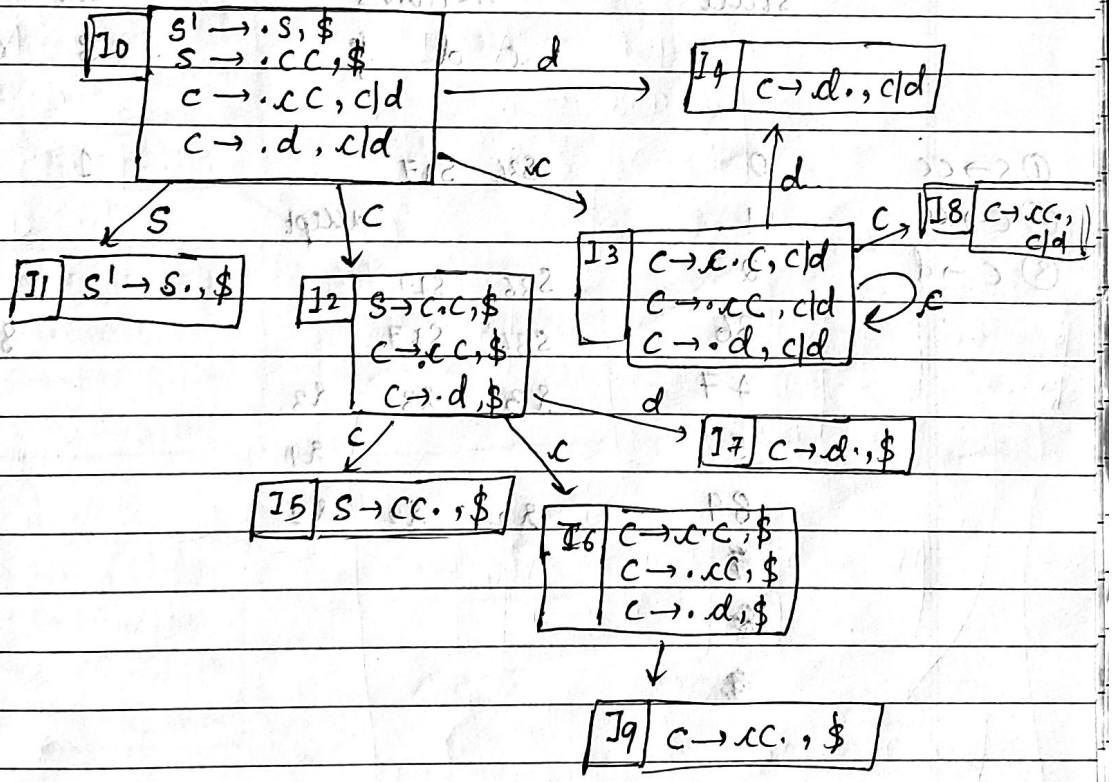
i) $C = \{I_0, I_1, \ldots, I_n\}$

ii) For each core present in LR(1) set of items, $J = I_1 \cap I_2 \cap \ldots I_n$, merge the set of item whose first right sentential form is same.

$S \rightarrow CC$

$C \rightarrow cC \mid d$

i) DO LR(1) ⇒ CLR



Find two states whose first right sentential form are same

like

I3 & I6 , I8 & I9 , I4 & I7.

I3 & I6 ⇒ Merge ⇒

I36
C → c · C, c|d|$
C → · cC, c|d,$
C → · d , c|d|$

I4 & I7 ⇒ Merge ⇒

I47
C → d · , c|d|$

I8 & I9 ⇒ Merge ⇒

I89
C → cC · , c|d|$

parse table.

state	ACTIONS			GOTO	
	c	d	$	S	C
0	S36	S47		1	2
1			accept		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

① S → CC
② C → cC
③ C → d

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

⇒ no common states, so, it is same as CLR

I0
$E' \to \cdot E, \$$
$E \to \cdot E+E, \$/+/*$
$E \to \cdot E*E, \$/+/*$
$E \to \cdot id, \$/+/*$

⇒ can be written as

I0
$E' \to \cdot E, \$$
$E \to \cdot E+E, \$/+/*$
$E \to \cdot E*E, \$/+/*$
$E \to \cdot id, \$/+/*$

E → **I1**

id → **I2**

I1
$E' \to E \cdot, \$$
$E \to E \cdot +E, \$$
$E \to E \cdot *E, \$$
$E \to E \cdot +E, +$
$E \to E \cdot *E, +$
$E \to E \cdot +E, *$
$E \to E \cdot *E, *$

I2
$E \to id \cdot, \$$
$E \to id \cdot, +$
$E \to id \cdot, *$

⇒ **I2** $E \to id \cdot, \$/+/*$

\+ → **I3**

* → **I4**

I3
$E \to E+ \cdot E, \$/+/*$
$E \to \cdot E+E, \$/+/*$
$E \to \cdot E*E, \$/+/*$
$E \to \cdot id, \$/+/*$

I4
$E \to E* \cdot E, \$/+/*$
$E \to \cdot E+E, \$/+/*$
$E \to \cdot E*E, \$/+/*$
$E \to \cdot id, \$/+/*$

E → **I6**

id → **I2**

I6
$E \to E*E \cdot, \$/+/*$
$E \to E \cdot +E, \$/+/*$
$E \to E \cdot *E, \$/+/*$

\+ → **I3**

* → **I4**

E → **I5**

id → **I2**

I5
$E \to E+E \cdot, \$/+/*$
$E \to E \cdot +E, \$/+/*$
$E \to E \cdot *E, \$/+/*$

\+ → **I3**

* → **I4**

States	ACTIONS				GOTO
	+	*	id	$	E
0			S2		1
1	S3	S4		accept	
2	R3	R3		R3	
3			S2		5
4			S2		6
5	S3/R1	S4/R1		R1	
6	S3/R2	S4/R2		R2	

Viable prefix: (valid prefix)

prefixes of Right sentential form can appear on the stack of a SRP (shift reduce parser):

Ex:

Grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

$E \Rightarrow F*id \Rightarrow (E)*id$ (rmd)

handle = (E)

except handle all the previous items are viable prefixes

$\therefore \Rightarrow$ viable prefix = { (, (E }

proper prefix of handles

↓ except the handle

viable prefix

NOTE: can't include ε.

Unit-1

Introduction to Automata

DFA : Deterministic Finite Automata

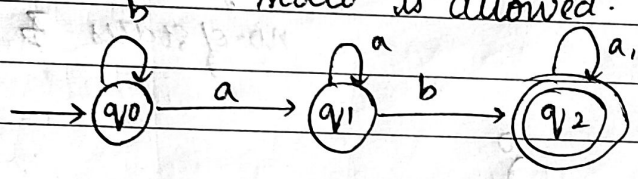
* consists of 5-tuple machine

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

- Q → finite no. of states in building the machine
- Σ → character, symbol i/p required to build the machine
- δ → transition f^n $(Q \times \Sigma)$
- q_0 → initial state
- F → Final state

* only one deterministic path from one state to another state is allowed.

Ex:



$\Sigma = \{a, b\}$

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

$\delta(q_0, a) = q_1$

$\delta(q_1, b) = q_2$

$q_0 \rightarrow$ initial state / start state

$F_2 \rightarrow$ final state

Transition table

Present state	Next state for input a	Next state of input b
→ q_0	q_1	q_0
q_1	q_1	q_2
* q_2	q_2	q_2

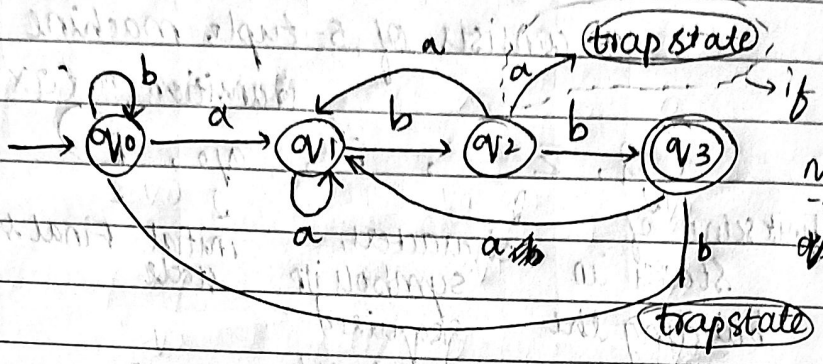
* Regex: $(a|b)^* abb$ = { abb, aabb, babb, ... }

length = 3 (min) so, atleast 4 states should exist

condition: ends with "abb" or can be "abb" alone

construct the DFA

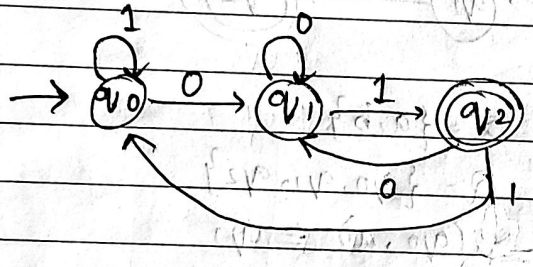
$\Sigma = \{a, b\}$



* Regex: $(0|1)^* 01$ = { 001, 101, 0001, ... }

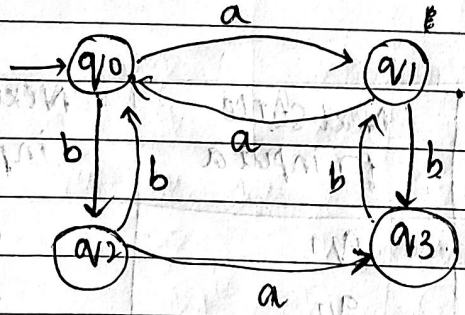
min length = 2

no. of states = 3



* Accept even a's & even b's including empty string.

= { empty, aabb, bbaa, abba, baab, ... }



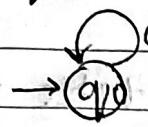
q_0 → a, b are even

q_1 → odd a's, even b's

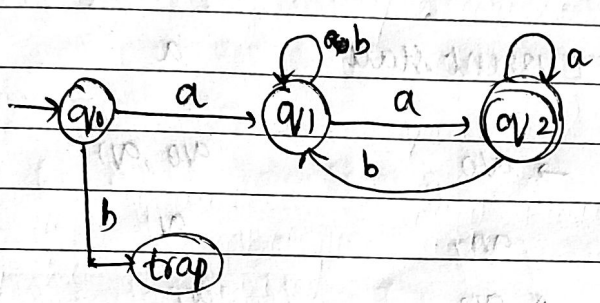
q_2 → even a's, odd b's

q_3 → odd a's, odd b's

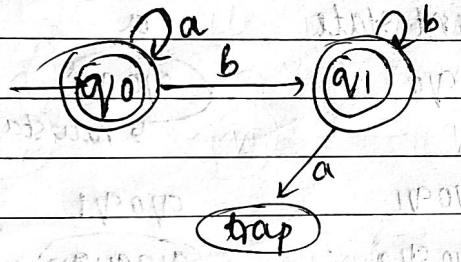
* construct DFA for (a|b)*



* Construct DFA for a (a|b)* a



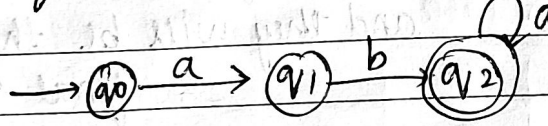
* construct DFA for a* b* = {ε, a, b, ab, bb, ...}



NFA (Non-Deterministic Finite Automata)

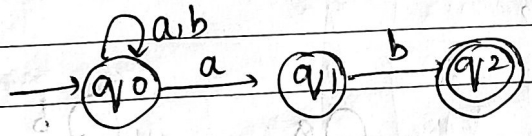
5 tuple (Q, Σ, δ, q0, F)

* NFA for ab(a|b)*



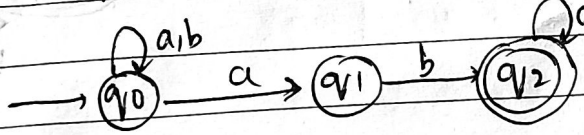
also a DFA

* NFA for (a|b)*ab



not a DFA

* NFA for (a|b)* ab (a|b)*



classmate
Date
Page

* Conversion of NFA to DFA

Step 2: Draw the transition table for NFA

For previous example:

current state	a	b
→ q_0	q_0, q_1	q_0
q_1	q_1	q_2
* q_2	q_2	q_2

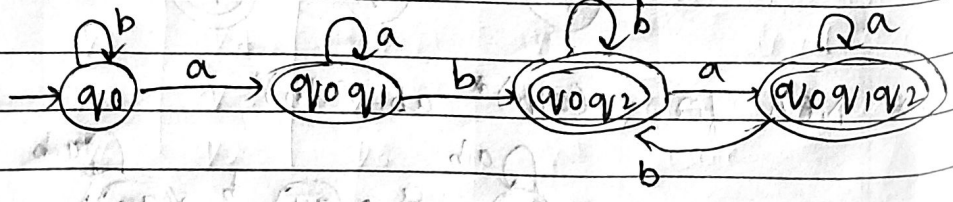
Step 2: construct transition table for DFA

current state	a	b
→ q_0	q_0q_1 → new state, so include next	q_0
q_0q_1	q_0q_1	q_0q_2 → new state so indu
* q_0q_2	$q_0q_1q_2$ → new state, so include	q_0q_2
* $q_0q_1q_2$	$q_0q_1q_2$	q_0q_2

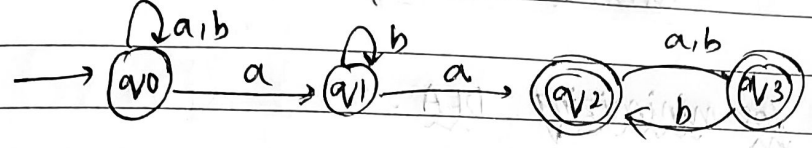
Check in NFA, q_2 was final state
check for q_2 in DFA transition table
and they will be the the transition
final state for DFA

Step 3: construct DFA

q_0, q_1, q_2



4. Given NFA convert to DFA



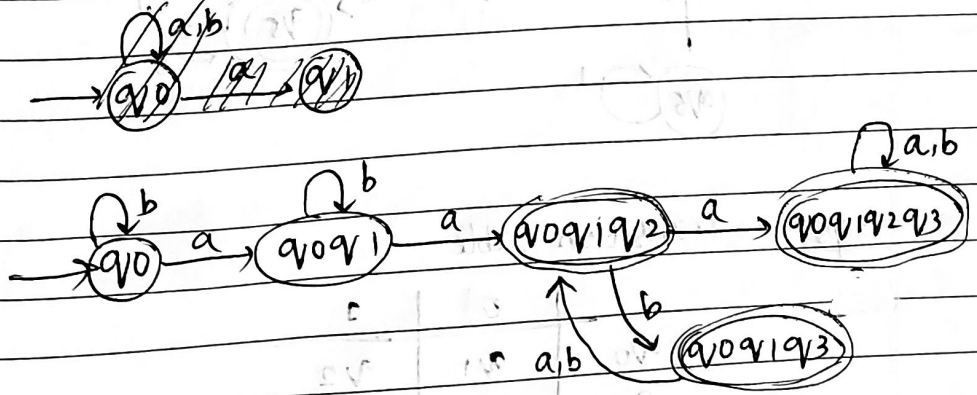
NFA:

current state	a	b
→ q_0	$q_0 q_1$	q_0
q_1	q_2	q_1
* q_2	q_3	q_3
* q_3	—	q_2

DFA:

current state	a	b
→ q_0	$q_0 q_1$	q_0
$q_0 q_1$	$q_0 q_1 q_2$	$q_0 q_1$
* $q_0 q_1 q_2$	$q_0 q_1 q_2 q_3$	$q_0 q_1 q_3$
* $q_0 q_1 q_2 q_3$	$q_0 q_1 q_2 q_3$	$q_0 q_1 q_3 q_2$
* $q_0 q_1 q_3$	$q_0 q_1 q_2$	$q_0 q_1 q_2$

DFA construction:



Minimization of DFA

Steps for minimizing DFA:

i) Remove the unreachable / dead state.

ii) construction of the transition table for DFA with Σ, as set of alphabets

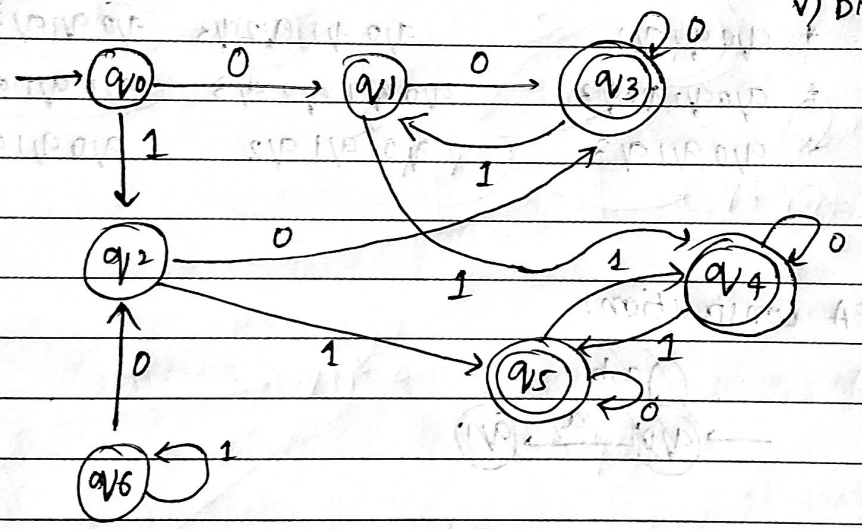
iii) Divide into equivalance classes i) states which are not final / accepting

ii) Final / accepting states.

iv) Continue finding equivalence classes until no new ones are found.

v) Draw final DFA

Ex:



Step1: Transition table

	0	1
→ q_0	q_1	q_2
q_1	q_3	q_4
q_2	q_3	q_5
q_3	q_3	q_1
q_4	q_4	q_5
q_5	q_5	q_4
* q_6	q_2	q_6

now, unreachable / dead state.

q_6 → is unreachable state.

So, remove it.

step2: current transition table:

	0	1
→q_0	q_1	q_2
q_1	q_3	q_4
q_2	q_3	q_5
q_3	q_3	q_1
q_4	q_4	q_5
q_5	q_5	q_4

step 3: C1 (not final states) $\{q_0, q_1, q_2\}$ C2 (final states) $\{q_3, q_4, q_5\}$

check for more equivalence classes,

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

i) $\{q_0, q_1\}$ with 0 = q_1 q_3 (in C1, in C2)
both in different classes so discard
no, need to check with $\{q_0, q_1\}$ with 1.

ii) $\{q_0, q_2\}$ with 0 = q_1 q_3
same problem so discard
& no need to check $\{q_0, q_2\}$ with 1.

iii) $\{q_1, q_2\}$ with 0 = q_3 q_5 } all belong to same class C2
$\{q_1, q_2\}$ with 1 = q_4 q_5 }

so, accept it.

& it can be written as $\{q_0\}$ $\{q_1 q_2\}$
q_0 as seperate class bcuz it didn't satisfy with 0 & 1.

Step4: Similarly for $\{q_3, q_4, q_5\}$

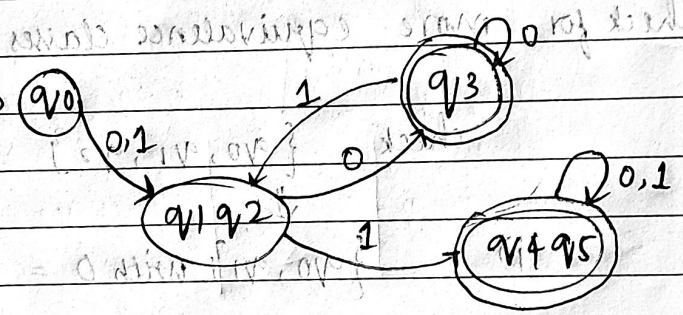
$\Rightarrow \{q_3\}\ \{q_4, q_5\}$

1st equivalence $\Rightarrow \{q_0\}\ \{q_1 q_2\}\ \{q_3\}\ \{q_4 q_5\}$

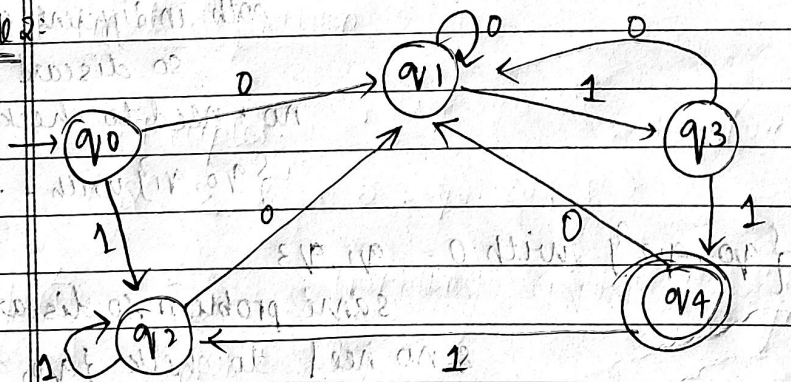
2nd equivalence → further q_0, q_3 can't be classified as they are isolated and remaining $\{q_1, q_2\}$ & $\{q_4, q_5\}$ can't be further classified.

so, 2nd equivalence: $\{q_0\}\ \{q_1 q_2\}\ \{q_3\}\ \{q_4 q_5\}$

Step5: Draw finalized DFA



Example 2:



No dead/unreachable state.

Step2&2: Transition table:

	0	1
→ q_0	q_1	q_2
q_1	q_1	q_3
q_2	q_1	q_2
q_3	q_1	q_4
* q_4	q_1	q_2

so, no removal of any dead or unreachable states

classmate
Date
Page

Step3: $C1 = \{q0\ q1\ q2\ q3\}$ $C2 = \{q4\}$

check $q0q3$ with 0 $\Rightarrow$ $q1, q1$ } different classes
$q0q3$ with 1 $\Rightarrow$ $q2, q4$ } so, discard

$q1q3$ with 0 $\Rightarrow$ $q1, q1$ } different classes
$q1q3$ with 1 $\Rightarrow$ $q3, q4$ } so, discard.

$q2\ q3$ with 0 & 1 $\Rightarrow$ discard.

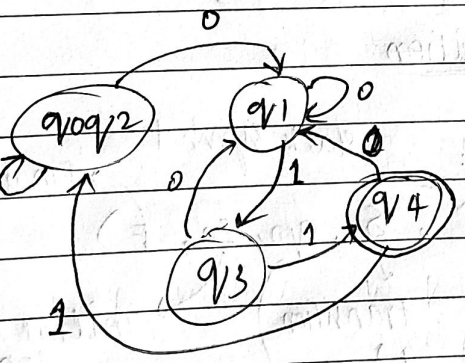
So,

1st equivalence : $\{q0\ q1\ q2\}\ \{q3\}\ \{q4\}$

2nd equivalence : $q0q1$ with 0 $= \{q1, q1\}$ } $\Rightarrow$ discard.
$q0q1$ with 1 $= \{q2, q3\}$

$q1q2$ with 0 $= \{q1\ q1\}$ } $\Rightarrow$ discard.
$q1q2$ with 1 $= \{q2\ q1\}$

So, 2nd equivalence : $\{q0q2\}, \{q1\}, \{q3\}, \{q4\}$



UNIT-2

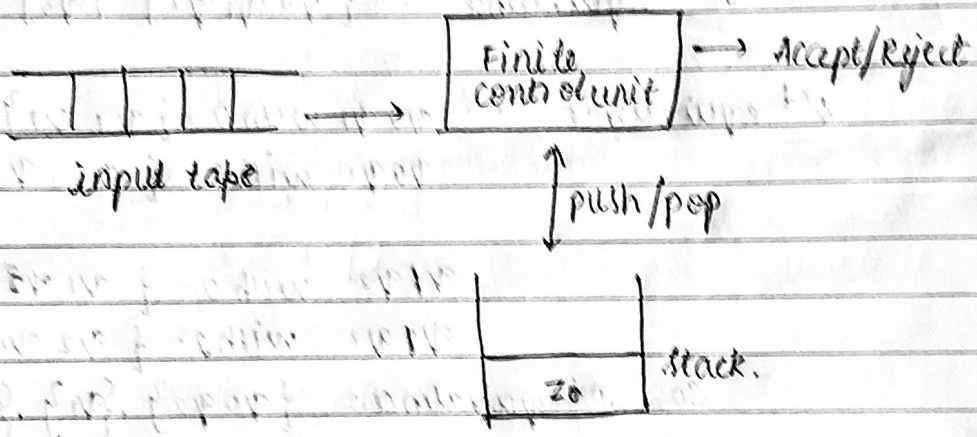
Push Down Automata (PDA):

→ a way to implement a CFG

→ PDA has 3 components.

- input tape
- control unit
- stack with infinite size.

PDA has to read the top of the stack in every transition



Formal definition:

7 tuples: → stack symbol → Final/accepting state

$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

- Q: finite set of states
- Σ: input alphabets
- δ: transition fn
- q_0: state/actual state
- Z_0: initial stack top symbol.

* Construct a PDA which all will accept strings for language

$L = \{ 0^n 1^{(2n)}, n \geq 1 \}$ → no. of times it occurs.

The no. of 1s is the twice the no. of 0s

Push phase:

- For each 0 encountered in the input, push a symbol (e.g., X) onto the stack
- This keeps track of the number of 0s

pop phase:

- When encountering the 1s, pop one symbol from the stack for every two 1s
- This ensures there are exactly twice as many 1s as 0s.

Acceptance:

After processing the entire input, the stack should be empty, and the automation will accept the string.

I/p string: 001111 ε_0

state, i/p, stack top

$\delta(q_0, 0, z_0) = (q_0, 0z_0)$

$\delta(q_0, 0, 0) = (q_0, 0, 0)$

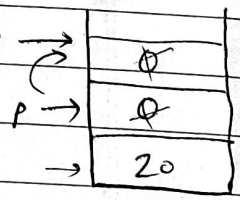
$\delta(q_0, 1, 0) = (q_1, 0)$

$\delta(q_1, 1, 0) = (q_2, \varepsilon_0)$ → pops operation

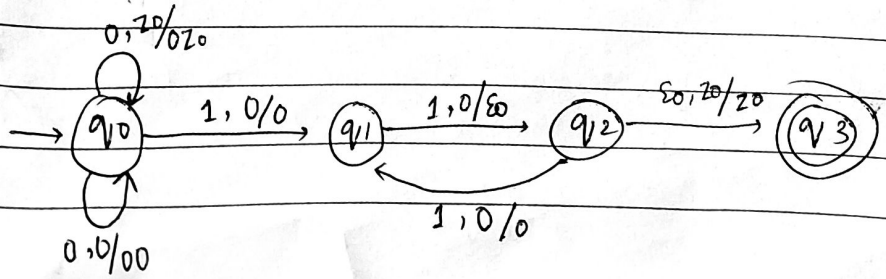
$\delta(q_2, 1, 0) = (q_1, 0)$

$\delta(q_1, 1, 0) = (q_2, \varepsilon_0)$

$\delta(q_2, \varepsilon_0, z_0) = (q_3, z_0)$



(when 2nd 1 is encountered, move to new state & then pop)



UNIT-IV

Synthesized attribute:

↳ value is computed from children or node itself.

SDD → S attributed definition
↓
synthesized

Inherited attribute:

↳ value is computed from parent/siblings /itself.

Top Down parser stack (use of inherited attribute)

$D \rightarrow DB \mid B$ it has left recursion so eliminate it

$B \rightarrow 0 \mid 1$

↓

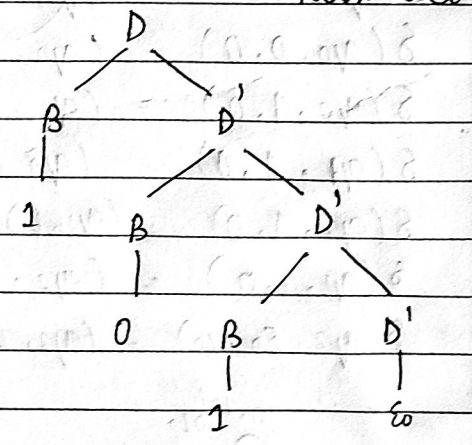
$D \rightarrow BD'$

$D' \rightarrow BD' \mid \varepsilon$ } ⇒ suitable for top-down parser

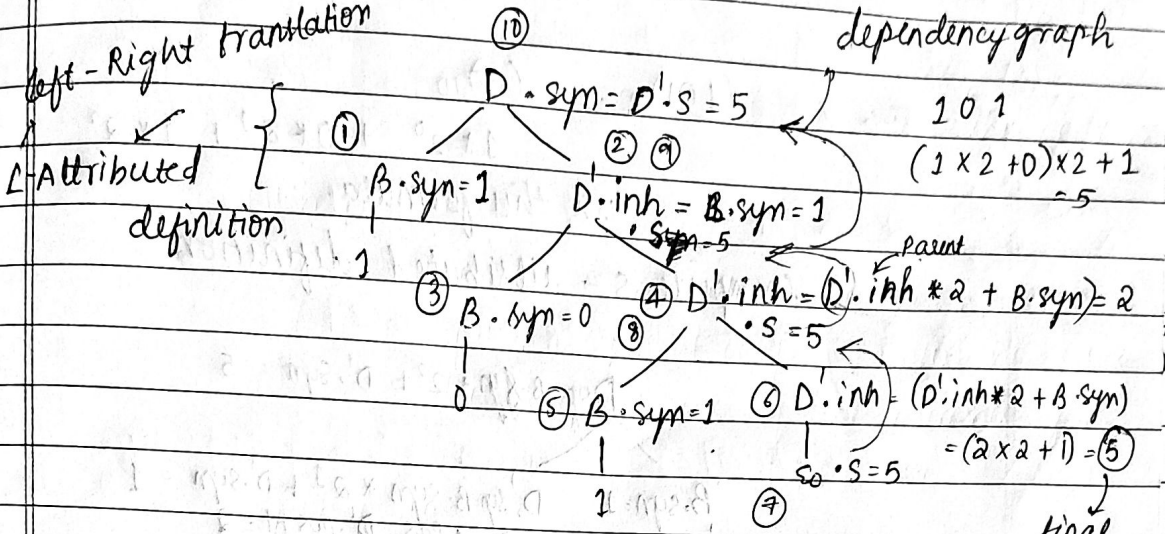
$B \rightarrow 0 \mid 1$

i/p :- 101

parse tree



⇒ convert parse tree to Annotated parse tree (post order traversal) so, start from left



production	Semantic Rule
$D \rightarrow B D'$	$D'.inh = B.syn$ $D.syn = D'.S$
$D' \rightarrow B D_1'$ (child)	$D_1'.inh = D'.inh * 2 + B.syn$ $D_2'.S = D_1'.S$
$D' \rightarrow \varepsilon$	$D'.S = D'.inh$
$B \rightarrow 0$	$B.syn = 0$
$B \rightarrow 1$	$B.syn = 1$

now pass this final result to root to print the result.

So, create a new synthesised attribute(S) to pass the result from leaf to root & then at root, write it as $D.syn = D'.S = 5$ because it will hold n print the result.

Dependency Graph:

2

DAG to find the order in which the parser executes.

An edge is made b/w two consecutive nodes that are performed one after another.

$i \rightarrow j$ then 'i' should be executed before 'j'.

Then write the topological sort.

Example:

$D \rightarrow BD'$

$D' \rightarrow BD' \mid \varepsilon_0$

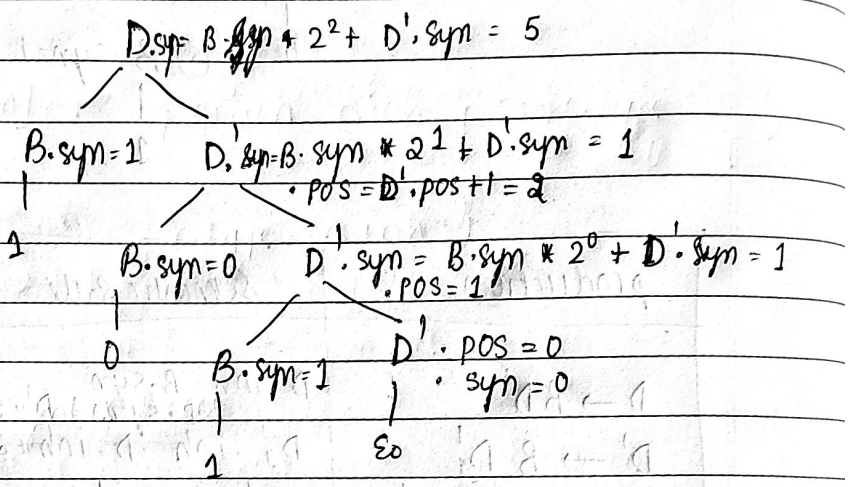
$B \rightarrow 0 \mid 1$

$(101)_2 = (5)_{10}$

$= 1 * 2^0 + 0 * 2^1 + 1 * 2^2$

use this formula.

Compute S - attributed definition.



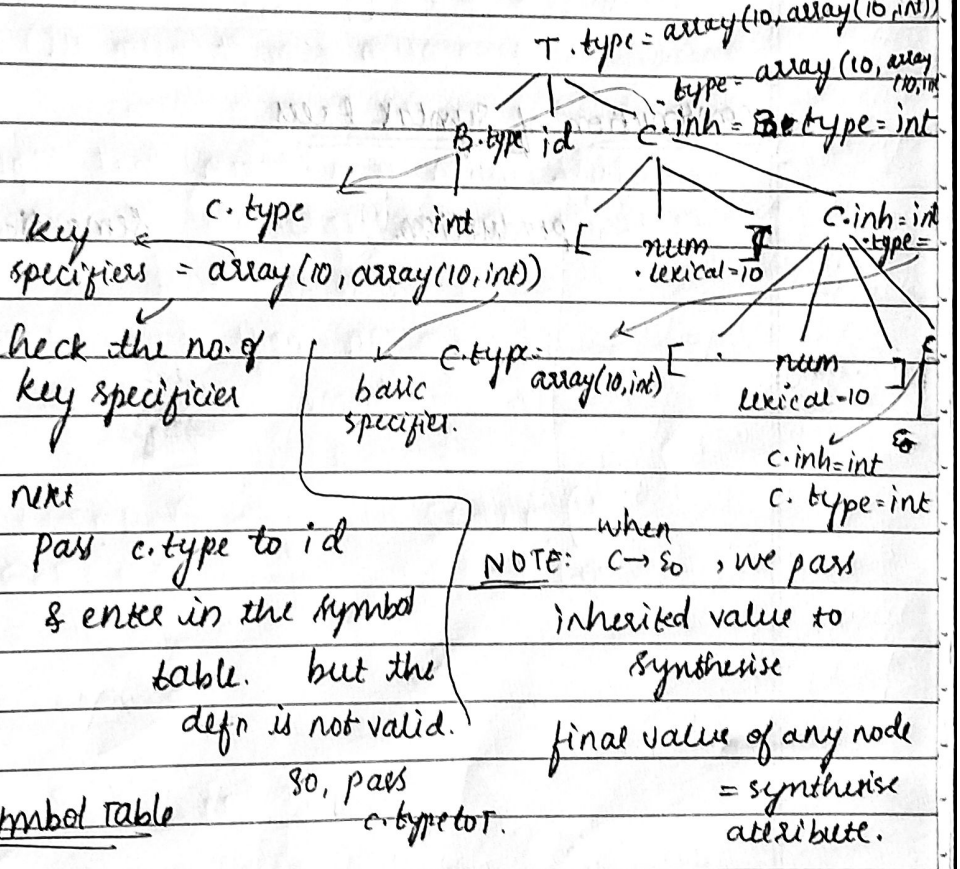
Application of SDT

SDD for array type declaration

production	Semantic Rules
$T \rightarrow B\ id\ C$	C.inh = B.type = int T.type = C.type & addType(id.entry, T.type)
$B \rightarrow int$	B.type = int
$B \rightarrow float$	B.type = float
$C \rightarrow [num]\ C_1$	C_1.inh = C.inh & C.type = array(num.lexical, C_1.type)
$C \rightarrow \varepsilon$	C.type = int

Parse tree.

"type = syn." i/p: int a[10][10]



check the no. of key specifiers

next pass c.type to id & enter in the symbol table. but the defn is not valid.

So, pass c.type to T.

NOTE: when $C \rightarrow \varepsilon$, we pass inherited value to synthesise

final value of any node = synthesise attribute.

Symbol Table

Ref	LV	type
1	a	)nt[10][10]

so, addType (id.entry, T.type)

SDT for array type declaration.

Translate SDD to SDT by placing all synthesize at the end & inherited at the beginning of the body of production

production	Semantic Rules / Translation
$T \to B\ id\ C$	$\to B\ id\ \{C.inh = B.type;\}\ C\ \{T.type = C.type;\}$
$B \to int$	$\{B.type = int;\}$
$B \to float$	$\{B.type = float;\}$
$C \to [num]\ C_1$	$\{C_1.inh = C.inh;\}\ C_1\ \{C.type = array(num.lexical, C_1.type);\}$
$C \to \varepsilon$	$\{C.type = C.inh;\}$

⇒ Construction of Syntax trees

production	Semantic Rules.

Eliminating Left recursion from SDTs

SDT for postfix SDT
(Syntax Directed Translation)

$A \longrightarrow A_1 Y \{ A.a = g(A_1.a, Y.y) \}$
$A \longrightarrow X \{ A.a = f(X.x) \}$

X, Y → tokens / terminals

A.a → Synthesized attribute of left-recursive nonterminal A & X, Y are single grammar symbols with synthesised attributes X.x & Y.y

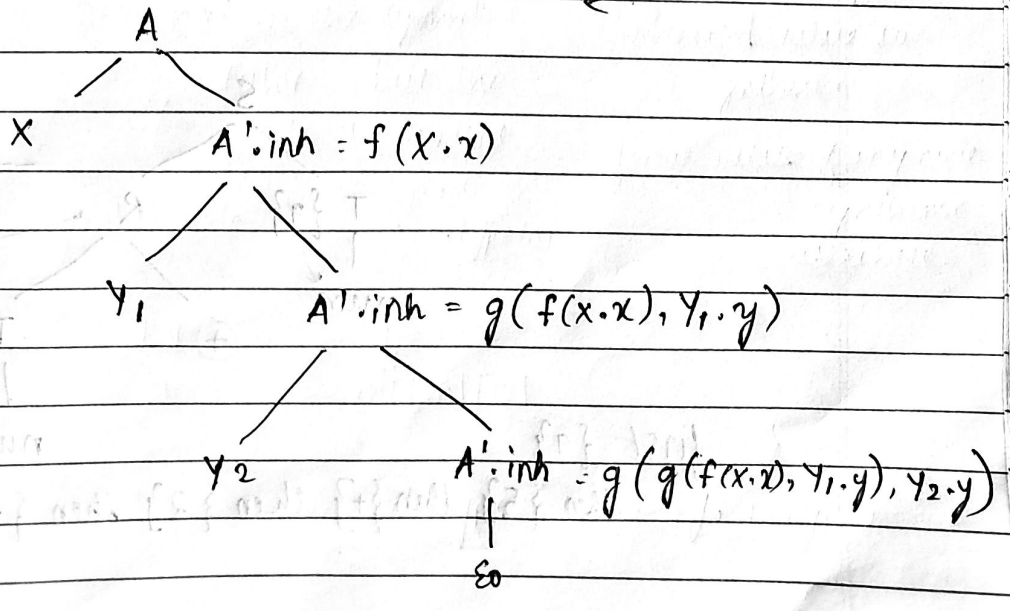
Actions are written in the end of body of production.

g( ) → group computation.
f → functional computation

$A \longrightarrow AX$
$A \longrightarrow X$

$A \longrightarrow XA'$
$A' \longrightarrow YA' / \varepsilon$

Annotated parse tree



SDT for L-attributed Defn:

$A \longrightarrow X \{A'.inh = f(X.x)\}\ A'$
$A' \longrightarrow Y \quad A'$
$A' \longrightarrow \varepsilon$

* I/p: 9+5+2

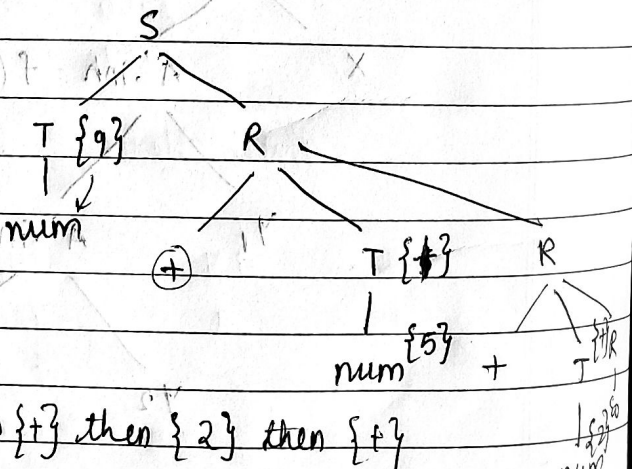
Consider the translation scheme shown

$S \longrightarrow TR$
$R \longrightarrow +T \{print('+');\} R \mid \varepsilon$ → actions
$T \rightarrow num \{print(num.val);\}$

Here num is a token that represents an integer and num.val represents the corresponding integer value.

For an input string '9+5+2', this translation scheme will print:

a) 9+5+2
~~b)~~ 95+2+
c) 952++
d) ++952



first {9} then {5} then {+} then {2} then {+}

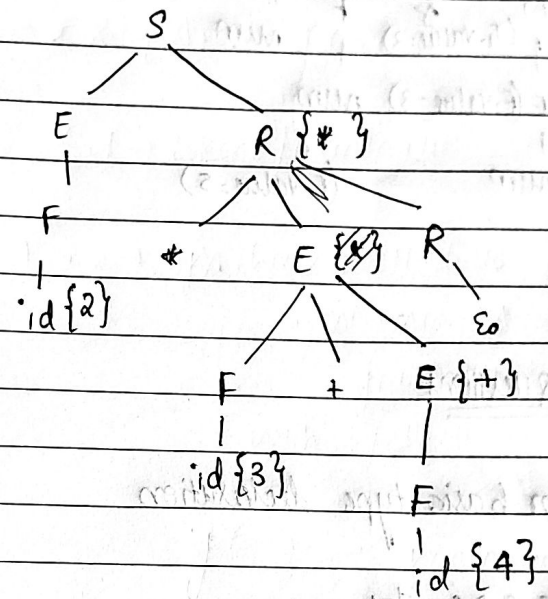
* $S \rightarrow ER$

$R \rightarrow *E \{print('*');\} R \mid \varepsilon_0$

$E \rightarrow F+E \{print('+')\} \mid F$

$F \rightarrow (S) \mid id \{print(id.value);\}$

i/p : '2*3+4'



a) 2*3+4

~~b) 23*4+*~~

c) 23*4+

d) 2*+34

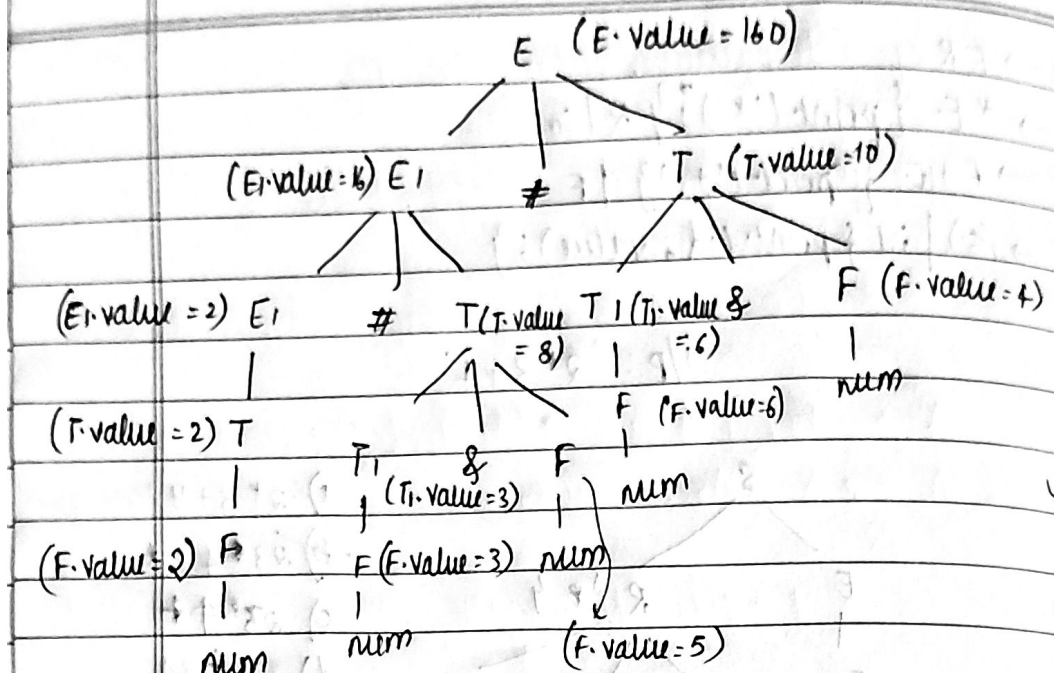
* consider the grammar with the following translation rules and E as the start symbol.

$E \rightarrow E_1 \# T \quad \{E.value = E_1.value * T.value\} \mid T \quad \{E.value = T.value\}$

$T \rightarrow T_1 \& F \quad \{T.value = T_1.value + F.value\} \mid F \quad \{T.value = F.value\}$

$F \rightarrow num \quad \{F.value = num.value\}$

compute E.value for the root of the parse tree for the expression: 2# 3 & 5 # 6 & 4



a) 200
~~b) 160~~
c) 180
d) 40

Tutorial Questions

① Write SDT for basic type declaration

② Draw APT & DG for

i) float a, b, c, d

ii) int a[2]

iii) int a[2][3][2]