Since above has no left recursion and no leftmost common factor, we have built parse table directly.

1)3. It has no leftmost common factor or left recursion, so we build table directly.

NT \ input	a	b	c	d	$
S	$S \to ABC$				
A	$A \to aB$				
B		$B \to \varepsilon$, $B \to bB$	$B \to \varepsilon$		$B \to \varepsilon$
C			$C \to c$		$C \to \varepsilon$
D				$D \to d$	

Follow(B) = {c, $, b} Follow(C) = {$}

1)4. It has no left recursion or leftmost common factor, so we build the parse table directly.

NT \ input	a	b	$
S	$S \to AB$		
A	$A \to aB$		
B		$B \to b$ / $B \to \varepsilon$	$B \to \varepsilon$

1)2. Here, there is direct left recursion.
In $L \to L, id \mid id$, we remove it as $L \to id\, L'$,
$L' \to , id\, L' \mid \varepsilon$

The grammar is now
$S \to L = R$
$L \to id\, L'$
$L' \to , id\, L' \mid \varepsilon$
$R \to r L$

There is no leftmost common factor so we build parse tree directly. Follow(L') = {=, $}

LALR → rightmost derivation
look-ahead → left-to-right

NT \ input	=	id	,	r	$
S		$S \to L=R$			
L		$L \to id L'$			
L'	$L' \to \varepsilon$		$L' \to , id L'$		$L' \to \varepsilon$
R				$R \to rL$	

start from here for CLR-II

2/12/24* **LALR(1) parser** (most space-consuming parser) → but most powerful also → inefficient

1. Construct $C = I_0, I_1, \ldots, I_n$ from LR(1) sets
2. Merge the lookahead character if RSF (right sentential form) is the same → first component

RSF → don't consider look-ahead, merge those two items.
Eg: → Consider $S \to cc$, $C \to cC \mid d$ } check notes before for item sets in CLR(1)

In the item sets, I_3 and I_6 are the same, except for their lookaheads, so we merge them as:

I_{36} = $C \to c.C,\ c|d|\$$
$C \to .cC,\ c|d|\$$
$C \to .d,\ c|d|\$$

Similarly, I_4 and I_7 are the same, so

I_{47}
$C \to c\to d.,\ c|d|\$$

Also, I_8 and I_9 are the same, so

I_{89}
$C \to c.C,\ c|d|\$$ } check

Final items: I_0, I_1, I_2, I_{36}, I_{47}, I_5, I_{89}

Page No.
Date. / /

Parse table:

	Action			GOTO	
State	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		

Eg: Do LALR for $E \to E+E$, $E \to E*E$, $E \to id$

I_0
$E' \to .E, \$$
$E \to .E+E, \$|+|*$
$E \to .E*E, \$|+|*$
$E \to .id, \$|+|*$

$\xrightarrow{E}$ I_1
$E' \to E., \$$
$E \to E.+E, \$|+|*$
$E \to E.*E, \$|+|*$

$\xrightarrow{+}$ I_2
$E \to E+.E, \$|+|*$
$E \to .E+E, \$|+|*$
$E \to .E*E, \$|+|*$
$E \to .id, \$|+|*$

$\xrightarrow{id}$ I_2
$E \to id., \$|+|*$

$\xrightarrow{*}$ I_4
$E \to E*.E, \$|+|*$
$E \to .E+E, \$|+|*$
$E \to .E*E, \$|+|*$
$E \to .id, \$|+|*$

$\xrightarrow{id}$ I_2

$\xrightarrow{E}$ I_5
$E \to E+E., \$|+|*$
$E \to E.+E, \$|+|*$
$E \to E.*E, \$|+|*$

$\xrightarrow{+}$ I_3

$\xrightarrow{*}$ I_4

I_6
$E \to E*E., \$|+|*$
$E \to E.+E, \$|+|*$ $\xrightarrow{+}$ I_3
$E \to E.*E, \$|+|*$ $\xrightarrow{*}$ I_4

Page No.
Date. / /

Parse table:

	Action				GOTO
State	+	*	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	

Q) Do LALR parsing for $S \to L=R \mid R$, $L \to *R \mid id$, $R \to L$

check why no $|\$$ in the last 3

I_0
$S' \to .S, \$$
$S \to .L=R, \$$
$S \to .R, \$$
$L \to .*R, =|\$$
$L \to .id, =|\$$
$R \to .L, \$$

$\xrightarrow{S}$ I_1
$S \to S., \$$

$\xrightarrow{L}$ I_2
$S \to L.=R, \$$
$R \to L., \$$

$\xrightarrow{=}$ I_6
$S \to L=.R, \$$
$R \to .L, \$$
$L \to .*R, \$$
$L \to .id, \$$

$\xrightarrow{R}$ I_9
$S \to L=R., \$$

$\xrightarrow{R}$ I_3
$S \to R., \$$

$\xrightarrow{R}$ I_7
$L \to *R., =|\$$

$\xrightarrow{id}$ I_5
$L \to id., \$|=$

$\xrightarrow{*}$ I_4
$L \to *.R, =|\$$
$R \to .L, \$|=$
$L \to .*R, \$|=$
$L \to .id, \$|=$

$\xrightarrow{*}$ I_4

$\xrightarrow{id}$ I_5

$\xrightarrow{L}$ I_8
$R \to L., \$|=$

check * Viable Prefix (Valid Prefix) → proper prefixes of handles

Prefix in the RSF that can appear in the top of a SRP (shift-reduce parser)

$E \xrightarrow{rmd} \ldots \xrightarrow{rmd} F*id \xrightarrow{rmd} (E)*id$ → final answer

↳ handle

~~CE, C, CE~~ handle should not be left on top of stack

Page No.
Date. / /

3/12/24

UNIT - I

Introduction to Automata

* DFA - Deterministic Finite Automata ~~(DFA)~~

5-tuple format - $(Q, \Sigma, \delta, q_0, F)$

where Q = finite set of states (components)

Σ = finite set of alphabets (symbol)

δ = transition function (delta)

q_0 = initial / start state

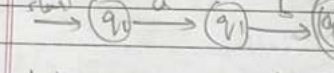
F = final / accepting state

Deterministic - there should be only one deterministic path for each token / character

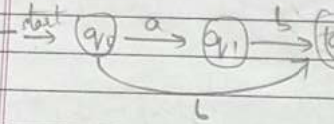
Start state — start → q_0

Final state — → q_2

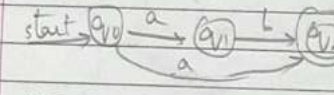
Deterministic means that on a state there should be no more than one ~~state~~ transition ~~from~~ for a particular character to another particular state.



✓ deterministic



✓ deterministic



X not deterministic

$Q = q_0, q_1, q_2$

$\Sigma = a, b$

$q_0 = q_0$

$F = q_2$

Transition functions:

$\delta(q_0, a) = q_1$

$\delta(q_0, b) = q_2$

$\delta(q_1, b) = q_2$

transition function from q_0 on a to q_1

should represent start state as start q_0 and final as *q_2 or q_2 in table

Page No.
Date. / /

Transition table

Q	a	b
start q_0	q_1	q_2
q_1	—	q_2
* q_2	—	—

or q_2

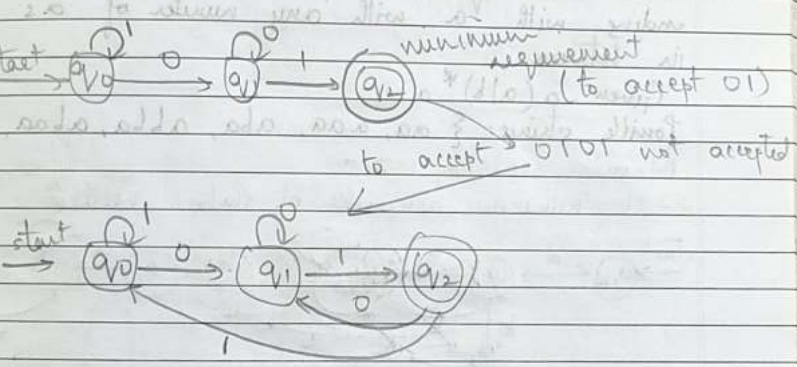
Q) Given $(0|1)^*01$, give DFA for strings that can be accepted

$\Sigma = 0, 1$

Accepted strings : { 01, 001, ~~00~~ 101, 0001, 1101, 0101, 1001 ... }

minimum substring to be accepted + 1 = number of min state in accepted

Eg: 0+1 = 2+1 = 3 states (0,1,2) for min req.



check for atleast some 10 different strings and stop in exam

Transition table

Q	0	1
start q_0	q_1	q_0
q_1	q_1	q_2
* q_2	q_1	q_0

not be written start also

Trap state

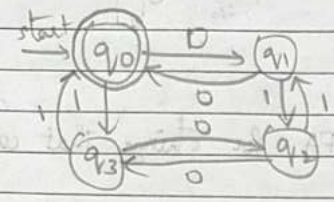
011 not accepted, so when this occurs, we should stop from going to final state

Q) Make a DFA which can accept Σ, even number of

start state is final state if ε_0 should get accepted

Page No.
Date. / /

1s and even number of 0s.
Possible strings : { ε_0, 00, 11, 0011, 1100, 1010, 0101, 0110, 1001 ... }



Transition diagram

Q	0	1
start → q_1		
q_2		
q_3		
* q_4		

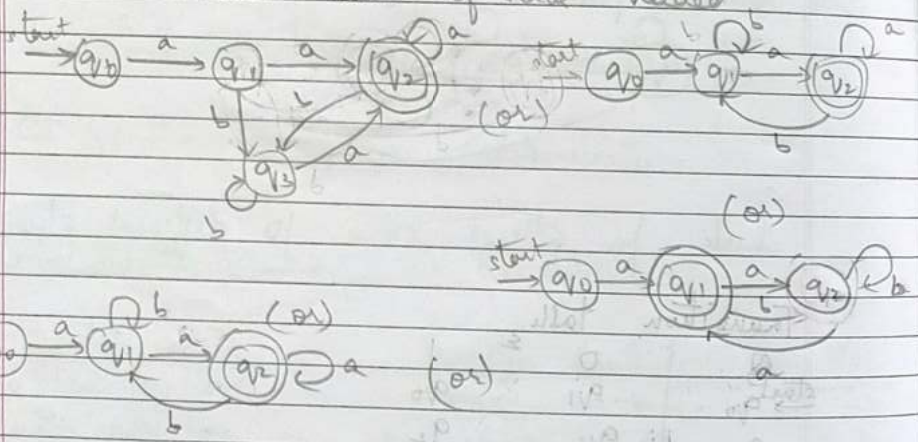
Q) Make DFA for the regular expression starting and ending with a with any number of a's or b's in between.

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

Possible strings: { aa, aaa, aba, abba, abaa, ababa ... }

Minimum substring = 2

2+1 = minimum number of state needed



Q) Make DFA for even a's and odd b's ~~and ε_0~~.

Possible strings : { ε_0, aab, aabaa, aabbb, ababb, baa, ababbaa ... }

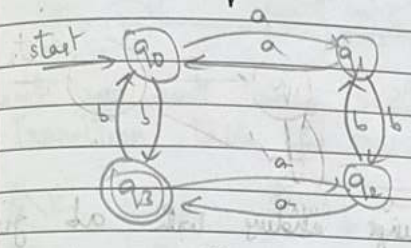
aba

Page No.
Date. / /

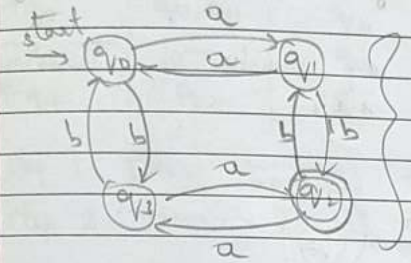
Q) Make DFA for even b's and odd a's, no ε_0

Q) Make DFA for odd b's and odd a's, no ε_0

4/12/24 1)

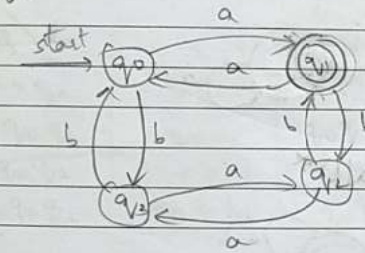


3)

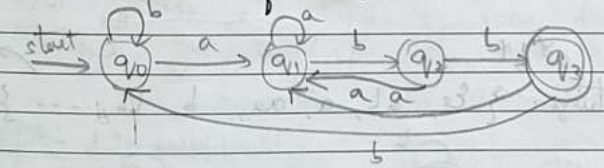


01
000111
101010
odd a's and b's

2) even b's and odd a's



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



Page No.
Date. / /

* Non-Deterministic Finite Automata (NFA)

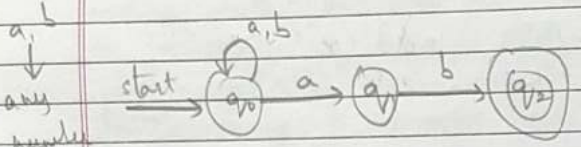
5-tuples $(Q, \Sigma, \delta, q_0, F)$ same as DFA components

(Q → states, Σ → alphabets, q_0 → start state, F → final state; δ → transition functions)

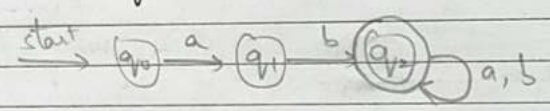
→ NFA and DFA can have more than one final state, if required

Q) Construct NFA for strings ending with ab given $(a|b)^*ab$

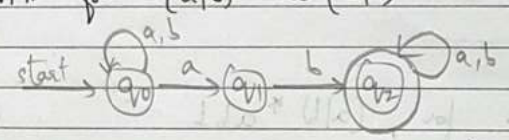
NFA – no restriction on number of transitions for one character on a state



Q) NFA for $ab(a|b)^*$



Q) NFA for $(a|b)^*ab(a|b)^*$



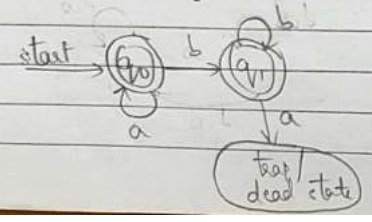
Q) NFA for a^*b^*

Possible strings: { ε, ab, a, aa, b, bb, ... }

ba not allowed

to make machine understand that the input has to be rejected

Trapp/dead state – if there is an 'a' after 'b', here should be a dead state to indicate error, show deadend/cannot be processed, not allowed to go to final state

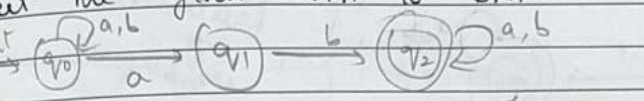


can be asked as:
1. given regular expression, give NFA, DFA and convert NFA → DFA
2. given transition table, give NFA, DFA and convert NFA → DFA

Page No.
Date. / /

* Conversion of NFA to DFA

Q) Convert the given NFA to DFA



Transition table for NFA $(Q \times \Sigma)$ (rows, columns)

$Q \backslash \Sigma$	a	b
→ q_0	q_0, q_1	q_0
q_1	—	q_2
* q_2	q_2	q_2

→ step 1: Draw transition table for NFA
step 2: Draw transition table for DFA

Transition Table for DFA

$Q \backslash \Sigma$	a	b
→ q_0	q_0q_1	q_0
q_0q_1	q_0q_1	q_0q_2
* q_0q_2	$q_0q_1q_2$	q_0q_2
* $q_0q_1q_2$	$q_0q_1q_2$	q_0q_2

q_0 on a union q_2 on a → new state from previous row

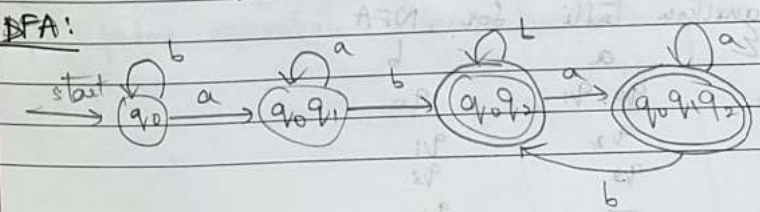
consider as new state (union of q_0q_1)

only start state remains same, rest all to be computed

since there are no state for which we haven't drawn transitions/ all have identified it is done

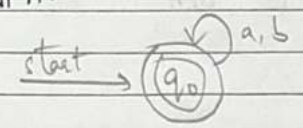
since q_2 was final state in NFA, all states with q_2 in DFA transition table are final states

DFA:



Q) Convert $(a|b)^*$ to DFA

NFA:



transition diagram

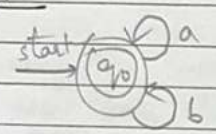
Transition table for NFA

$Q \backslash \Sigma$	a	b
$\rightarrow * q_0$	q_0	q_0

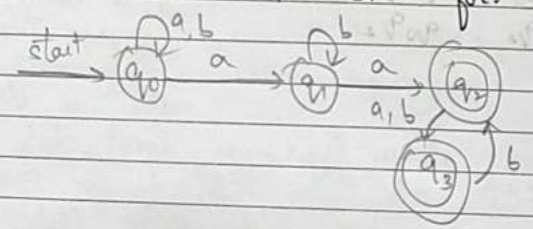
Transition table for DFA

$Q \backslash \Sigma$	a	b
$\rightarrow * q_0$	q_0	q_0

DFA:



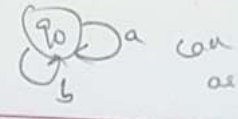
Q) Convert NFA to DFA for:



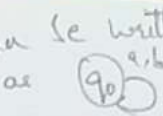
Transition table for NFA

$Q \backslash \Sigma$	a	b
$\rightarrow q_0$	q_0, q_1	q_0
q_1	q_2	q_1
$* q_2$	q_3	q_3
$* q_3$	—	q_2

For DFA also,



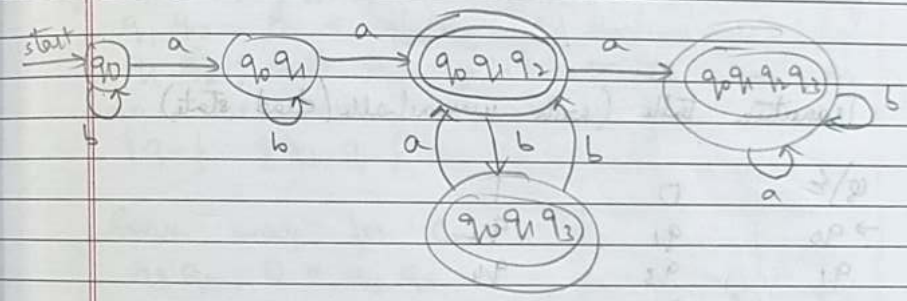
can be written as



Transition table for DFA:

$Q \backslash \Sigma$	a	b
$\rightarrow q_0$	$q_0 q_1$	q_0
$q_0 q_1$	$q_0 q_1 q_2$	$q_0 q_1$
$* q_0 q_1 q_2$	$q_0 q_1 q_2 q_3$	$q_0 q_1 q_3$
$* q_0 q_1 q_2 q_3$	$q_0 q_1 q_2 q_3$	$q_0 q_1 q_2 q_3$
$* q_0 q_1 q_3$	$q_0 q_1 q_2$	$q_0 q_1 q_2$

DFA:

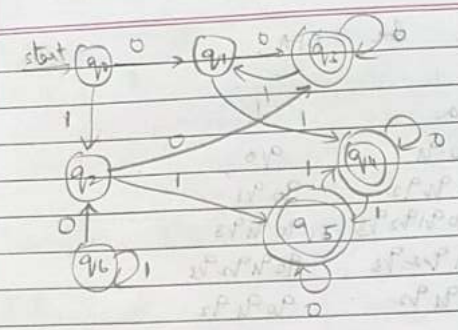


10/9/24 * ~~Minimal~~ Minimization of DFA

→ Steps

1. Remove all the state that are not reachable (unreachable) from initial state via any set of transitions of DFA
2. Construct new transition table by removing ~~the~~ unreachable / dead states
3. Split the transition table into two classes (equivalence classes)
   Class 1 contains all non-final states
   Class 2 contains all final / accepting states
4. After finding equivalence classes, draw the DFA.

Eg:



Step 1: Search for all state which do not have path from source (start state) to itself – for example q_6 called non-reachable / dead state
Construct transition table with dead state

Transition table (with unreachable / dead state)

$Q \backslash \Sigma$	0	1
$\rightarrow q_0$	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

Step 2: Transition table without dead / unreachable state without q_6 in the table here)

Step 3: Equivalence classes

Class 1 — C_1 (q_0, q_1, q_2) all non-final states
Class 2 — C_2 (q_3, q_4, q_5) all final states
called 0-equivalence class

We have to construct 1-equivalence class using 0-equivalence class as input
Take combinations as (q_0, q_1), (q_0, q_2), (q_1, q_2)
$q_0 q_1, 0$ — on 0 like q_0 on $0 = q_1$ and
q_1 on $0 = q_3$
So $q_0 q_1, 0 = q_1, q_3$ × which does not belong to same 0-equivalence class as q_3 is in C_2 and q_1 is in C_1
Do not need to check for 1 for that combo if 0 does not satisfy
Now, $q_0, q_2, 0 = q_1 q_3$ ×
$q_1 q_2, 0 = q_3, q_3$
$q_1 q_2, 1 = q_4, q_5$ belong to same C_2
So we have two classes from C_1 as
$\{q_0\}$ $\{q_1, q_2\}$

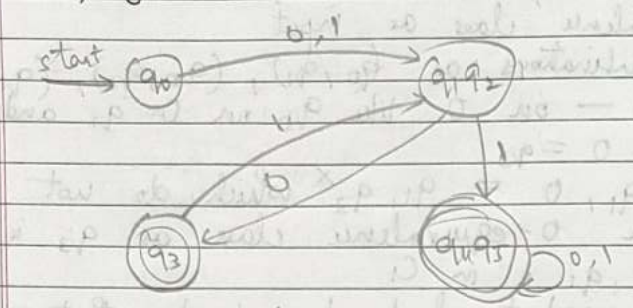
Same way for C_2
$q_3 q_4, 0 = q_3 q_4$
$q_3 q_4, 1 = q_1 q_5$ × not true so discarded
$q_3, q_5, 0$ ×
$q_4, q_5, 0 = q_4 q_5$
$q_4 q_5, 1 = q_5 q_4$ ✓
So C_2 is divided as $\{q_3\}$, $\{q_4, q_5\}$

For 2-equivalence classes,
$\{q_0\}$ we need not check
$\{q_1 q_2\}$ we have to check with 1-equivalence
$q_1 q_2, 0 = q_3 q_5$ ✓
$q_1 q_2, 1 = q_4 q_5$ ✓
So these remain unchanged
For $q_4, q_5, 0 = q_4 q_5$ ✓
$q_4, q_5, 1 = q_5 q_4$ ✓ These also remain unchanged
Here, we stop

Final equivalence classes are taken as new state

Page No.
Date. / /

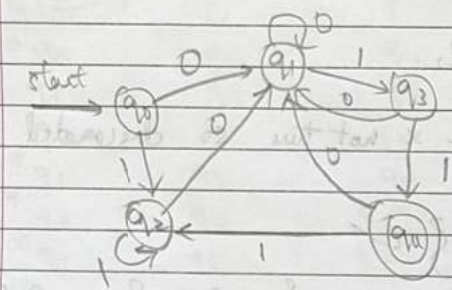
Now, draw DFA as:



Since q_3, q_4 and q_5 were final state in the begining all groups/classes lains any of these are final states

draw DFA using transition table by taking union as before

Q) Minimize the DFA:



Here, there are no dead/unreachable states
Transition table (with and without dead state)

Q \ Σ	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

In q_0, q_1, q_2, q_3 if any one gives q_4 (not in this class) that becomes a separate group as $\{q_3\}$

Page No.
Date. / /

0-equivalence classes:

Class 1: (q_0, q_1, q_2, q_3) C_1

Class 2: (q_4) C_2 → no need to check for this $\{q_0 q_1 q_2\}$

1-equivalence classes

$q_0q_1q_2, 0 = q_1$ ✓ $\quad q_0q_1q_2, 1 = q_2q_3$ ✓

$q_0q_1q_3, 0 = q_1$ ✓ $\quad q_0q_1q_3, 1 = q_2q_3q_4$ ✗

$q_1q_2q_3, 0 = q_1$ ✓ $\quad q_0q_1q_3, 1 = q_3q_2q_4$ ✗

$q_0q_2q_3, 0 = q_1$ ✓ $\quad q_0q_2q_3, 1 = q_0q_4$ ✗

The classes are $\{q_0q_1q_2\}$ $\{q_3\}$ and $\{q_4\}$

2-equivalence classes

~~$q_0q_1q_2, 0$~~

$q_0q_1, 0 = q_1$ ✓ $\quad q_0q_1, 1 = q_2q_3$ ✗

$q_1q_2, 0 = q_1$ ✓ $\quad q_1q_2, 1 = q_2q_3$ ✗

$q_0q_2, 0 = q_1$ ✓ $\quad q_0q_2, 1 = q_2$ ✓

The classes are $\{q_0q_2\}$ $\{q_1\}$ $\{q_3\}$ and $\{q_4\}$

Continue till equivalence classes of the previous step are same as next step.
In exam, do each step for all ~~no~~ terminals (0 and 1) even for single equivalence classes as $q_4, 0 = q_1$ ✓ and $q_4, 1 = q_2$ ✓

3-equivalence class

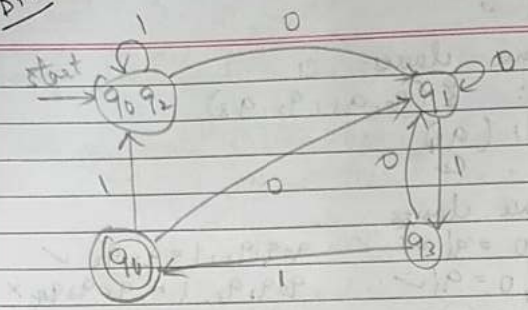
~~$q_0, q_2, 0 =$~~ $q_0, 0 = q_1$ ✓ $\quad q_0, 1 = q_2$ ✓

~~$q_0q_2, 1 =$~~ $q_2, 0 = q_1$ ✓ $\quad q_2, 1 = q_2$ ✓ } Both are same

so no need to split

∴ Final classes are $\{q_0q_2\}$, $\{q_1\}$, $\{q_3\}$, $\{q_4\}$

DFA:



BNF
↳ general form of grammar representation
↳ one of the types/example is CFG

11/12/24

UNIT - II

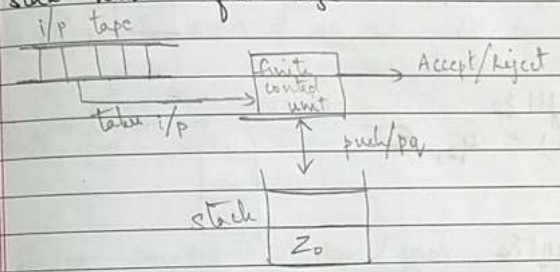
UNIT IV
① for any grammar, write semantic rules
② S-attributed, L-attributed definitions

* Pushdown Automata (PDA)

→ way to implement CFG to design DFA

→ PDA has three components:

1. input tape → called so because input is passed one (by one left to right) (inputs are stored in blocks and passed one by one) check
2. control unit
3. stack with infinite size



→ Formal definition → 7-tuple format
$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

where Q → finite set of states
Σ → input alphabet
tow ← Γ → stack symbol) extra from DFA and NFA
delta ← δ → transition function
q_0 → initial state
Z_0 → initial stack top symbol) extra from DFA and NFA
F → final / accepting state

Q) Construct PDA for the language $\{L = 0^n 1^{2n}, \forall n \geq 1\}$ (language description given)

First, identify all input strings to be accepted

$0^1 1^2$, n=1 — one zero, two ones

i/p strings → {011, 001111 ...}

For n=1, $0^1 1^2$ ⇒ one zero, two ones

Start with transition function as $\delta(q_0, \text{input}, \text{stack})$ → one by one

every input string ends with ε_0 (here, not \$)

if we see 2 ones, it has to match with one zero

Eg: i/p = 001111ε_0

$\delta(q_0, 0, z_0) = q_0, 0z_0$ — push 0 to stack

don't have to do anything

i/p = 001111ε_0

$\delta(q_0, 0, 0) = q_0, 00$ — push 0 to stack

current input, current stack top

i/p = 001111ε_0

$\delta(q_0, 1, 0) = q_1, 0$ — no need to push

when we see first 1, go to a new state and do nothing

i/p = 001111ε_0

$\delta(q_1, 1, 0) = q_2, \varepsilon_0$ — pop current stack top

q_1 on input 1

represents pop one element from stack (top)

i/p = 001111ε_0

$\delta(q_2, 1, 0) = q_1, 0$ — don't do anything

i/p = 001111ε_0

automatically $(q_1, 1, 0)$ goes to q_2, ε_0, pop last 0 on stack and stack becomes

i/p = 001111ε_0

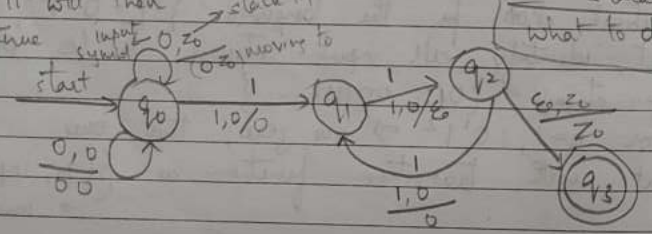
$\delta(q_2, \varepsilon_0, z_0) = q_3, z_0$ — make new state to stop

cannot be made as final state since it will then continue

final state

(no more input symbols to be passed)

current input symbol, stack top, what to do (0 (nothing), ε_0 (pop))



00 push 0 to stack

$0z_0$ → push 0 to stack

Page No.
Date. / /

Q) Construct PDA for $L = 0^n 1^n$, $n \geq 1$

i/p strings = {01, 0011, 000111, ...}

δ(q0, input symbol, stack)

Let input string be : 0011ε_0

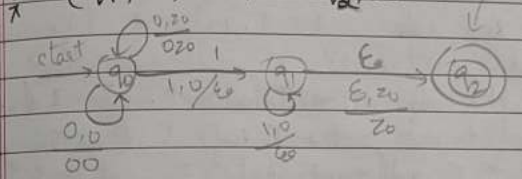
i/p = 0011ε_0 $\delta(q_0, 0, z_0) = q_0, 0z_0$

i/p = 0011ε_0 $\delta(q_0, 0, 0) = q_0, 00$

i/p = 0011ε_0 $\delta(q_0, 1, 0) = q_1, \varepsilon_0$

i/p = 0011ε_0 $\delta(q_1, 1, 0) = q_0, 0$ or q_1, ε_0

i/p = 0011ε_0 $\delta(q_1, \varepsilon_0, z_0) = q_2, z_0$



ε_1

Unit 4

12/11/24 * Syntax Directed Definition (SDD)

If library / extras are attached, how to process

Production	Semantic Rule
$L \to E\$$	
$E \to E+T$	
$E \to T$	
$T \to T*F$	
$T \to F$	
$F \to (E)$	
$F \to$ digit	

why augmented? attaching extra to grammar

only bottom-up parser can parse it

since it is left-recursive (no top-down possible)

$3*5n$

L

E n

3 = F.val

T = F.val * F.val = 3*5

synthesized attribute

T.val = 3

F.val = 5

synthesized

F.val = 3

digit

digit 5

Have to ensure that 15n is given

L should give 15 and n is already there in definition of L

assume first digit as 3

check how 3 is passed to digit → lexical analyser checks if input (3,5) keyboard

3 matches digit regex and then

~~Attributes~~

→ If any grammar is getting value from child / ~~attribute~~ node itself, it is attached as F. att and called synthesized

→ If value is passed from top-down / parent / left-to-right or right-left, it is attached as F. – and called inherited.

For question before

Lexical analyzer passes 3 to digit

At F on left side, F = F.val = 3 (from child, so synthesized

At T on left side, T = T.val = F.val = 3 (synthesized)

At F on right side, F = F.val = 5 (synthesized)

parent ← T.val = T_1.val * T_2.val = F_1.val * F_2.val = 3*5 = 15

child 1 child 2

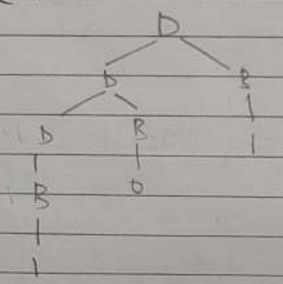
E = T.val = 15 which is then sent to L

→ If a grammar is using only synthesized attribute, it is called S-attributed / Synthesized attributed definition (no inherited attributes used)

Q) Binary to decimal

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

$(101 = 1*2^0 + 0*2^1 + 1*2^2 = 5)$ won't work

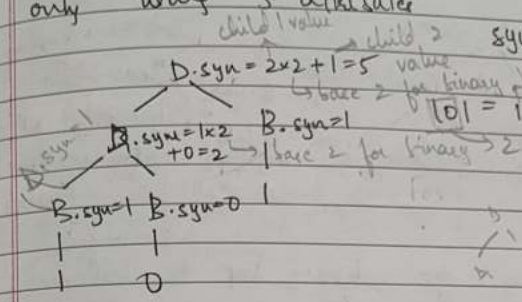


Aim: Get 5 from root using only synthesized attributes only using bottom-up

First, ~~use~~ take number of binary digits and pass to parse tree

Draw ~~for~~ bottom-up parse tree for the conversion only using S-attributes

syn = synthesized attributed



|0| = 1×2+0 = 2×2+1=5

2 so base=2

D_1 → child

D → parent

Production	Semantic Rules
$D \to D_1 B$	$D.syn = D_1.syn \times 2 + B.syn$
$D \to B$	$D.syn = B.syn$
$B \to 0$	$B.syn = 0$
$B \to 1$	$B.syn = 1$

The above is called SDD for the given question

Q) CFG given or do CFG, draw parse tree for bottom-up S-attributed definition and then write semantic rules

Given $D \to DB \mid B$, $B \to 0 \mid 1$

→ To do the same for top down, eliminate left-recursion as

$D \to BD'$

$D' \to BD' \mid \varepsilon$

$B \to 0 \mid 1$

$D_1' = D'$

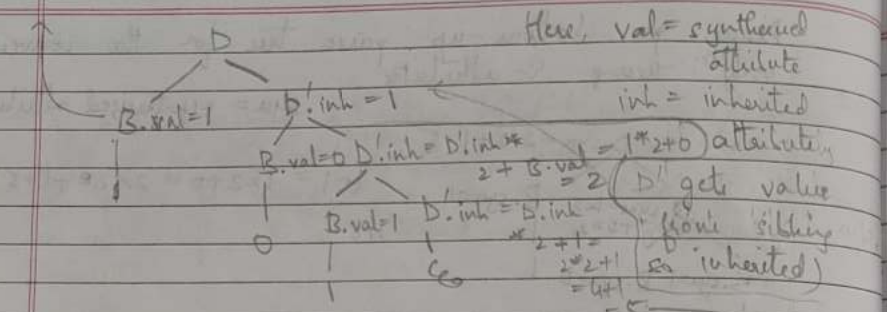
child parent

Inherited attributes

Production	Semantic Rules
$D \to BD'$	$D'.inh = B.val$, $D.val = D'.syn$
$D' \to BD_1'$	$D_1'.inh = D'.inh * 2 + B.val$, $D'.syn = D_1'.syn$
$D' \to \varepsilon$	$D'.syn = D'.inh$
$B \to 1$	$B.val = 1$
$B \to 0$	$B.val = 0$

Let input = 101, draw parse tree

this should be passed from left to right to D'

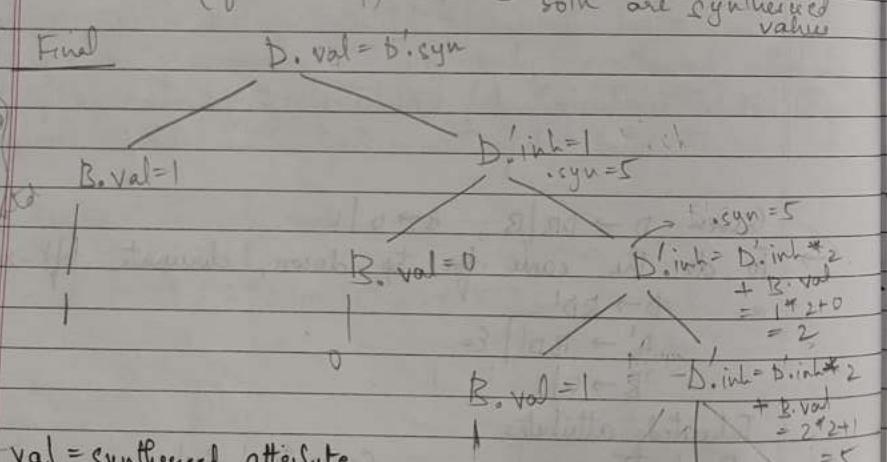


Here, val = synthesized attribute
inh = inherited attribute
(D' gets value from sibling so inherited)

In the above go from top to bottom

final value should come from root, so synthesized attribute is used to pass result to root so used syn (for passing) and val (for holding) → both are synthesized value

Final



annotated and L-attributed parse tree

val = synthesized attribute (for holding)
syn = synthesized attribute (for passing)
inh = inherited attribute

Annotated parse tree — modified parse tree — can be top down or bottom up

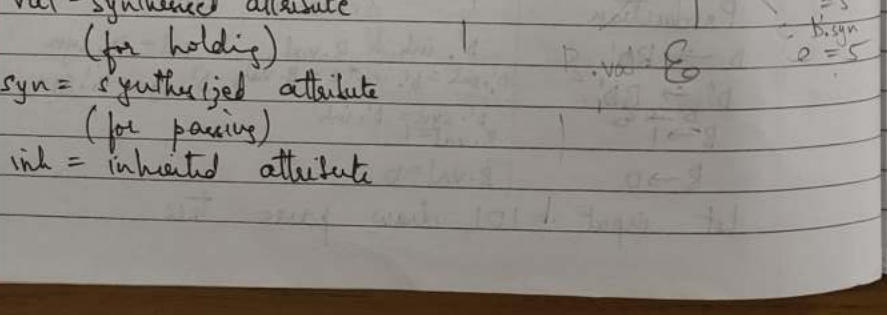
L-attributed parse tree — can have only inherited or both inherited and synthesized attributes

Q) Draw annotated parse tree for 3*5 using

$T \to FT'$
$T' \to *FT'$
$T' \to \varepsilon$
$F \to digit$

Use top-down approach

annotated / L-attributed / modified parse tree



for passing left to right

Production	Semantic Rule
$T \to FT'$	~~T.val =~~ (T'.inh = F.val) T'.val = T'.syn
$T' \to *FT'$	T'_1.inh = T'.inh * F.val = T'.syn = T'_1.syn
$T' \to \varepsilon$	T'.syn = T'.inh
$F \to digit$	~~F.val = 5 F.val~~ ~~F.val=5, F.val=3~~ F.val = digit.lex-value. (can be 3 or 5)

Q) Construct S-attributed definition for input = 101 for ~~D → BD|B, B → 0|1~~ $D \to BD'$, $D' \to BD' | \varepsilon$, $B \to 0|1$

Do using $101 = 1 * 2^0 + 0 * 2^1 + 1 * 2^2$

Page No.
Date. / /

5 1101

$D.val = D_1'.val + B.val * 2^{D_1'.pos} = 4+1 = 5$

$= 1 + 1 * 2^2$

$.pos = D_1'.pos + 1 = 1+1 = 2$

Here, val = synthesized variable

$B.val = 1$

$D'.val = D_1'.val + B.val * 2^{D_1'.pos}$

$= 1 + 0 * 2 = 1$

$.pos = D_1'.pos + 1 = 1$

$B.val = 0$

$D'.val = D_1'.val + B.val * 2^{D_1'.pos}$

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

$B.val = 1$

$D'.val = 0$

$.pos = 0$

ε_0

pos = synthesized attribute for $2^{0, 1, 2}$

For each D', $D_1'.pos$ is incremented for next iteration, after calculating $D'.val$ for that step

For D' → parent, D' → child: while writing, D_1' is child of D'

Production	Semantic Rules
$D \to BD'$	$D.val = D_1'.val + B.val * 2^{D'.pos}$
$D' \to BD'$	$D'.val = D_1'.val + B.val * 2^{D_1'.pos}$ $D'.pos = D_1'.pos + 1$
$D' \to \varepsilon_0$	$D'.val = 0$ $D'.pos = 0$
$B \to 1$	$B.val = 1$ (given in grammar, so written)
$B \to 0$	$B.val = 0$

16/12/24* Definitions

1. S-attributed — if there are only synthesized attributes in rules
2. L-attributed — if there are both synthesized and inherited attributes or only synthesized (S-attributed) or only inherited
   → LR — left to right translation of rules

Page No.
Date. / /

Q) Is the rule valid? Is it an L-attributed definition?

A → B C D ← C.inh — from A or B or D

L-attributed only when translation is from left to right so D to C is not L-attributed

Production	Semantic Rules
$A \to BC$	$A.s = B.i$ $B.i = f(C.c, A.a)$ b, a, c = lexical values

only left to right is L-attributed

A.s → B.i, C

But, this is not L-attributed since B.i is getting from A and C (right to left)

This is valid, & just not called L-attributed.

Q) Given $D \to BD'$, $D' \to BD' \mid \varepsilon_0$, $B \to 0$, $B \to 1$, draw annotated parse tree (APT) for input 1101 and make it L-attributed.

$D.syn = D_1.val = 13$

$B.s = 1$

$D'.i = B.s = 1$
$.val = 13$

$B.s = 1$

$D_1'.i = D'.i * 2 + B.s = 1*2+1 = 3$
$.val = 13$

$B.s = 0$

$.val = 13$
$D_1'.i = D'.i * 2 + B.s = 3*2+0 = 6$

$B.s = 1$

$D_1'.i = D'.i * 2 + B.s = 6*2+1 = 13$
$.val = 13$

ε_0

D_1 = child
i = inherited attribute
s = synthesized attribute

Compiler sees the symbol table when there is identifier. Symbol table is external so called side effect.

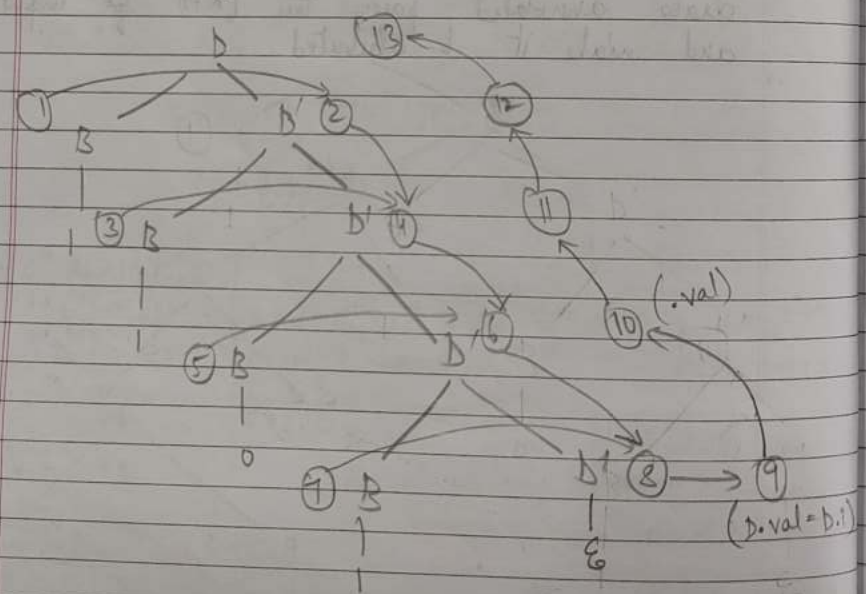
Production	Semantic Rules
$D \to BD'$	$D.syn = D_1.val$; $D'.i = B.s$
$D' \to BD_1'$	$D'.val = D_1'.val$; $D_1'.i = D'.i * 2 + B.s$
$D' \to \epsilon$	$D_1'.i = D'.i * 2 + B.s$; $D'.val = D'.inh$
$B \to 0$	$B.s = 0$
$B \to 1$	$B.s = 1$

* Dependency Graph

Tokens / terminals — at leaf nodes

1st action => number the synthesized attributes (first computation)

Follow order of computations and keep numbering (in increasing order)



The above is called a dependency graph

Need to give set of rules to control side effects

Whenever id is seen, check in symbol table for its presence

AddType function to add type to symbol table

↳ done by backend of compiler — called controlled side effect

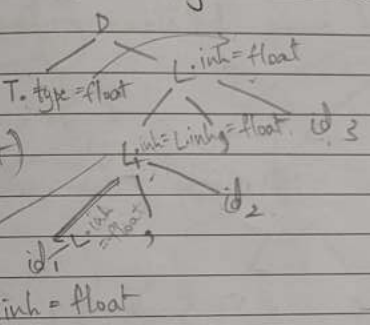
17/12/24* Semantic Rules with Controlled Side Effects → external

For float id1, id2, , here we ~~also~~ check how much memory is needed for each, etc. involvement has to be related

→ id.entry — a lexical value that points to a symbol table object

id val type a 3 int

Eg: input to be analyzed — float id1, id2, id3;



In the symbol table, we check if all the variables are present

here, symbol table is external data

type = synthesized attribute — can use syn also

Production	Semantic Rules
$D \to TL$	$L.inh = T.type$
$T \to int$	$T.type = int$
$T \to float$	$T.type = float$
$L \to L_1, id$	$L_1.inh = L.inh$; addType(id.entry, L.inh)
$L \to id$	addType(id.entry, L.inh)

Symbol table is external data set so we can expect side effect. Hence, we use addType( ) to ensure that the type of variables is set in the symbol table

meaningful function

→ this is a controlled side effect

here, aim is to make symbol table

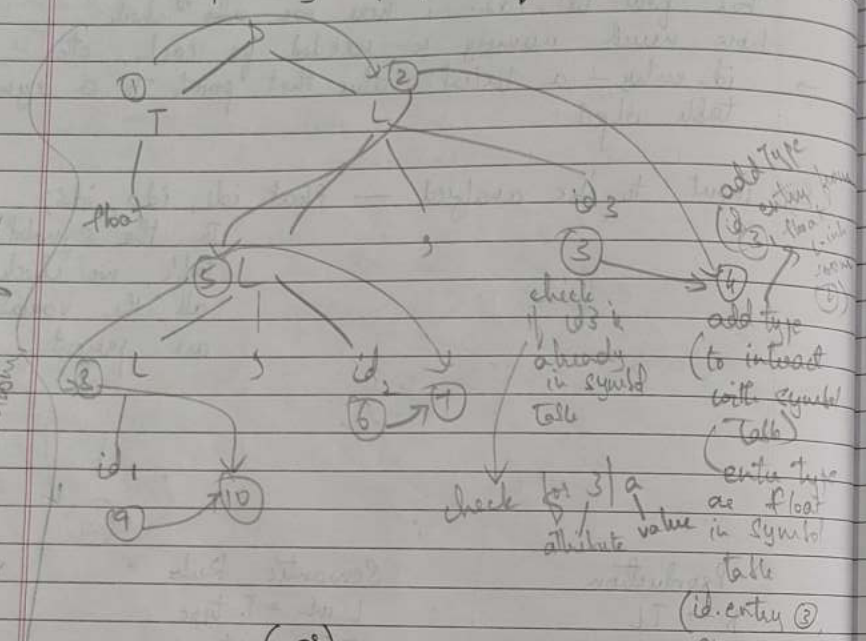
Symbol Table

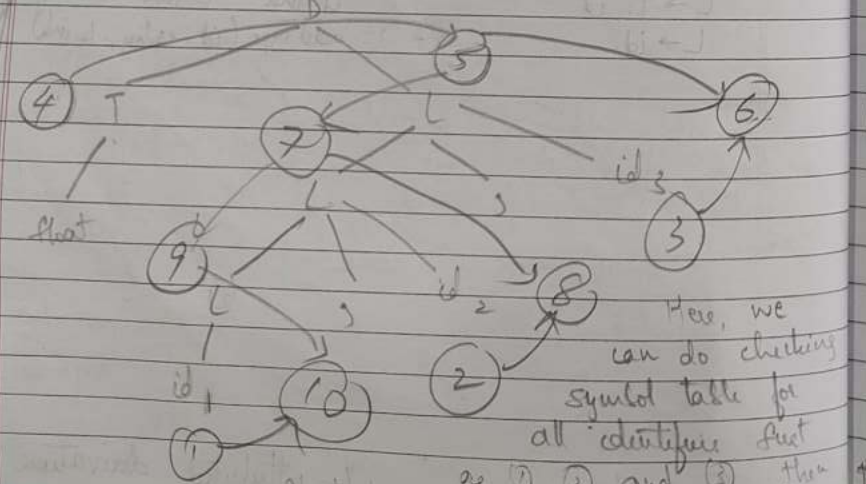
		Type
3	a	float
2	b	A

The above is an L-attributed derivation

Q) Is there any side effect? Give ways to deal with it (write annotated parse tree (APT))
→ grammar can also be given

→ Draw dependency graph for previous question



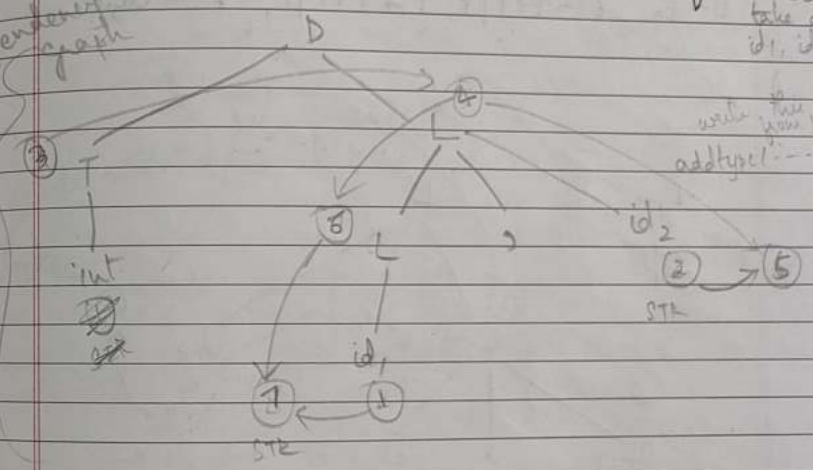


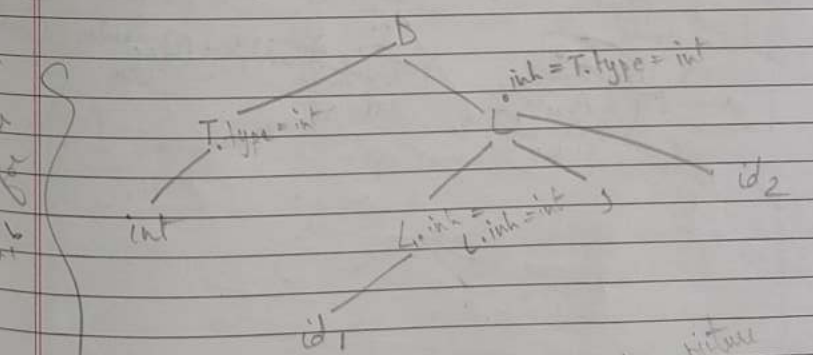
Here, we can do checking symbol table for all identifiers first as ①, ② and ③ then float is passed through ④, and addtype through ⑤... etc.

Compiler has to refer symbol table to know details for assigning value
First, ST contains id, | a, in this phase.
type is added

Q) Give topological sort order for evaluating the dependency graph.
→ all nodes in order as ①, ②, ③, ④, ⑤...
as 1,2,3,4,5,6,7,8,9,10
Topological sort is the order of evaluation given to parser for evaluation

Q) Do ~~synt~~ annotated parse tree (ATP) for int a,b
take as id1, id2





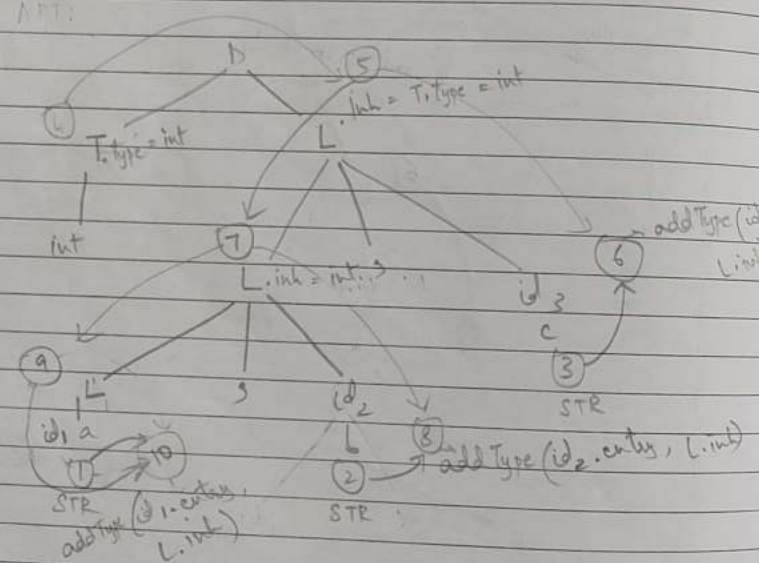
→ check picture in phone

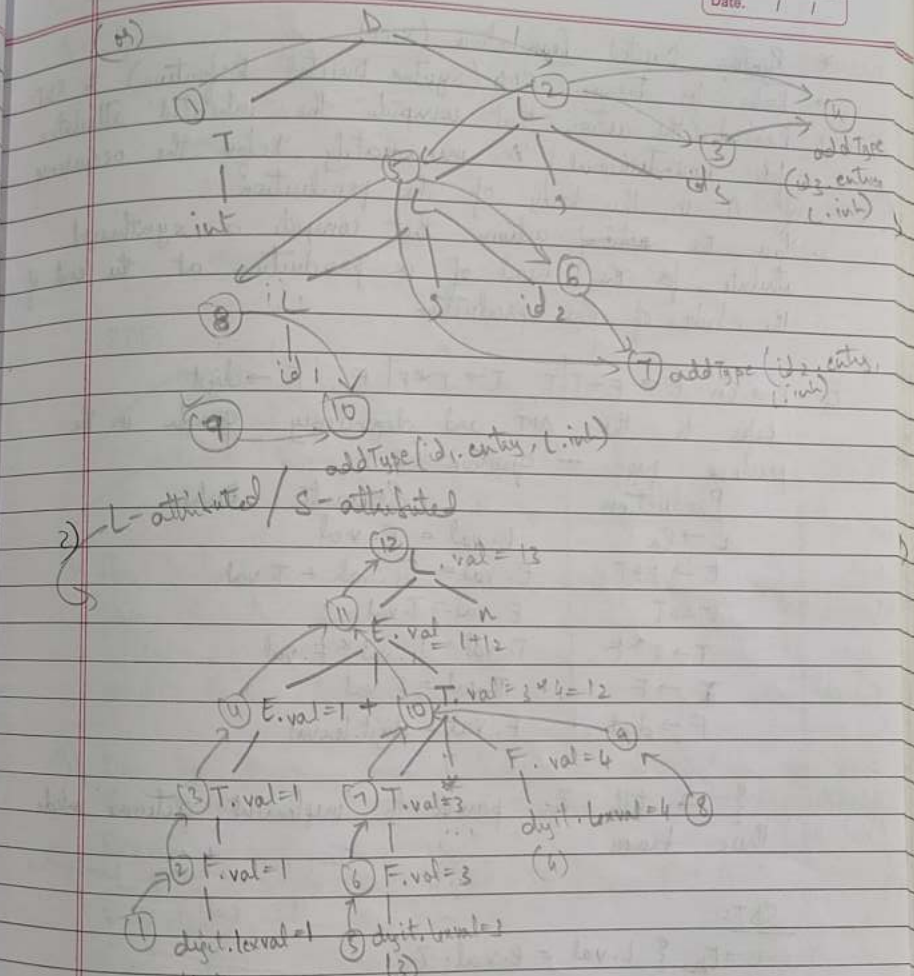
Parser checks symbol table for identifier.
If it exists, it uses addType() to add type of variable in the symbol table.
This is called controlled side effects.

18/12/24 1) Construct dependency graph and APT for "int a, b, c" considering semantic rules for simple type declarations.

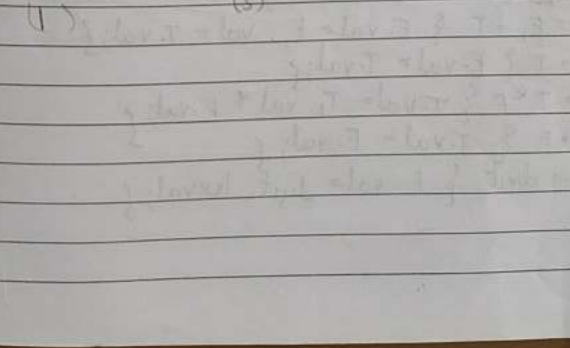
2) Draw APT for 1+3*4n for
$L \to En$, $E \to E+T \mid T$, $T \to T*F \mid F$, $F \to digit$

1) APT:





2) L-attributed / S-attributed



Page No.
Date. / /

* Syntax Directed Translation (SDT)

→ Rules for Turning SDD (Syntax Directed Definition) to SDT:

1. Embed the action that compute the inherited attributes for non-Terminal 'A' immediately before the occurrence of A in the body of the production.
2. Place the ~~actions~~ actions that compute a synthesized attribute for the head of a production at the end of the body of the production.

Eg: $L \to En$, $E \to E+T \mid T$, $T \to T*F \mid F$, $F \to digit$

Refer to the APT and dependency graph in the previous page — question (2)

Production	Semantic Rules
$L \to En$	$L.val = E.val$
$E \to E+T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T*F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to digit$	$F.val = digit.lexval$

do not use 'lex' for lexval ↳ anything else is accepted (not lex)

{ → tells the parser to implement actions inside these braces

synthesized attributes always computed at the end

SDT:

$L \to En$ { L.val = E.val; }
$E \to E_1 + T$ { E.val = E_1.val + T.val; }
$E \to T$ { E.val = T.val; }
$T \to T*F$ { T.val = T_1.val * F.val; }
$T \to F$ { T.val = F.val; }
$F \to digit$ { F.val = digit.lexval; }

since for synthesized attributes

inherited — from parent or sibling

Page No.
Date. / /

Eg: Given $D \to TL$, $L \to L_1, id$, $L \to id$, $T \to int$, do SDD → SDT

Production	Semantic Rules
$D \to TL$	$L.inh = T.type$
$T \to int$	$T.type = int$
$L \to L_1, id$	$L_1.inh = L.inh$ addType(id.entry, L.inh)
$L \to id$	addType(id.entry, L.inh)

for int a, id.entry = a

SDT:

→ after calculating L, you pass it to L

$D \to T$ { L.inh = T.type; } L
$T \to int$ { T.type = int; }
$L \to$ { L_1.inh = L.inh; } L_1, id { addType(id.entry, L.inh); }
$L \to id$ { addType(id.entry, L.inh); }

for inherited attributes, after calculating L value, we pass it to L

Here, after calculating L_1 value, we pass to L_1 and then calculate id and pass to addType()

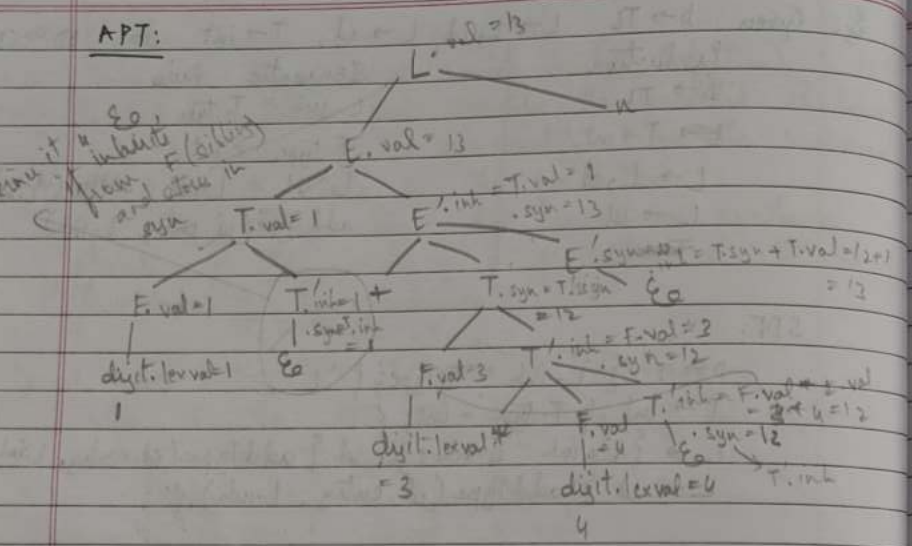
Q) Do top down parser stack for $1+3*4n$ for the grammar $L \to En$, $E \to E+T \mid T$, $T \to T*F \mid F$, $F \to digit$

Removing left recursion,

$L \to En$, $E \to TE'$, $E' \to +TE' \mid \varepsilon$, $T \to FT'$, $T' \to *FT' \mid \varepsilon$, $F \to digit$

First draw APT (next page)

APT:



Production	Semantic Rules

SDT:

19/12/24 * Applications of SDTs

→ SDT of array type

Production	Semantic Rules
$T \to BC$	
$B \to int$	
$B \to float$	
$C \to [num]\,C$	
$C \to \varepsilon$	

input : int a [10][10]; → not required

cannot be parsed successfully by the above given grammar since identifier is not defined for 'a'

Q) How can above CFG be changed to accept int a[10][10]

Changed to add $T \to B\,id\,C$ ';' → not required

→ can cause side effect

Symbol Table

a	datatype int	array type 10×10×4 (400 bytes of memory to be allocated)

Correct grammar for array type:

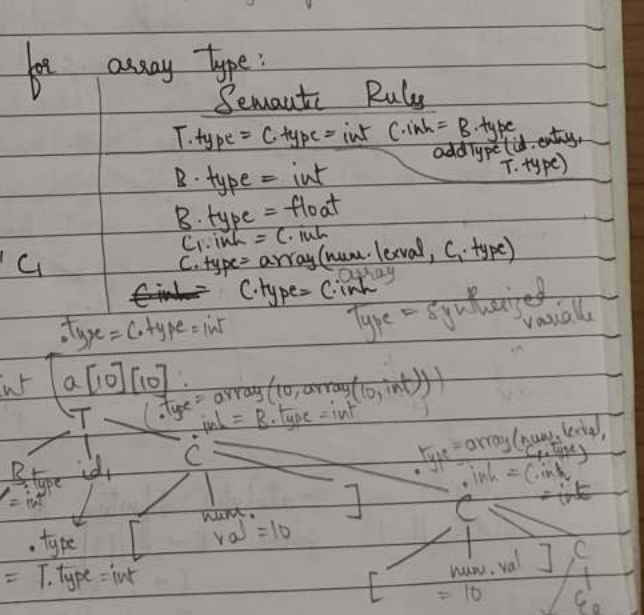
Production	Semantic Rules
$T \to B\,id\,C$	$T.type = C.type = int$ $C.inh = B.type$ addtype(id.entry, T.type)
$B \to int$	$B.type = int$
$B \to float$	$B.type = float$
$C \to$ '[' num ']' C_1	$C_1.inh = C.inh$ $C.type = array(num.lexval, C_1.type)$
$C \to \varepsilon$	~~C.inh =~~ $C.type = C.inh$

not attributed, get type from B → L-attributed

array

Type = synthesized variable

Parse tree for int a[10][10]:



parse from left to right

C.inh = int should be passed down till ε is encountered

Specify array using "array(dimension value, subtype from where)" array(10, int)

Page No.
Date. / /

→ **Elimination of Left Recursion from SDTs**

General formula:

$A \to A_1 Y \{ A.a = g(A_1.a, Y.y) \}$ X and Y are terminal/tokens, A is non-terminal

$A \to X \{ A.a = f(X.a) \}$

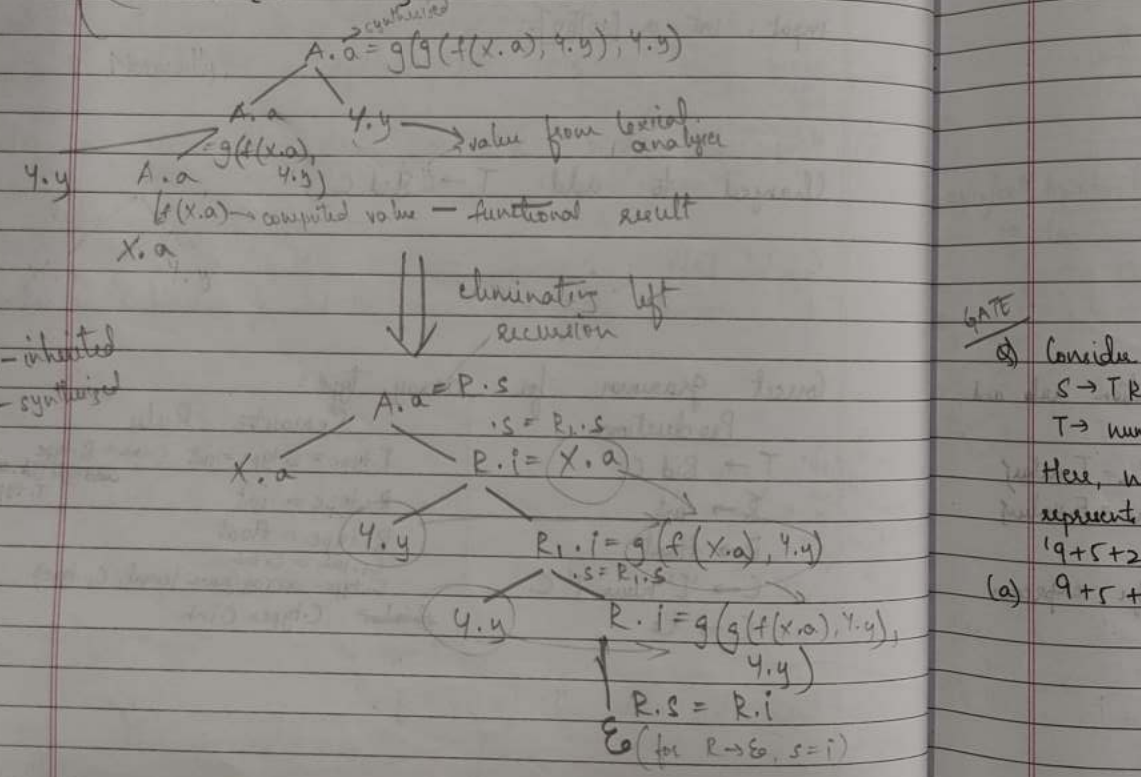
learn, will be asked to write

⇒ $(A \to XR,\ R \to YR \mid \varepsilon_0)$ same rule for eliminating left recursion

$A \to XR \quad \{ R.i = f(X.a);\ A.a = R.s \}$

$R \to YR \quad \{ R_1.a = g(R.i, Y.y);\ R.s = R_1.s \}$

$R \to \varepsilon_0 \quad \{ R.s = R.i \}$



20/12/24 Q) Write an S-attributed definition to compute the expression (&&, ||, !) given

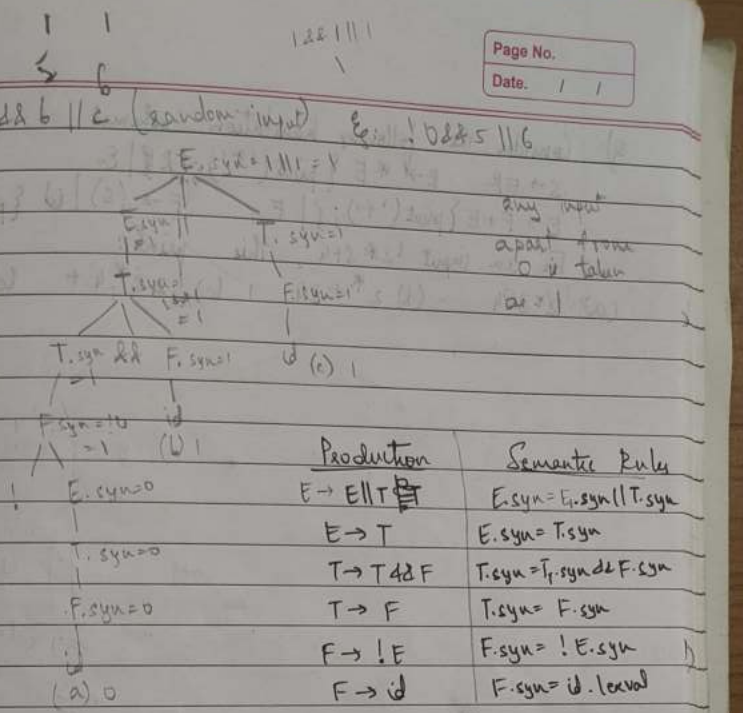
$E \to E \| T \mid T$

$T \to T \&\& F \mid F$

$F \to !E \mid id$

Page No.
Date. / /

Taking !0 && 5 || 6 (random input) ε: !0&&5||6



Production	Semantic Rules
$E \to E \| T$	$E.syn = E_1.syn \| T.syn$
$E \to T$	$E.syn = T.syn$
$T \to T \&\& F$	$T.syn = T_1.syn \&\& F.syn$
$T \to F$	$T.syn = F.syn$
$F \to !E$	$F.syn = !E.syn$
$F \to id$	$F.syn = id.lexval$

GATE

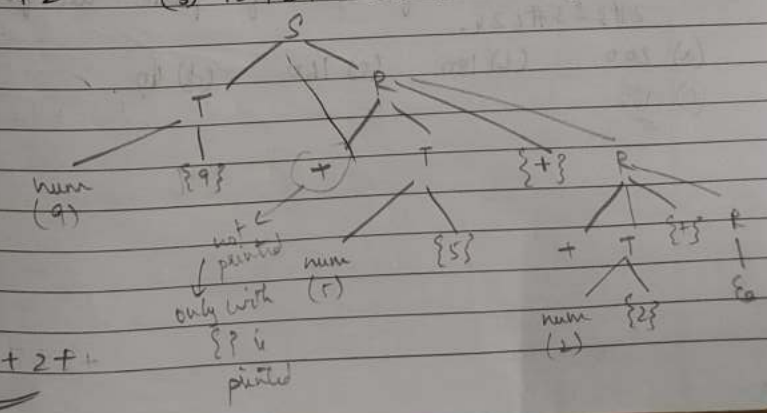
Q) Consider the translation scheme:

$S \to TR \qquad R \to +T \{ print('+'); \} R \mid \varepsilon_0$

$T \to num \{ print(num.val); \}$

Here, num is a token that represent an integer and num.val represent the corresponding integer value. For an input string '9+5+2', this translation scheme prints:

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



(b) 95+2+

these kinds of questions will be asked

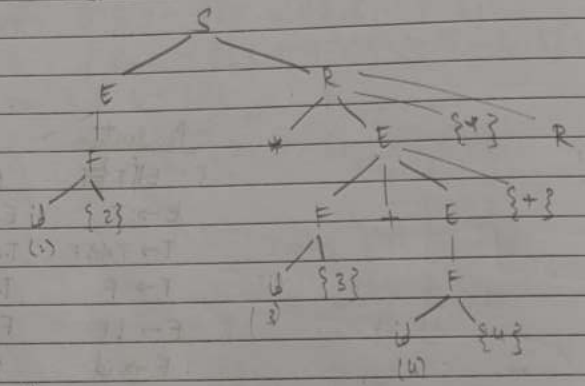
Q) Consider the following translation scheme:

$S \to ER \quad R \to *E\{print('*');\}\{R\} \mid \varepsilon$

$E \to F+E\{print('+');\} \mid F \qquad F \to (S) \mid id\ \{print(id.value);\}$

For an input '2*3+4', this prints:

(a) 2*3+4 (b) 2*+34 (c) 23*4+ (d) 234+*



d) 234 + *

Q) Consider the grammar with the following translation rules and E as the start symbol.

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

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

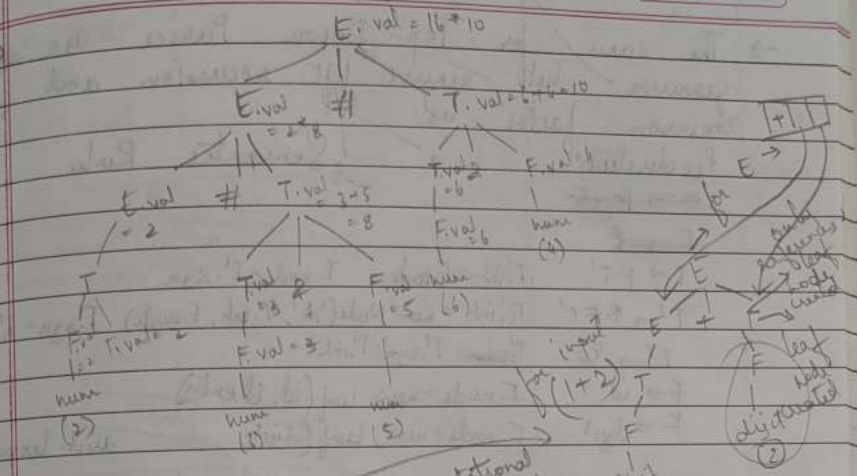
$F \to num\ \{F.value = num.value\}$

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

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

(c) 160

translation structure – using postorder traversal



23/12/24* Applications of SDT

→ Construction of Syntax tree from Bottom Up Parser (BUP)

Production	Semantic Rule
$E \to E_1 + T$	E.node = new Node('+', E_1.node, T.node)
$E \to T$	E.node = T.node
$T \to T_1 * F$	T.node = new Node('x', T_1.node, F.node)
$T \to F$	T.node = F.node
$F \to id$	F.node = new Leaf (id, id.entry)
$F \to digit$	F.node = new Leaf (digit, digit.lexval)

since it's bottom up, we do from leaf node

computational result

not leaf but left node

right node

synthesized

only assigning the node

creating node

In symbol table

Input – 1 + 2

this is how each token is represented

name of token

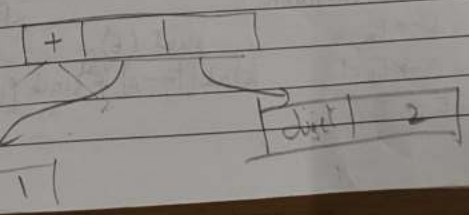
either direct / attribute value

→ value from lexical analyzer

For operator, operator node:

operator	left node	right node

For 1 + 2



node represents synthesized attribute

$T \to FT'$
$T' \to *FT'$
$T' \to \varepsilon$
$F \to id \mid digit$

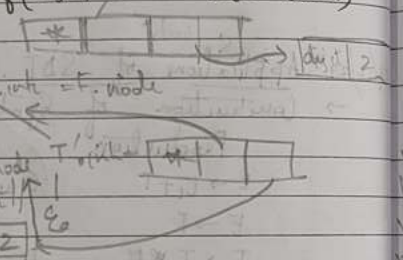
→ The same grammar for Top-Down Parser — use ~~parser~~ but remove left recursion and left common factor as:

Production	Semantic Rules
~~$E \to E_1 + T$~~	
~~$E \to T$~~	
$T \to FT'$	$T'.inh = F.node$ $\quad T.node = T'.syn$
$T' \to *FT'$	$T_1'.inh = new\ Node('*', T'.inh, F.node)$ $\quad T'.syn = T_1'.syn$
$T' \to \varepsilon$	$T'.syn = T'.inh$
$F \to id$	$F.node = new\ Leaf(id, id.entry)$
$F \to digit$	$F.node = new\ Leaf(digit, digit.lexval)$

similar but syn shows passing and node shows holding value

digit | 1

Input: 1 * 2



① F.node passed from left to right

② T'.inh = F.node

SDT to Top Down / Bottom Up

Q) Can be asked to do same for '-', '/' → same as for '*'
same as for '+'

→ Postfix SDTs → or Polish → only for bottom-up parser stacks
→ means in which type of SDT, we write all rules at the end of production — like $S \to ET\{\ ;\}$
this happens only in S-attributed definitions

→ Parser Stack Implementation

Production	Semantic Rules
$L \to En$	print (E) → when n is on stack top
$E \to E_1 + T$	$stack[top-2].val = stack[top].val + stack[top-2].val$; $top = top - 2$
$E \to T$	

Yacc — tool for implementing parser
↳ Yet another Compiler Compiler

Production
$T \to T_1 * F$
$T \to F$
$F \to digit$

For 1+2

top	T
top − 1	+
top − 2	$E_1 = 3$

for $E \to E + T$
stack

use Yacc to get these values

First assume there is stack and for $E \to E + T$ get stack[top − 2].val + stack[top.val] and stored in stack[top − 2].val

Now, E+T, we do not need E_1 + and so stack becomes top−2 as

	T			T
top − 1	+	=		+
top − 2	E_1			E_1
				stack

For 1+2+3
$E + T$
top−2 top−1 top

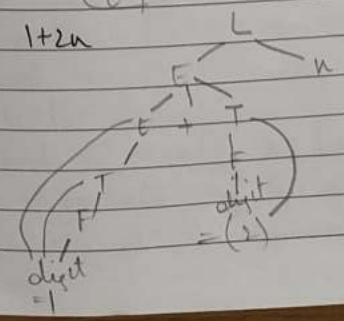
This is how YACC produces bottom up parser

Whenever there is n we print final value

For grammar $E \to T_1 * F$ → items of grammar pushed into stack in reverse order

F
*
T_1
E → already there

For 1+2n



n	top
E = 3	top − 1

end of input, print [top − 1] value

T = 2	top
+	top − 1
1 + E = 3	top − 2

result calculated and stored in E top

result

26/12/24) Try octal to decimal conversion

UNIT - V

30/12/24

* Intermediate Code Generator (ICG)

Variant of syntax tree - DAG (directed acyclic graph)

G: $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$ Input expression: a+b+(a+b)

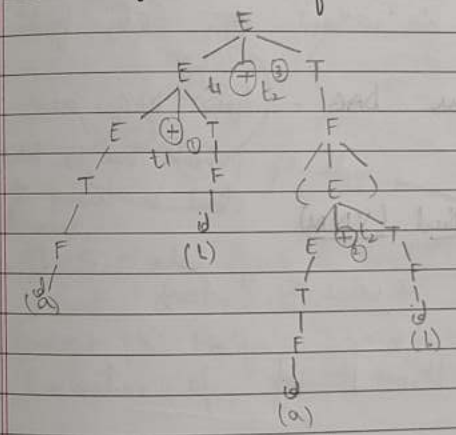
→ Three Address Code (3AC)

generated from ICG - non-optimized ICG → 3AC

→ Local Optimization

→ Common sub-expression elimination

→ Draw syntax tree for above input = a+b+(a+b)



Generation of 3AC from above tree,

3AC

() - only to check if brackets are matching, no use in 3AC

$t_1 = a+b$

$t_2 = a+b$

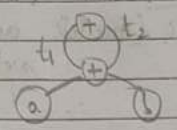
$t_3 = t_1 + t_2$

t_1 and t_2 = a+b, so local optimization done using DAG - evaluate left to right

For a+b+(a+b)

Page No.
Date. / /

Construct leaf node for a, then b, then + and then goes to right side, sees that a+b is already constructed and ignores, then constructs + for final computation



Optimized:
$t_1 = a+b$
$t_2 = t_1 + t_1$

Local Optimization — if node have already been created, re-use it.

→ Intermediate Code Generator (ICG) cannot do optimization and does not use DAG.

Q) Construct 3AC — do without DAG, only from syntax tree

Q) Construct 3AC from DAG — draw DAG and then get 3AC

* Value Number Method (VNM) → nodes of syntax tree or DAG stored in array of records

Leaf node [|] — attribute; where to get value from

Operational node [operator | left node | right node]

→ VNM for above example

①	id	a	
②	id	b	
③	+	①	②
④	+	③	③

draw using leaf nodes and operational nodes

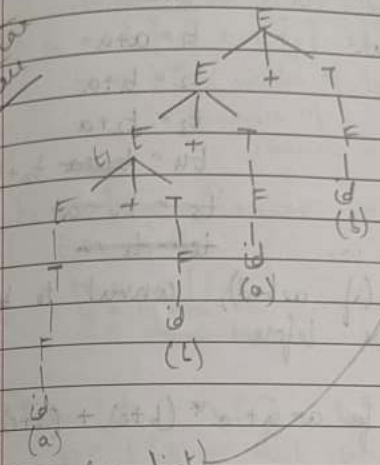
→ check phone for picture (Mere)

Page No.
Date. / /

Q) For a+b+a+b, draw syntax tree, DAG and 3AC.

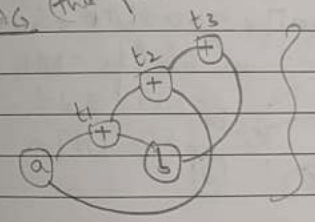
grammar is same as before

Syntax tree



then this 3AC
$t_1 = a+b$
$t_2 = t_1 + a$
$t_3 = t_2 + b$

DAG (this first)



don't mix up left and right for this

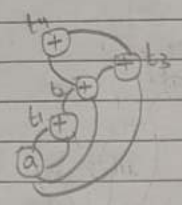
→ Shortcut to check if correct:
Input: a+b+a+b
Step 1: Convert to Postfix: ab+a+b+
Step 2: Evaluation of postfix = DAG

→ Steps of Construction of DAG:
1. p_1 = new leaf (id, entry.a);
2. p_2 = new Leaf (id, b); left, right
3. p_3 = new Node ('+', p_1, p_2);
4. p_4 = p_1
5. p_5 = new Node ('+', p_3, p_4) or p_1 since $p_4 = p_1$
6. $p_6 = p_2$
7. p_7 = new Node ('+', p_5, p_6) or p_2 since $p_6 = p_2$

(→ check if any of these are same as previously created. If yes, reference that only

Q) Construct DAG and 3AC for: $\underbrace{a+a+a}_{\text{left child}} + \underbrace{(a+a+a+a)}_{\text{right child}}$

DAG:



3AC

$t_1 = a+a$

$t_2 = t_1+a$

$t_3 = t_2+a$

$t_4 = \cancel{t_3+a}\ t_2+t_3$

$\cancel{t_5 = t_4+a}$

$\cancel{t_6 = t_5+a}$

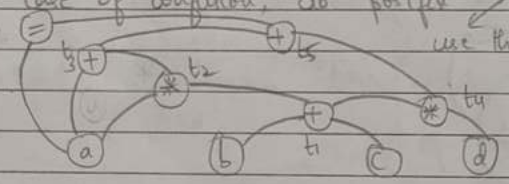
can be asked → For steps of construction (if asked), convert to ~~DAG~~ postfix and write steps as before

Q) Construct DAG and 3AC for $a = a + a * (b+c) + (b+c) * d$

* has higher precedence

In case of confusion, do postfix = $a\,a\,b\,c\,+\,*\,+\,b\,c\,+\,*\,d\,*\,+\,a\,=$ ← use this for DAG

DAG



write left and right nodes as it, don't swap

3AC

$t_1 = b+c$

$t_2 = a*t_1$

$t_3 = a+t_2$

$t_4 = t_1*d$

$t_5 = t_3+t_4$

$a = t_5$

if – conditional, else – unconditional } GOTO

→ For ~~validation of~~ evaluation of conditional statements, if and goto used.

→ param is used ~~for~~ if there is an argument/parameter

→ y = call p, n — indicate function call

p — name of function

n — number of parameters

* Three – Address Code Translation

Suppose int b[3], c[4], a[2]

Eg: ① $a = b[i] + c[j]$ translated as:

$t_1 = i*4$ (number of bytes occupied by one integer) (number of elements in b array)

b array (3, int)

c array (4, int)

$t_2 = b[t_1]$ // mean $b+t_1$ (getting index t_1 of array b)

$t_3 = j*4$

$t_4 = c[t_3]$

$t_5 = t_2+t_4$

$a = \cancel{t_2+t_4}\ t_5$

there are without optimization

Q) ② Translate $a[i] = b*c + b*d$

$t_1 = i*4$ → first compute index assuming a is an integer array

$t_2 = b*c$

$t_3 = b*d$

$t_4 = t_2+t_3$

$a[t_1] = t_4$

→ Quadruple – with four fields – op, arg_1, arg_2, result

(op: operator; arg_1: argument 1; arg_2: argument 2)

For ①

	op	arg_1	arg_2	result
①	*	i	4	t_1
②	= [ ]	b	t_1	t_2
③	*	j	4	t_3
④	= [ ]	c	t_3	t_4
⑤	+	t_2	t_4	t_5
⑥	=	t_5		a
	-[ ]	t_1	t_4	a

can do this for assignment to array index

Q) ③ $x = f(y+1) + 2$ ← function call

3AC

after t_2 call, can also mention,
// return t_2

all parameters must be specified one by one separately

$t_1 = y+1$
param t_1
t_2 = call f, 1 → function name, parameters number
automatically returns also using
$t_3 = t_2 + 2$
$x = t_3$

→ Representing the above in Quadruple

	op	arg_1	arg_2	result
①	+	y	1	t_1
②	param	t_1		
③	call	f	1	t_2
	return	t_2		
④	+	t_2	2	t_3
⑤	=	t_3		x

→ Triple form – has 3 fields – op, arg_1, arg_2 only
→ Indirect triple – contains references / pointers to values in hash table

see ppt for above

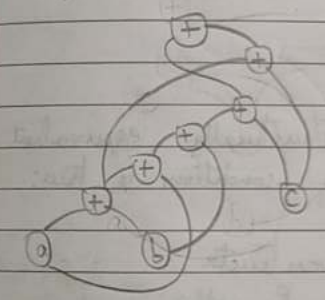
→ Computing types and their widths – SDT of array types – see picture (Mccc)

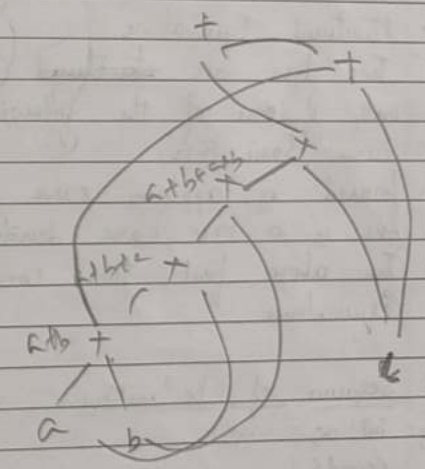
Q) Write 3AC for:
1. val = x (min+1, min−1, 1)
2. x = f (a [i]) // a is array of (3, int)
3. x = a[i]+c

1.	2.	3.
t_1 = min+1	$t_1 = i * 4$	$t_1 = i*4$
t_2 = min−1	$t_2 = a[t_1]$	$t_2 = a[t_1]$
param t_1	~~t_3 = call f,~~	$t_3 = t_2 + c$
param t_2	param t_2	
param 1	t_3 = call f, 1	
t_3 = call x, 3	& x = t_3	
val = t_3		

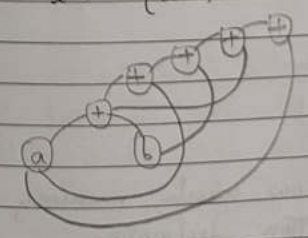
Q) Construct DAG for:
1. (a+b) + a + b+c + (a+b+c)
2. a+b+ a+b+ (a+b) + a

1. ~~Postfix: ab+ ab+c+ ab+c+ +~~





2. a+b + a+b + (a+b) + a



Page No.
Date. / /

1/1/25 * <u>Type and Declaration</u>

→ Type Expressions can be any of the ones below

1. basic type (int, float, double)
2. type name
3. key specifiers (array, record, etc.) → short declaration
4. type constructor for function types $s \to t$ (function from type s to type t) → t being called by s

→ Type Equivalence

If two type expressions are equal, then return type, else error.

→ Structural Equivalence

Two types are ~~structural~~ structurally equivalent if and only if one of the following conditions is true:

1. same basic type
2. formed by applying same constructor
3. (one is a type name denotes the other) one of the type denotes the other
→ The above first two conditions lead to Name Equivalence

function overloading

* <u>Sequence of Declaration</u>

Eg: int a; – 4 bytes
record {
int b; – 4 bytes
float c; – 8 bytes } 12 bytes
} p; } 16 bytes

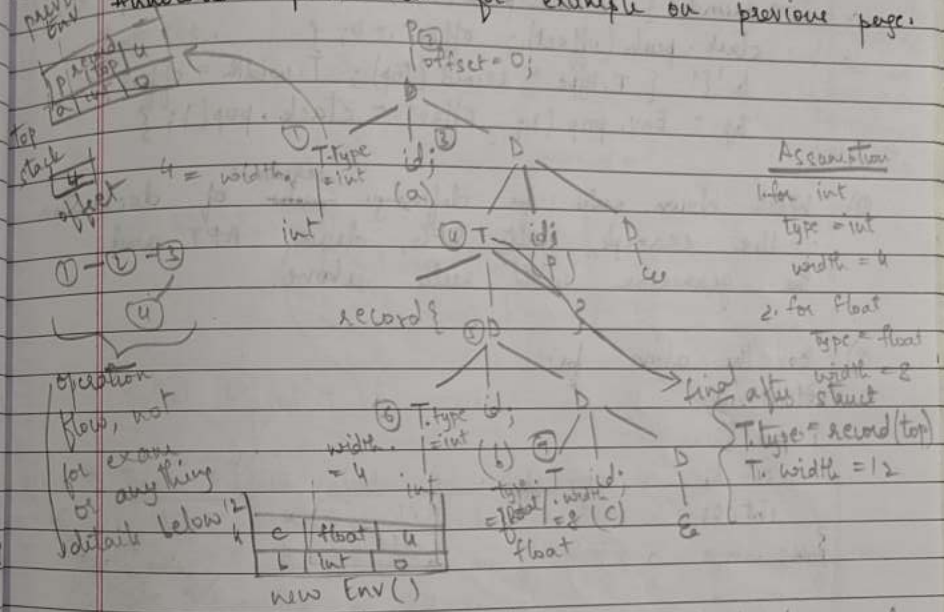
How to tell parser how much memory is required.
"define" used for function declaration

offset – before declaration used, which is initialized to 0

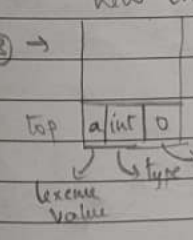
Page No.
Date. / /

G: $D \to T\ id;\ D_1 \mid \varepsilon$
$T \to B\ C \mid record\ '\{'\ D\ '\}'$
$B \to int \mid float$
$C \to [num]\ C_1 \mid \varepsilon$

Annotated parse tree for example on previous page.



At ③ →



take stack pointed to by top and use keyword called put
Here also calculate next offset = current offset + width of lexeme value type

Before seeing declaration for structure (struct), here, record, have to push top to Env stack / Environment variable and take another new Env (new symbol table created)
After ⑧, add to prev env the record declaration of p.

width = type = synthesized attribute

Page No.
Date. / /

the entire thing is S-attributed as we use only synthesized attribute

P → { offset = 0; } D

D → T id; { top.put (id.lexeme, T.type, offset); offset = offset + T.width; }

D → ε (do nothing function)

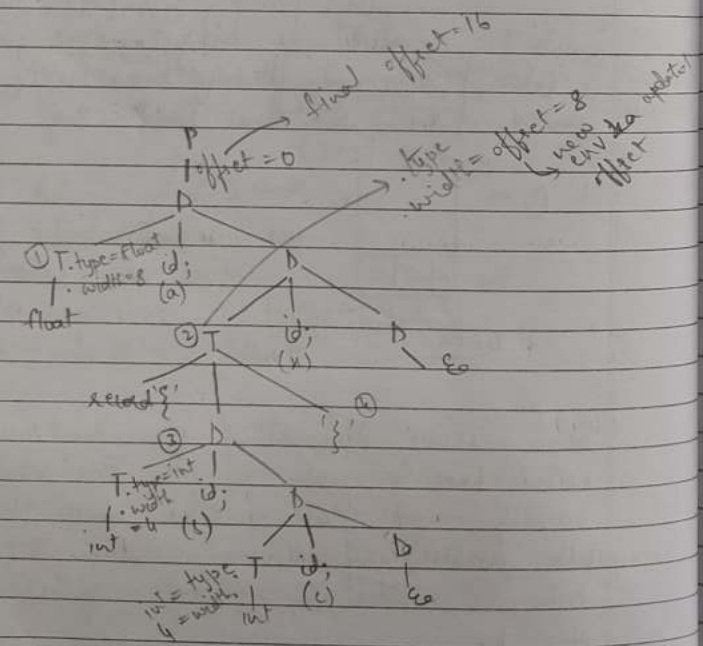
T → record '{' { Env.push (top); top = new Env();
stack.push (offset); offset = 0; }
D '}' { T.type = record (top); T.width = offset;
top = Env.pop (); offset = stack.pop(); }

↳ previous

Q) Write down rule for defining ~~rule~~ sequence of declaration like example, write CFG, draw APT and the grammar with rule above.

Q) Do the above for:

```
float a;
record {
    int b;
    int c;
} x;
```



Page No.
Date. / /

① at T → float, Env 8 (updated offset)

a	float	0

at ③ and ④ new Env 8

c	int	4
b	int	0

at ②

16
8
4

stack

offset = 0.

at ④ Env 16

x	record (top)	8
a	float	0

Total Offset = 16

2/1/25 * **Three - Address Code for Expressions** → one operation per statement

E.addr = lexeme value of the id stored
↳ identifier

E.code = " " since there is no 3AC generated for it

for computation, generate "new Temp()" to get new temporary variable t_1 to store result

gen() is function to generate 3AC

For E → E_1 + E_2, E.code = E_1.code || E_2.code || gen (E.addr '=' E_1.addr '+' E_2.addr)

|| → pipe operator for concatenation

E.code = E_1.code || E_2.code — to be done for each parent with child nodes to get 3AC of each child also

→ Unary have higher precedence than binary operators

For unary, use "minus" in E.code
subtraction,

addr — synthesized attribute

All programming languages Java, C++, etc are all row-major forms ~~tak~~ for array reference where we specify rows first and then columns

Page No.
Date. / /

Symbol Table pointed to by top so to get value of identifier, use "top.get (id.lexeme)"
gen() — function to communicate with output device

→ No need to explicitly check if 3AC for sub-nodes has been explicitly generated — Incremental Translation — there is no .code attribute here — this is the way it is actually done

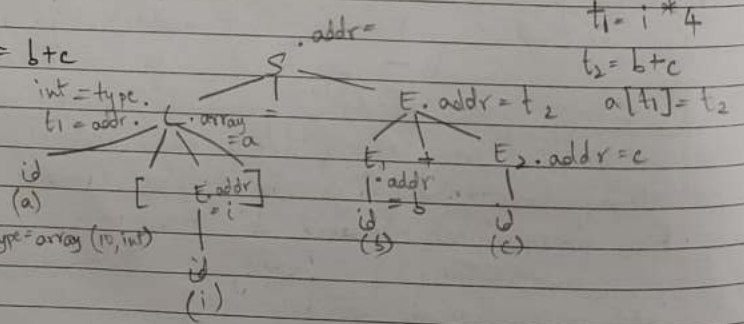
Eg: c+ a [i][j] input: int a[2][2];
a.type = array (2, array (3, int))
For each array reference, there are 3 attributes:
1. .type
2. .addr
3. .array

subtype to be fetched using .type

Q) Write down SDT for array reference, or write semantic actions for array reference
grammar will be given
have to write all semantic rule by taking an example
check pictures (Meet)

6/1/25 Q) a[i] = b+c

$t_1 = i*4$
$t_2 = b+c$
$a[t_1] = t_2$



.addr — check if it exists in symbol table and assign its [illegible]

Data Analytics tools use column-major format for array references.

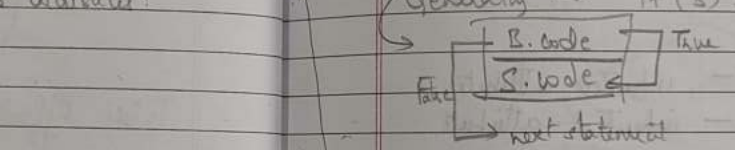
Page No.
Date. / /

* Control Flow
↳ translation (3AC generation) 1 = true, 0 = false

→ 3AC for if statement
Eg: if (x<100 || x>200 && x != y) x=0;

Between arithmetic, logical and relational operators, precedence
highest 1. arithmetic
2. relational
lowest 3. logical (NOT①, AND②, OR③)

(Generally ⇒ if (B) S;

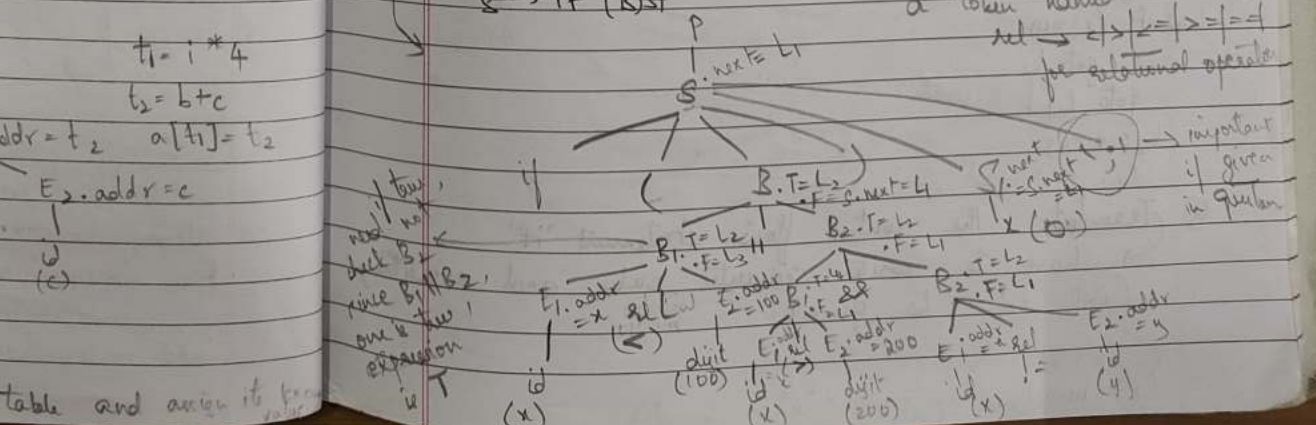


P → procedural statement — can be declaration / programming
check pictures in (Meet) for SDD

~~SDD for if:~~

~~Production~~	~~Semantic Rules~~
~~P → S~~	~~S.next~~
~~S → assign~~	
~~S → if (B)S1~~	

Assume that there is a token named rel → <|>|<=|>=|==|!= for relational operator



→ Two types:

1. Label (2-pass) – not efficient if too many unconditional goto statements
   1st pass – root to all leaves (inherited)
   2nd pass – bottom to top (synthesised)
2.

Before expanding S, we need to tell where the "next" pointer should point to – next instruction – S.next
Before expanding B, we should mention possible values of B – true or false – use B.T and B.F

for previous page example
- .next – inherited attribute, .T – inherited attribute
- .addr – synthesised attribute, .F – inherited attribute

→ SAC for previous page example:

```
[if (x < 100)
      goto L2 → .T
  else
      goto L3
L3: if (x > 200)
      goto L4 → .T
    goto L1
L4: if (x != y)
      goto L2
    goto L1] B.code
L2: x = 0;
L1: undefined
```

only coming here if B1 is false

only coming here if B1 is true

only coming here if L2 is true

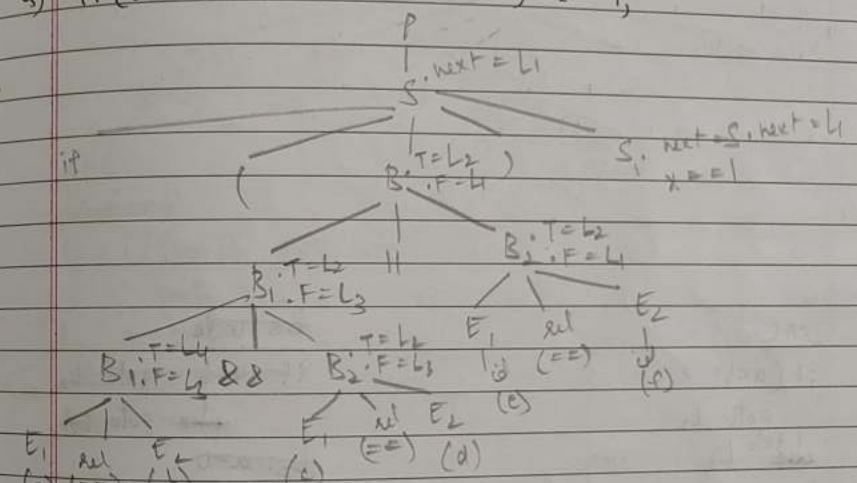
Q) Translate the control flow statement 'if'
→ have to write semantic rules and everything (will not be given)

L2: x = 0;
L1: undefined reference (nothing to do)

Precedence:

A → S
B → B || B | C
C → C && C | D
D → !D | E
E → id

Q) if (a==b && c==d || e==f) x==1;



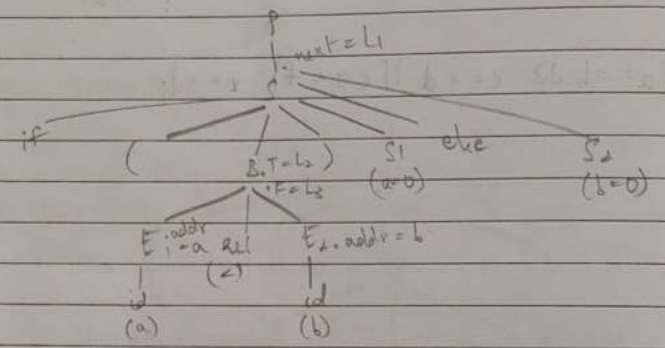

```
if (a==b)
    goto L4
  goto L3
L3: if
```

7/1/25 ~~Write 3-AC for~~:

→ 3AC for if else statement

Eg: $S \to$ if (B) S_1 else S_2

- input: if (a<b) a=0 else b=0; this is 2 pass



3AC:

if (a<b)
  goto L_2
  goto L_3) B's code
L_2: a=0
  goto L_1
L_3: b=0
L_1: nothing

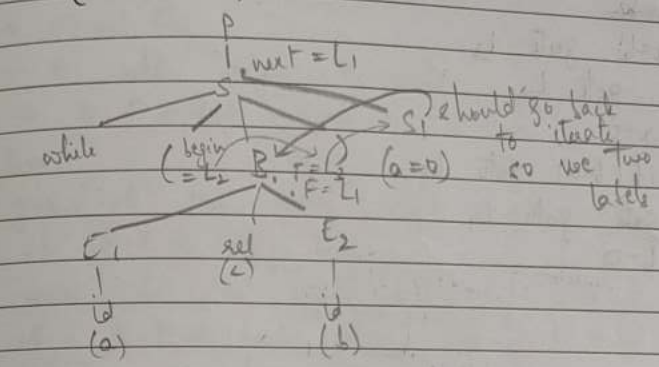
~~B's code~~
~~if a<b goto L_2~~
~~else goto L_3~~
~~L_2: a=0~~
~~goto L_1~~

Q) Know how to write semantic rules to generate 3AC for above

Q) Translate if (a<b) a=0 else b=0; given grammar, to a 3AC.
→ write semantic rules, APT, 3AC

→ 3AC for while

Eg: while (a<b) a=0;



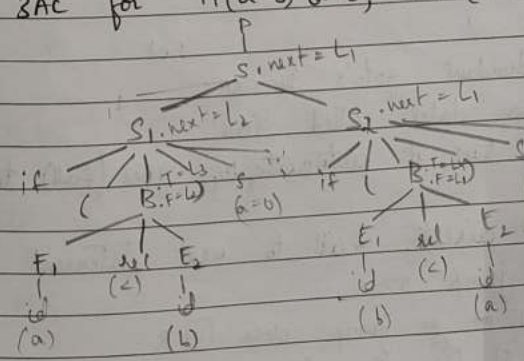
3AC

~~if (a<b)~~

L_2: if (a<b) goto L_3
  goto L_1
L_3: a=0
  goto L_2
L_1:

check semantic rules for control flow statements from "Compilers: Pearson New International..." page 407 something in online PDF

Q) Write 3AC for if(a<b) a=0; if (b<a) b=0; $S \to S_1 S_2$



Page No.
Date. / /

3AC:

~~t: if (a<b)~~

if(a<b) goto L_3
goto L_2

called labels

L_3: a=0
L_2: if(b<a) goto L_4
~~goto~~ goto L_1
L_4: b=0
L_1:

if question was
if(a<b) a=0;
if(b<a) b=0;
a=1; then L_1: a=1;

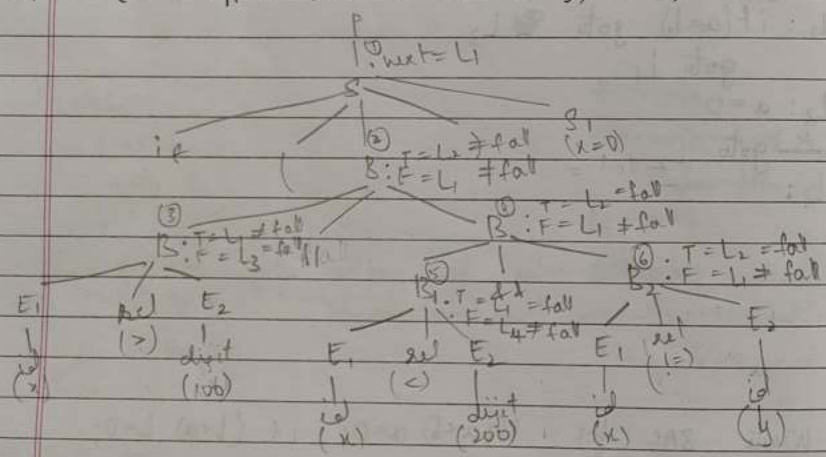
since we use label 'fall'

since it jumps over code

since we use circuit statement called if false

6/1/25 # Fall-through Technique (Jumping code / Short-circuit code)

Q) if (x>100 || x<200 && x!=y) x=0;



Avoid redundant goto's in 3AC
Here, assign label called 'fall'
if .T is true, it automatically goes (fall) to the next statement
= fall ⇒ directly fall to next statement
≠ fall ⇒ jump over
At ③ .T ⇒ jump over ④ B
⑤, ⑥, then ④

SOP — Standard Operating Procedures
→ Secure Coding

Page No.
Date. / /

if B.T ≠ fall ⇒ jump
if true, automatically falls to next statement
B → E_1 rel E_2
test = E_1.addr relop E_2.addr
S = if B.T ≠ fall and B.F ≠ fall,
then
gen('if' test 'goto' B.T) || gen('goto' B.F)
else if B.T ≠ fall, then
gen('if' test 'goto' B.T)
else if B.F ≠ fall then gen('if False' test 'goto' B.F)
else ''

follow conditions need not be written specifically

B → B_1 || B_2
B_1.T = if B.T ≠ fall then B.T else newlabel()
B_1.F = fal
B_2.T = B.T
B_2.F = B.F
B.code = if B.true ≠ fall then B_1.code || B_2.code
else B_1.code || B_2.code || label(B_1.T)

At ③ B .T ≠ fall
if (x > 100) goto L_2

② B.T → concatenate B1

At ⑤ B .F ≠ fall
~~if false~~ if false (x < 200) goto L_4

④ B.F

B.F ≠ fall

At ⑥ ~~if false~~ if false (x != y) goto L_1

⑤ B.F

→ B_2

Just above leaf nodes only

B → B_1 || B_2 — Execute B_2 only when B_1 is false

Q) Translate — no need of semantic rules write APT and translate and ~~draw~~ write 3AC
Q) Write or with help of semantic rules then only write rules along with APT and 3AC

Q) Write rules for && (opposite of ||)

Q) $P \to S$

$P.code = \{S.code \,||\, label(S.next)\}$

3AC:

B_1 if (x>100) goto L_2

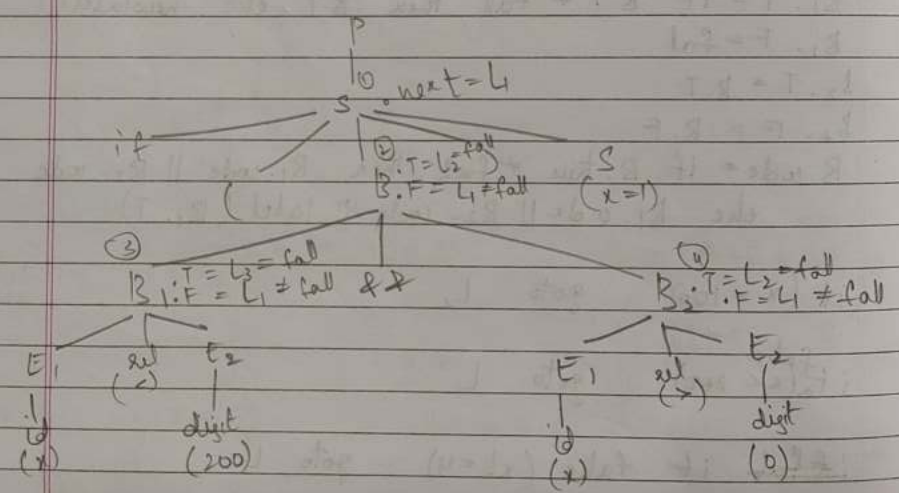
S if false (x<200) goto L_1

B_2 if false (x != y) goto L_1

L_2: x=0;

L_1:

Q) if (x<200 && x>0) x=1;



if false(x<200) goto L_1

if false (x>0) goto L_1

L_2: x=1

L_1:

$B \to B_1$ && B_2

$B_1.F$ = if $B.F \neq$ fall then $B.F$ else newlabel()

$B_1.T$ = fall

$B_2.F = B.F$

$B_2.T = B.T$

$B.code$ = if $B_1.F \neq$ fall then $B_1.code \,||\, B_2.code$

else $B_1.code \,||\, B_2.code \,||$ label($B_1.F$)

Page No.
Date. / /

$B \to B_1$ && B_2

$B_1.F =$ if $B.F \neq$ fall then $B.F$ else newlabel()

$B_1.T =$ fall

$B_2.F = B.F$

$B_2.T = B.T$

$B.code =$ if $B_1.F \neq$ fall then $B_1.code \,||\, B_2.code$

else $B_1.code \,||\, B_2.code \,||$ label $(B_1.F)$

9/1/25 * Back Patching → using a function to patch jumps

→ one-pass code translation

→ using only synthesized attributes

→ 3 functions need:

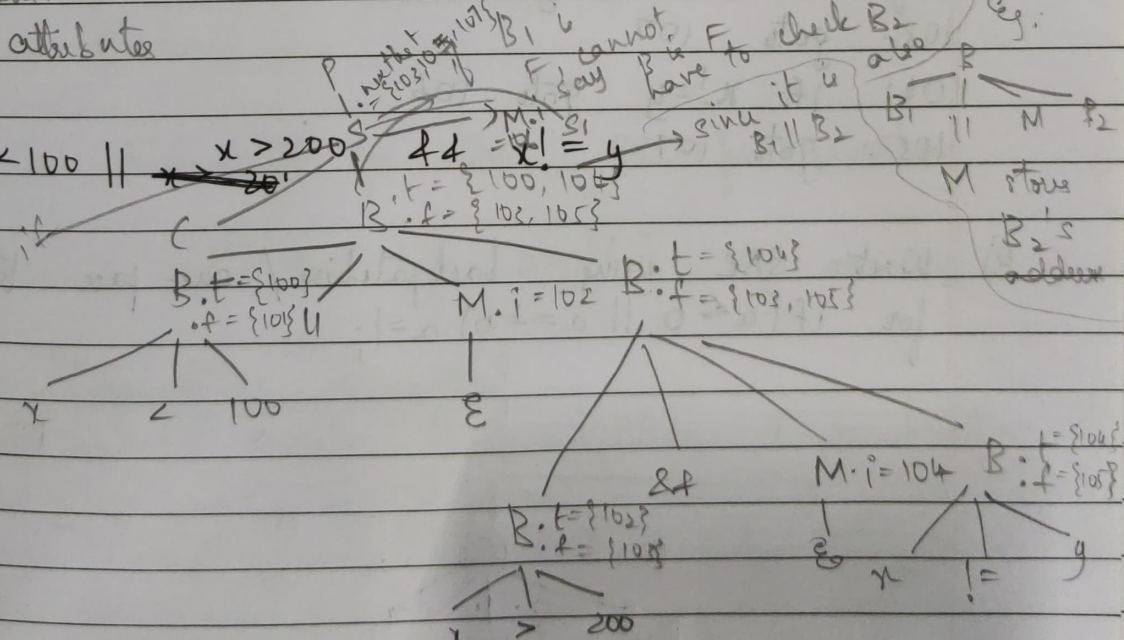
1. makelist(i) — generate next instruction number
2. merge (p_1, p_2)
3. backpatch (p, i) — arranging jumps — where to go?

M — marker nonterminal — ~~get~~ stores next instruction's address

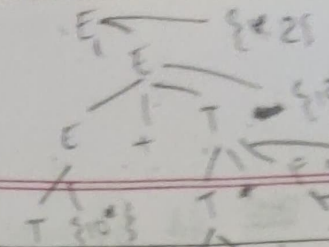
{ truelist − t falselist − f instruction − i

→ attributes

Eg: $x < 100 \,||\, x > 200 \,\&\&\, x != y$



asked for exit



Q) Consider the SDT:

$E_1 \rightarrow E$ out $\{*_2\}$

$E \rightarrow E+T$ out $\{1\}$

$E \rightarrow T$ out $\{10*\}$

$T \rightarrow T*F$ out $(*)$

$T \rightarrow F$ out $(100+)$

$F \rightarrow$ num out (num.val)

The input is '1+1*1' and output generated is equivalent to result of an arithmetic expression.

Value obtained is: 1100+10*1100+1*1*2

* <u>Backpatching</u> <u>continued</u> → always use synthesized attr and is bottom-up

Backpatching fills in goto —

→ 3AC for question in previous page:

100: if x<100 goto 106

101: goto 102

102: if x > 200 goto 104

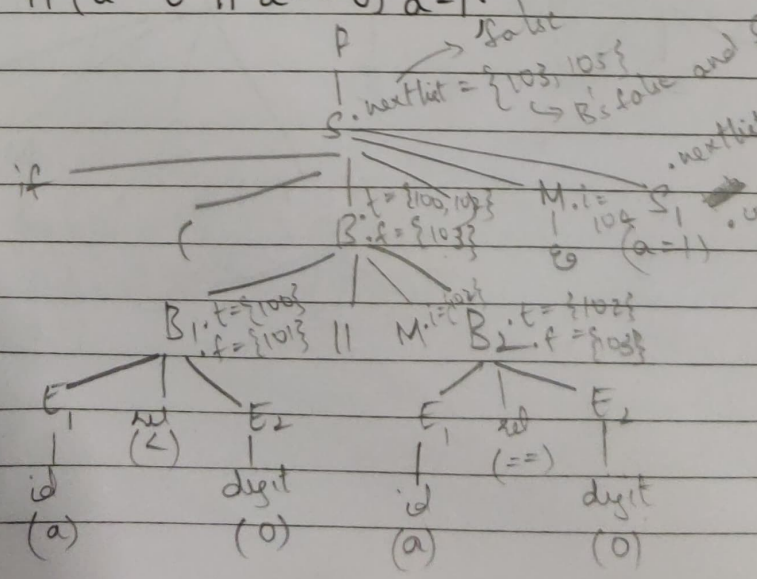
103: goto 107

104: if x != y goto 106

105: goto 107

Q) Write 3AC using backpatching / one-pass translation for if(a<0 || a==0) a=1.

Assume first instruction = 100



Page No.
Date. / /

3AC:

100: if a<0 goto 104
101: goto 102
102: if a==0 goto 104
103: goto 105
104: a=1
105:

* **Intermediate Code for Procedures**

learn 3AC for functions but semantic rules for functions not required
→ switch case, pointers not included

→ Semantic Rules ~~not~~ will not be given in the question paper / questions anywhere

important
Q) What are the issues in design of a code generator?
→ for CIE 2 also
→ check PPT

LD for load, ST for store, BR for branching

Q) Machine instructions for a = b[i] or similar

* **Code Generation Algorithm**

getReg(I) – get register of instruction I – function
→ check if value is already in register and reuse it, or create new register

Eg. I: t = u + v

LD R_1 v
LD R_2 u
ADD ~~R_1, R_2~~ R_3, R_1, R_2 or R_1, R_1, R_2
→ if R_1 is not being used again

Register descriptor – tells which registers are in use and what values they hold

Page No.
Date. / /

Reg descriptor

R_1	R_2
v, t	u

address

Addr descriptor

u	v	t
R_2	-	R_1

R_1 getting t stored in it so no Register for v as R_1 is getting replaced