unsaved values in symbol table
— should be written directly

FSM → Finite state machine

① Moore (z → depends on current state)

② Mealy (z → both current and inp w)

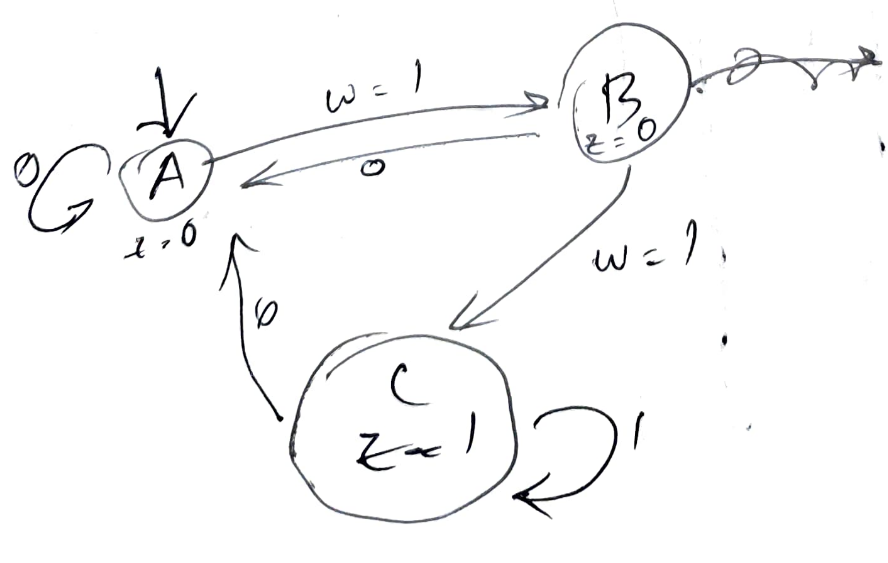
Moore

if W = 1 1 | 1

Z = 0 0 | $\boxed{1}$

Current state - y

Next " - Y



overlapp 1 1 1 1

non overlap 1 1 1 1

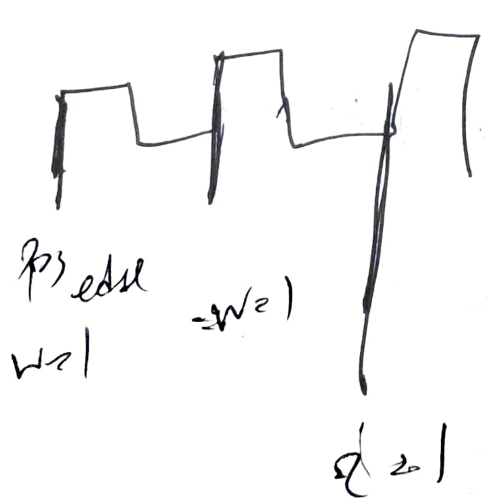
make state table

if

z	
w=0	w=1

mili type

③ make state assign table

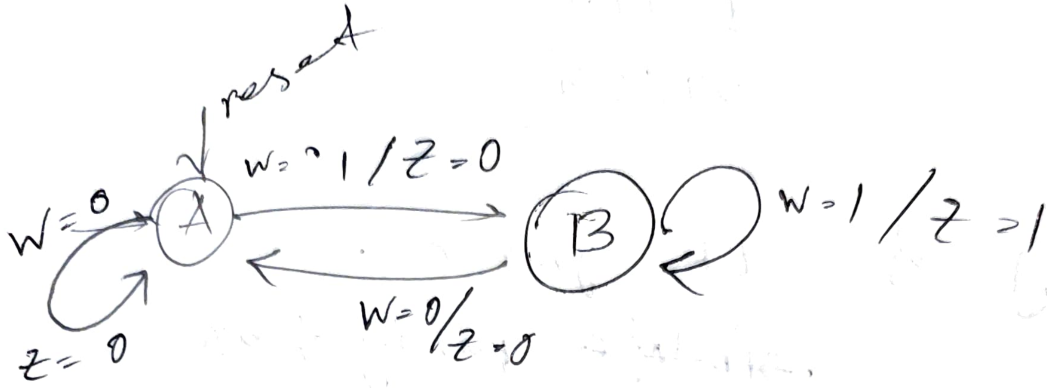


mealy

immediate

w = 1 1 1

z = 0 1 1



state table

curr state y	next state Y		z	
~~A~~	w=0	w=1	w=0	w=1
A(0)	A	B	0	0
B(1)	A	B	0	1

CSE 420

9/1/24

Parsing of SLR gramer.

- ~~simulate~~ given parsing table

given string - see if parsable

Syntax Directed Translation

1 denotes - its in prod body

$E \rightarrow E_1 + T$

example: ex: (a*c) + b*c

|| → concatination

postfix of a+b = $\boxed{ab+}$

semantic rule

E.code = E1.code || T.code||'+'

3 addr

High level
↓
LA
↓ tokens
SA (check tokens)
↓
SA - semantic analysis (if logic of code correct)
↓
Assembly

in x
$x := 1.25$

ADD BL, CL → BL = 11
5, 6
↓
machine language
0101 . . .

6.3 address codes:

$(a+b) - c * d$ - can't do all in one line of assembly code

Compiler converts this to an assembly language similar to

$t_1 = a+b$, $t_2 = c*d$

$t_3 = t_1 - t_2$

BL ADD CL

$t_1 = a + b$ $\longrightarrow$ Add BL, CL

$t_2 = \ldots$ conv to one line

2 operands and one operators
— 3 add code

compiler brings the code in a structure so that its possible to bring it in 3 address code format.

In syntax analyser:

- checks if token matches w grammar
- using symantic rule (E.code = E, c ...)
  - generates 3 address code that can be con to assembly

S. D. Definition is production rule with semantic rule

S. D. Translation: Implementation of SD. D.

semantic action: { }

convert semantic rule to semantic action

In semantic rule

Each ~~terminal~~ terminal and non terminal will have an attribute

↓

E . code

Ex

CFG rules

(S produces E)

1. $S \to E$
2. $E \to E + T$
3. $T \to T * F$
4. $F \to id$

Non terminal

E, T, F

Terminal

+, *, id

create semantic actions → both terminal and non can have attributes - not necessary

2. $E \to E + T$

$E.code = E_1.code + T.code$

OR = $E_1.code \, || \, +.code \, || \, T.code$

(for ex .code for plus is '+')

Sem rules!

- terminals and non terminals
- can have attributes.

$\boxed{E_1.var[0]}$ ← from an array of vars.

for a gramer a semantic rule will be given — formulate 3 add code

2 * 3 + 5 + 6

↓ can directly calculate w/o using 3 add codes

S → E

E → E.val + T.val | E

T → T * val * F.val | T.val | F.val

F → num

— using semantic rule can have various operation

Attributes 2 types

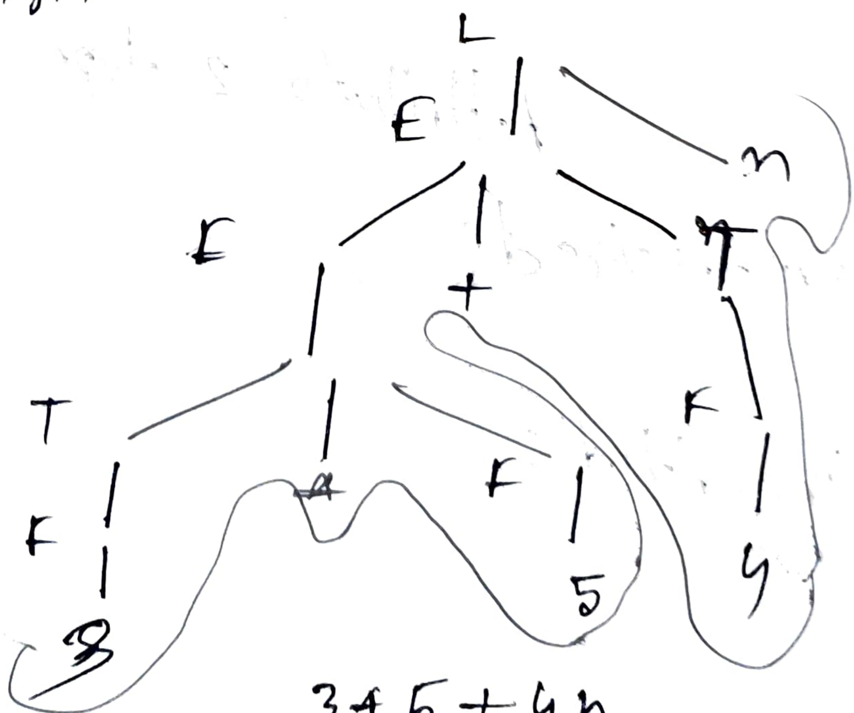
① synthesized:

② inherited:

① synthesis :

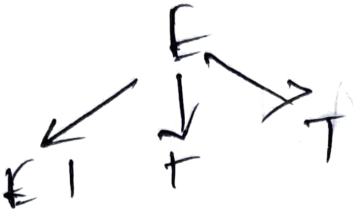
3 * 5 + 4 N

P	symantic rules
$L \to E n$	$L.val = E.val$
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val \times F.val$
$T \to F$	$T.val = F.val$
$F \to F$	$F.val = E.val$
$F \to (E)$	$F.val = digit.lexval$
$F \to digit$	



3 * 5 + 4 n

② $E \to E1 + T$ when construct a parse tree using this rule:



For each node, prod head $E.val$ which type

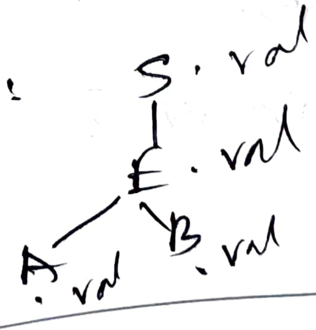
Find out: how it is calculated

$E.val = E1.val + T.val$

when value of a node is calculated by calculated by its own att val ~~and~~ att val of its children

→ synthesized

Inherited: ex:



$\therefore E.val = S.val + E.value$

inherited

$B.val = E.val$

synthesized

$E.val = A.val + E.val$

$E \to E.val$ → ~~inherited~~ synthesised

$E = A + B + E$

inherited: can from - sibling
- parent
-

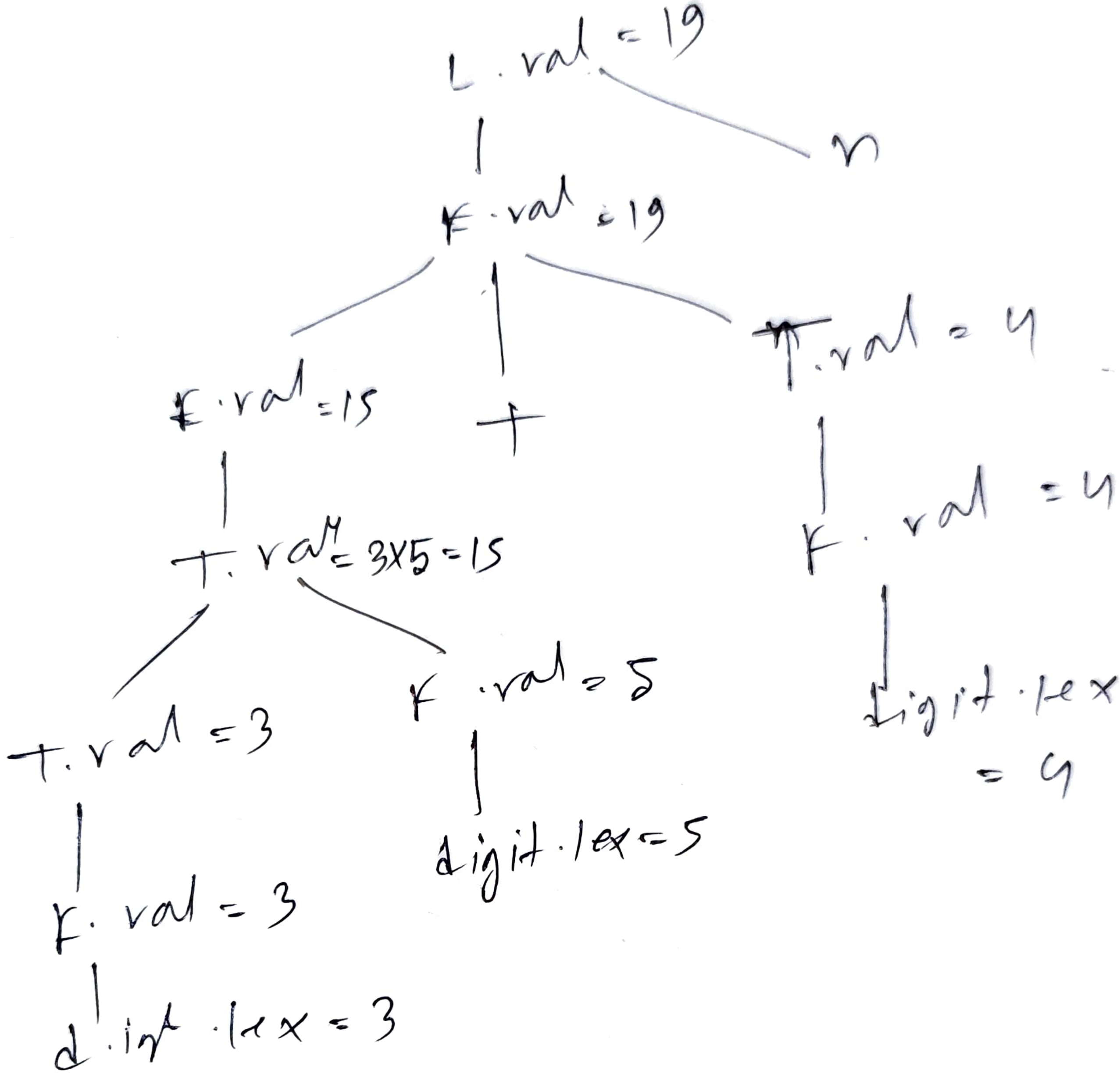
} itself

$B = A + S$

DFS if all are synthesized attributes for all nodes : for calculations
can't DFS for evaluating inherited attributes
- manually do

Evaluate an SDT if all are synthesize using DFS

1. create parse tree
2. write attributes.
3. Traverse in DFS



$\boxed{3 * 5 + 4 n}$

end of inp

~~not~~ different entity.

$3 * 5 + 4n$

(1) ~~Reach~~ digit. lexim : d.lex = 3

(2) Then F-val = dig.lex

$\therefore F = 3$

(3) T.val = F.val = 3

(4) ~~*~~

(5) digit.lex = 5

(6) F.val = digit.lex = 5

(7) T.val = T_1.val * F.val

= 3 * 5

(8) E.val = 15 (9) +

(10) digit.lex = 4 (11) F.val = 4

(12) T.val = 4 (13) E.val = E_1.val + T.val

= 15 + 4

= 19