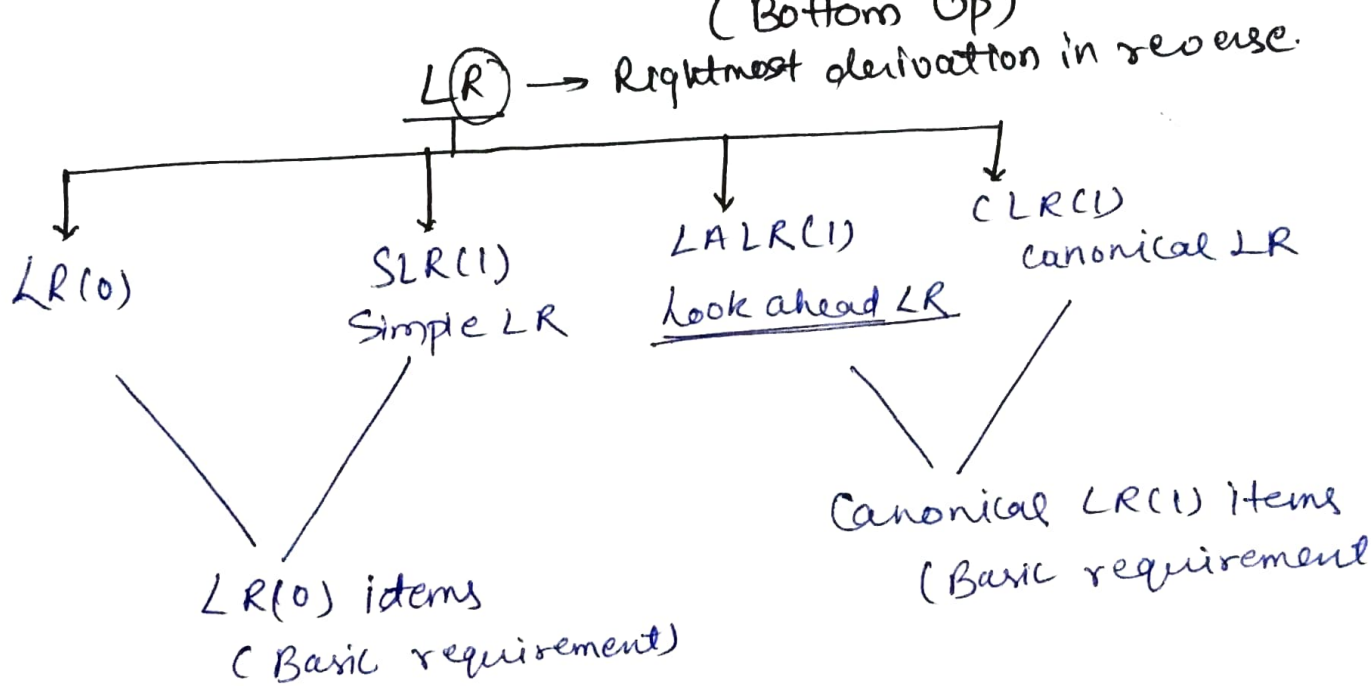
LR Parsing, LR(0) items and LR(0) parsing table

(Bottom Up)



Ex $S \rightarrow AA$

$A \rightarrow aA \mid b$

↓ $S' \rightarrow S$

Augmented Grammer

LR(0) Parsing table

- closure
- goto

1	$S' \rightarrow S$
2	$S \rightarrow AA$
3	$A \rightarrow aA \mid b$

→ Just add one production to make it augmented.

Any production with dot (•) at beggining called as LR(0) item.

$S \rightarrow \bullet AA$

Position of dot can be changed as -

$S \longrightarrow .AA$ → Not seen anything on RHS [LR(0) item]

$S \longrightarrow A.A$ → seen A on RHS

$S \longrightarrow AA.$ → seen everything from RHS. [final item]

Steps

① $S' \longrightarrow .S$: As you see '.' in the beginning of product then include all production derived from variable on RHS of '.' and their respective production

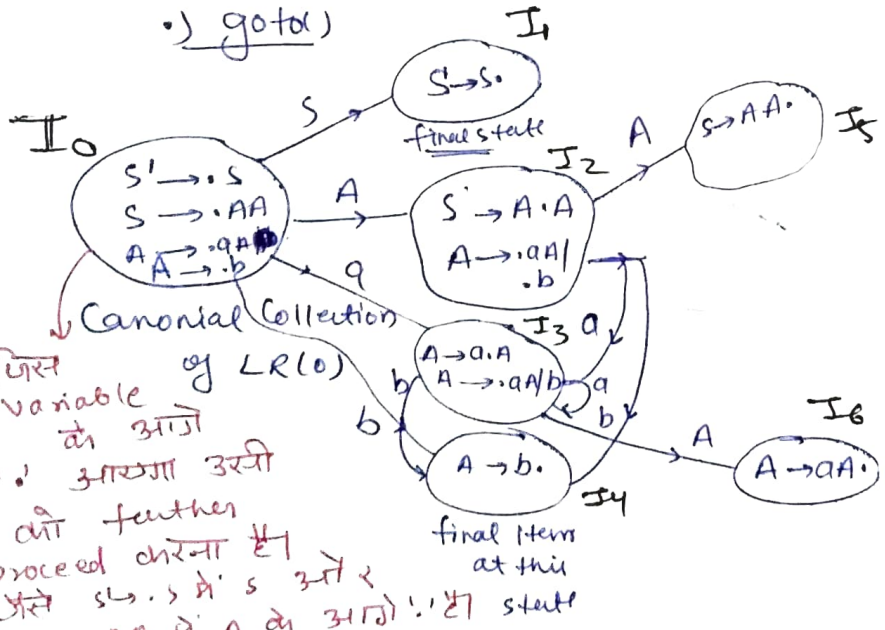
•) Closure of item: $S' \longrightarrow .S$ is

$S' \longrightarrow .S$

$S \longrightarrow .AA$

$A \longrightarrow .aA \mid .b$] Respective prodn of A.

•) goto()



Canonical Collection of LR(0)

जिस variable के आगे '.' आएगा उसी को further proceed करना है। जैसे $S \longrightarrow .S$ में S के आगे '.' है $S \longrightarrow .AA$ में A के आगे '.' है

Move '.' to one step RHS and find closure

$I_0 \longrightarrow I_6$ (7 States)

→ Same level पर indexing करो फिर next level पर आओ-वहाँ

LR(0) Parsing Table

I	(a) Action		(b) ($)	Goto(A)	(S)
0	S_3	S_4		2	1
1			accepting		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

→ We accept the I_1 because it derives from augmented production specially added to it. So, it needs to accept

Points to Remember

•) Assume

$S \rightarrow AA. \rightarrow r_1$
$A \rightarrow aA. \rightarrow r_2$
$A \rightarrow b. \rightarrow r_3$

] reduced move

$S_3 \rightarrow$ Shift on 3: we will do shift operation on looking a terminal from any set of production

$S_3 \rightarrow$ means I_0 on a shifts to I_3

we use numerics in case of showing transitions for variables. as.

Goto(A): 2 means I_0 on 'A' moves to I_2.

This way we use different notations for both terminal and variables.

Since I_4, I_5, I_6 are final reduced states so we write the productions whose finally reduced form represent by the state.

Ex, $A \rightarrow b$ is r_3 its reduced $\overset{\text{move}}{\Rightarrow}$ is $A \rightarrow b.$ and it is accepted by I_4.

So 4 number entry contain r_3 in all columns.

This way a complete table is designed.

LR(0) parsing example and SLR(1) table

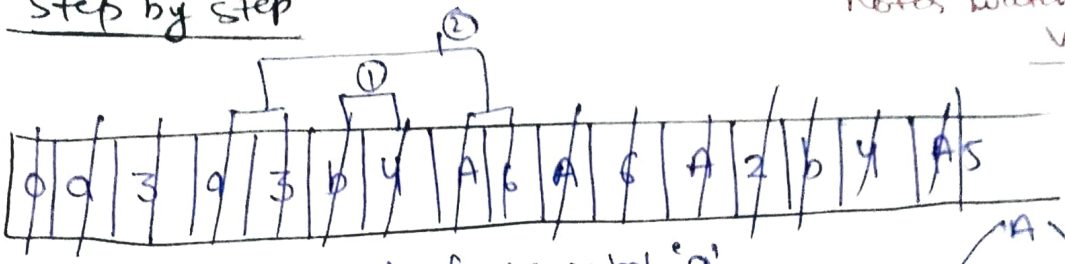
Example: $S' \rightarrow S$
$S \rightarrow AA$
$A \rightarrow aA \mid b$

] LR(0) Parsing table for given grammar prepared on previous page (backside)

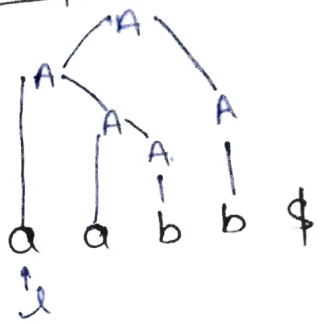
string $w =$ aabb $

→ If don't understand by notes watch the video

Step by step



Start from state 0 and first symbol 'a'



① $0 \xrightarrow{a} 3$ (Increment pointer)

② $3 \xrightarrow{a} 3$ (Increment i)

③ $3 \xrightarrow{b} 4$ (Increment i)

④ $4 \xrightarrow{b} r_3$ ($A \to b$)

Reduce $b \to A$ and then Push A in the stack. We are reducing the previous b (before 4) not current pointing b.

If in reduced production (r) have n elements on RHS then pop (2n) elements from stack.

No. of element on RHS means Ex→

$A \to a \to$ 1 element on RHS (a)

$A \to aB \to$ 2 element on RHS.

So reduced symbol $b \to q$ and pop 2 symbols from stack.

⑤ $3 \xrightarrow{A} 6$ (Pointer on b)

⑥ $6 \xrightarrow{b} r_2$ ($A \to aA$) reduce and pop 4 elements.

⑦ $3 \xrightarrow{A} 6$ (Pointer on b) ~~reduce~~

⑧ $6 \xrightarrow{b} r_2$ same as above and poped 4 elements

⑨ $0 \xrightarrow{A} 2$

⑩ $2 \xrightarrow{b} s_4$

⑪ $4 \xrightarrow{\$} r_3$ reduce $A \to b$

⑫ $5 \mid \xrightarrow{\$} r_1$ $S \to AA$ (pop 4 symbols and reduce to AA

SLR (1) → More powerful than LR(0). It also take LR(0) items to parsing.

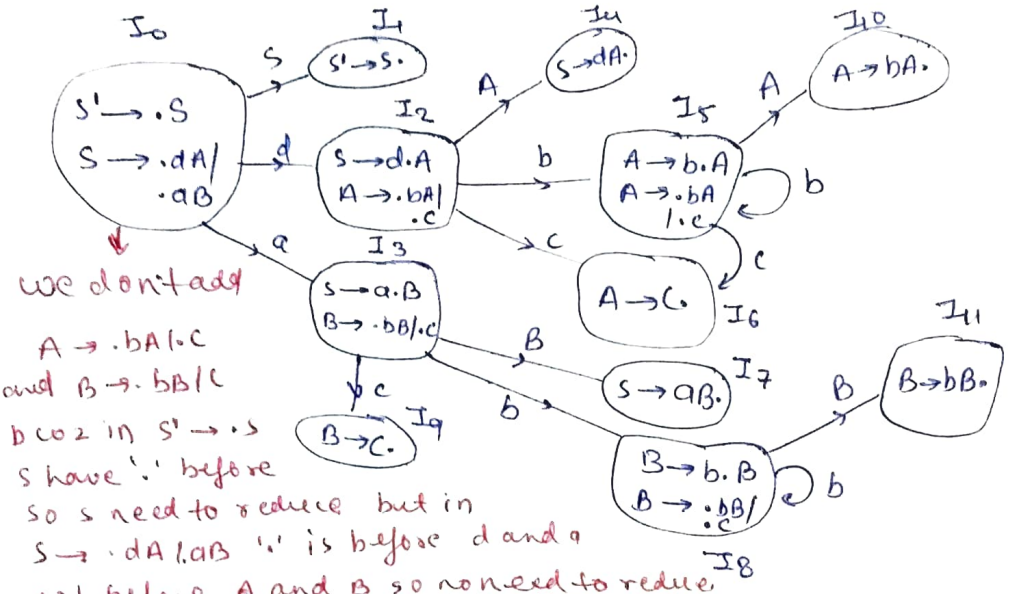
It avoids blind reduction. (Remaining part after next Qus) +11 ₹9

Ex- S → dA | aB
A → bA | c
B → bB | c

To find that Grammar is
i) LL(1) or not
ii) LR(0) or not
iii) SLR(1) or not

Do this question after reading this

→ There is no 2 production which comes in same row/column on finding first(variable). So it is definitely LL(1).



we don't add
A → .bA/.C
and B → .bB/C
bcoz in S' → .S
S have '.' before
so S need to reduce but in
S → .dA/.aB '.' is before d and a
not before A and B so no need to reduce

Canonical Collection of LR(0) items

If grammar follows LR(0) then it must be SLR(1) because SLR(1) is reduced form of LR(0).

The grammar is LR(0) because no final state is having any reduce move or shift move (RR / SR conflict).

final state means '.' at the end.

Remaining part of previous question

SLR(1) says whenever there is a final state then don't put the reduce moves blindly.

SLR(1)	action			goto	
	a	b	$	A	S
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

A
|
a a b b $
↑
SLR(1) says that — cont.

On reducing the variable 'b' to 'A' verify that the variable or terminal just after 'b' (here 'B') must be in follow of A otherwise don't reduce.

So follow(A) = { $, b, ... } → since it contains 'b' so we can reduce $b \rightarrow A$.

Remember ↓

It can be said as $[A \rightarrow b.,\ r_1]$ can be placed if the variable next to 'b' (here 'b') is in follow of LHS of 'r_1'.

SLR(1) detects the error earlier than LR(0).

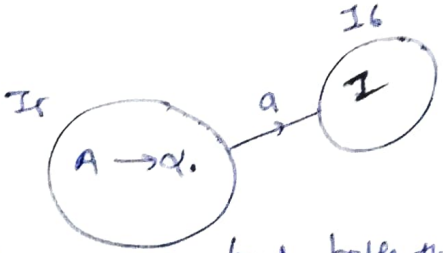
How to handle the error?

→ Provide pointer to function on blank space. so that function can print the error messages.

A Grammar can't be in LR(0) due to reasons –

① SR Conflict

Ex:

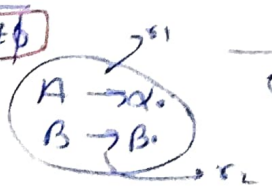


	a	b
5	$S6/r_1$	r_1

In I5 we need to put both the things the reduce move and the shift move.

② RR conflict

follow(A) ∩ follow(B) ≠ ϕ



	a	b	c
5	r_1/r_2	r_1/r_2	r_1/r_2

It generate the conflicts btw two reduce moves

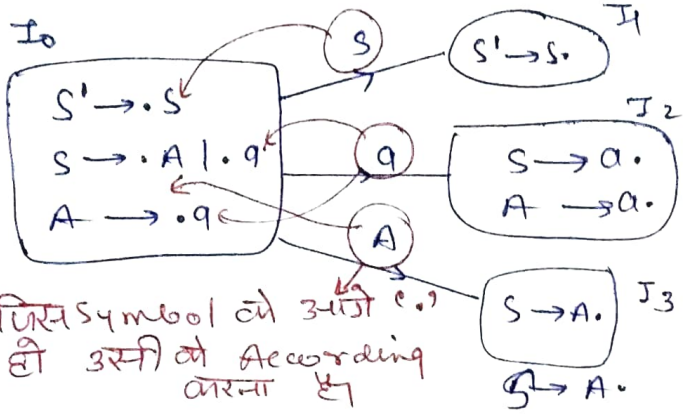
If in canonical LR(0) items graph there is no SR / RR conflicts then it must be in LR(0).

Drop

If grammar is LR(0) then it must be SLR(1), but if grammar is not LR(0) it doesn't mean that it will not be SLR(1).

Ex → S → A|a
A → a

① Not suitable for LL(1) bcoz first(S) = {a, a} and it is also ambiguous. Short Notes



→ Two reduce move in same state so not a LR(0).

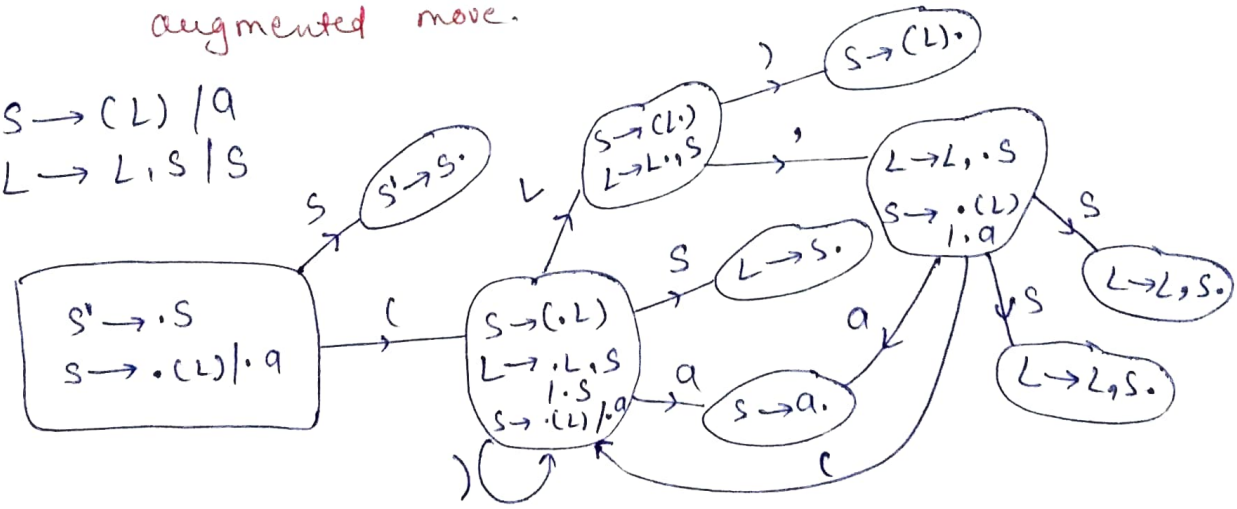
→ जिस Symbol को आगे (.) हो उसी को According वारना है

for SLR(1) find follow(S) = $ and follow of(A) is $.
so two entries needs to placed in same column not possible.

→ Note → ① Left recursive grammar can't be LL(1).

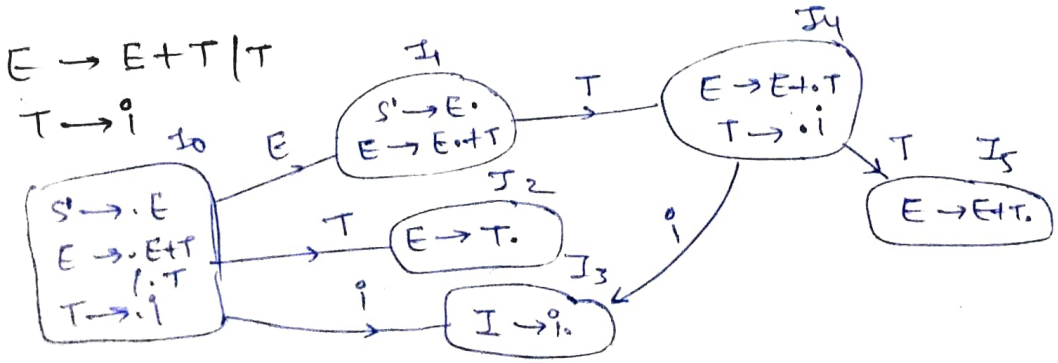
② for LR(0) & LR(1) check only final states except augmented move.

S → (L) | a
L → L,S | S



→ Grammar is not LR(0).
→ Grammar follows LR'(0) and SLR(1).

E → E+T | T
T → i



We can assume that I_4 is a final state due to $S' \to E.$ and its also have reduce move $E \to E.+T$ and shift move on T. So there is SR conflict.

But we don't consider $S' \to E.$ as a final state bcoz it is augmented production which is accepting, but not a final state. So there is no SR conflict.

Ex

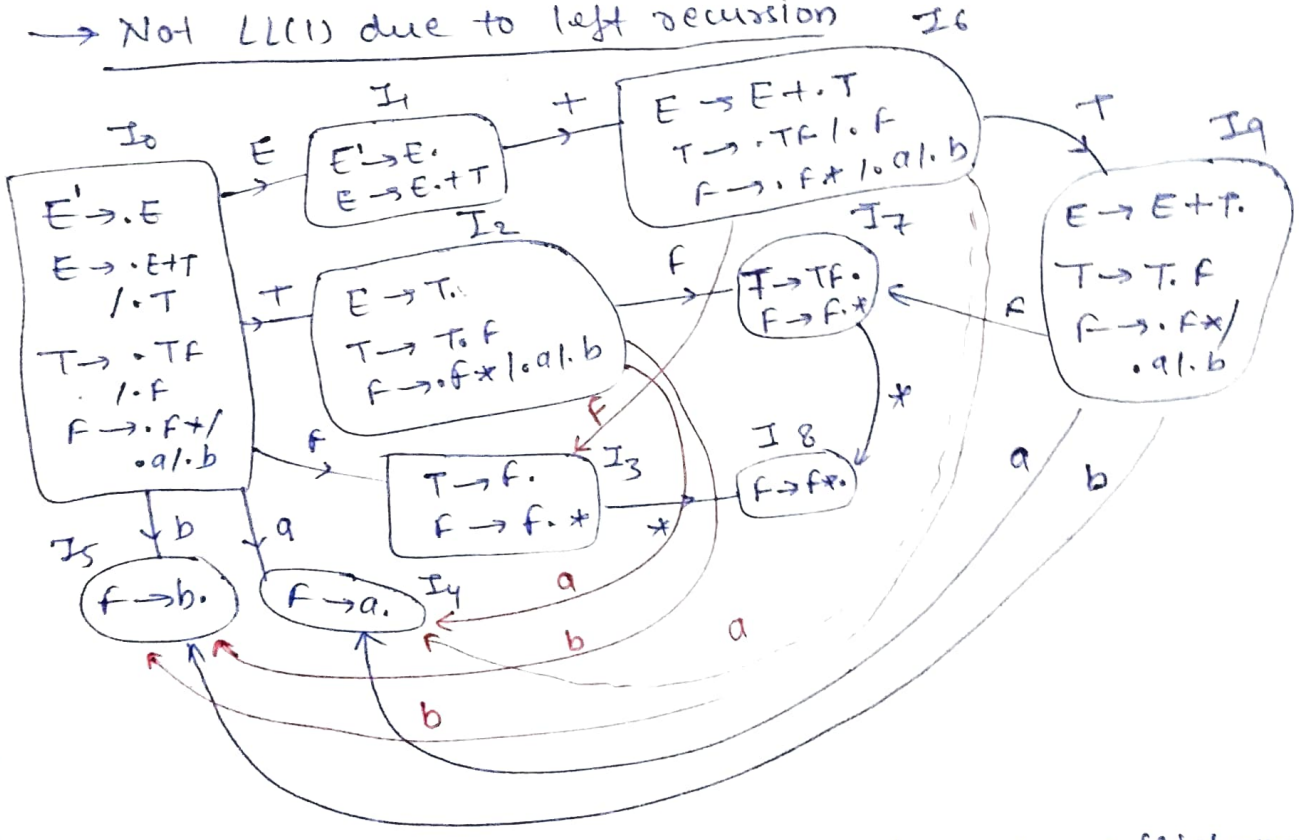
$E \to E+T \mid T$ (1, 2) → Union

$T \to TF \mid F$ (3, 4) → Concatenation

$F \to F* \mid a \mid b$ (5, 6, 7) → simple premitive / Kleen closure

this grammar represents regular expression

→ Not LL(1) due to left recursion



→ Grammar is not LR(0) due to SR conflict on I_2.

→ If needed then generate parsing table for only those reductions generated the conflicts, or conflicts generally observed in action part only.

I	a	b	+	*	$
I_2	S_4	S_5	r_2		r_2
I_3	r_4	r_4	r_4	S_8	r_4

↓ Not LR(0) but SLR(1)

follow(E) = {+, $}

follow(T) = {+, $, a, b}

follow(F) = {+, *, $, a, b}

	a	b	+	*	$	Continue
I_7	r_3	r_3	r_3	S_8	r_3	
I_9	S_4	S_5	r_1		r_1	

] ⟶ LR(0) ✗ due to SR but conflict resolved in SLR(1)

I_2, I_3, I_7, I_9] Generates SR conflict but it get resolved in SLR(1)

So grammar is SLR(1) but not LR(0).

CLR(1) and LALR(1) Parsers

LALR(1) CLR(1)

Canonical collection of LR(1) items

LR(1) ⟶ LR(0) item + look ahead

$S \longrightarrow .aA$ [LR(0)]

LR(1): $S \longrightarrow .aA$, a/b look aheads.

Ex: $S \longrightarrow AA$
$A \longrightarrow aA \mid b$

$S' \longrightarrow .S$ ⟶ LR(0)
$S' \longrightarrow .S \mid \$$ LR(1)
$S \longrightarrow .AA, \$$
$A \longrightarrow .aA / .b, a/b$

Process → ① write augmented production LR(0) form

$S' \longrightarrow .S$

Since this is augmented, so make it LR(1) by adding look ahead $.

$S' \longrightarrow .S, \$$ [LR(1)]

② on $S' \longrightarrow .S$, (".") dot is before S so add all production with S on LHS

$S \longrightarrow .AA$

To find out lookahead take the every symbol existing just after the symbol or variable followed by dot to

$S \longrightarrow AA$ is due to $S' \longrightarrow .S, \$$ (2) so in (2) after .S is only $. So find first($) to get look ahead. first($) = $ itself. So LR(1) of $S \longrightarrow .AA$ is $\boxed{S \longrightarrow .AA, \$}$

Generalise

if $A \longrightarrow \alpha . B\beta, a/b$

then $B \longrightarrow .$ (anything), look ahead

look ahead = first(βa) / first(βb).

because after B there is $\beta(a/b)$.

So LR(1) for given grammar

$S \longrightarrow AA$

$A \longrightarrow aA \mid b$

LR(1)

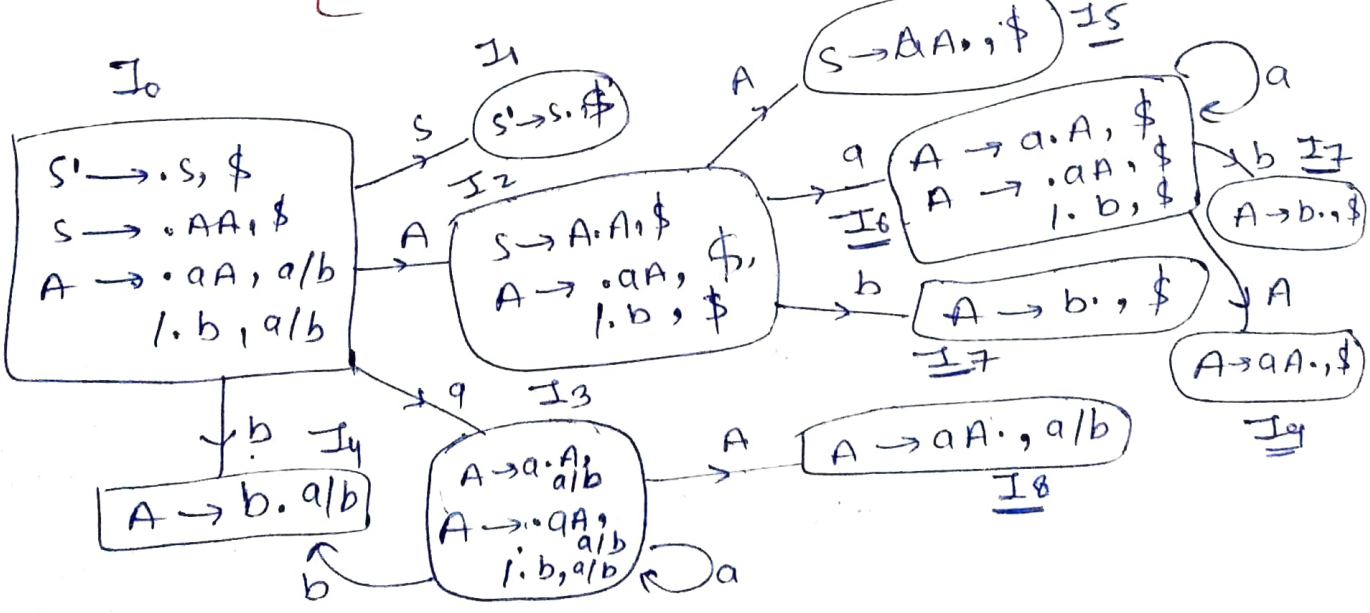
$I_0 \leftarrow$

$S' \longrightarrow .S, \$$

$S \longrightarrow .AA, \$$

$A \longrightarrow .aA$, first($A\$$) = a/b = $.aA, a/b$

$/. b$, first($A\$$) $/. b, a/b$



→ On applying transition look ahead might not changed but on applying closure look ahead will get changed.

No. of states in LR(1) canonical collection is more than LR(0) canonical collection for same grammar.

CLR(1) parsing table no. of states is equals to no. of states in LR(1) canonical collection. diagram.

In CLR(1) the reduced item will get placed in the look aheads.

In LR(0) we placing the reduced item in complete row but in SLR(0) we were placing it in follow(LHS) but in CLR(1) placing in only look aheads only. It reduce the no. of conflicts, but due to this Blank spaces and error detection capability is going to increased

① I_3, I_6 → both have same LR(0) items but different look aheads.

② I_4, I_7

③ I_8, I_9

CLR(1) Parsing table

	a	b	$	S	A
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7	r_2	r_2	r_3		
8			r_2		
9					

Shift move and goto move are same as LR(0) table but reduce moves are different

$S \rightarrow AA$ ... r_1

$A \rightarrow aA$; r_2

$A \rightarrow b$; r_3

Table have very big size so we can reduce the size by merging the space with same LR(0) as

$[I_3, I_6] \rightarrow I_{36}$ $[I_4, I_7] \rightarrow I_{47}$, $[I_8, I_9] \rightarrow I_{89}$

no. of states in CLR(1) $\geq$ no of states in LR(0) or SLR(1)
No of states LR(0) = SLR(1)

↓

After merging some spaces it becomes LALR(1).

$$CLR(1) \geq LR(0) = SLR(1) = LALR(1)$$

↳ No. of states.

then LALR(1) parsing table look like

CLR(1) Reduced to LALR(1).

	a	b	$	S	A
0	S_{36}	S_{47}			2
1					
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3			
5			r_1		
36	S_{36}	S_{47}		89	
47			r_3		
89	r_2	r_2			
89			r_2		

union

⇒

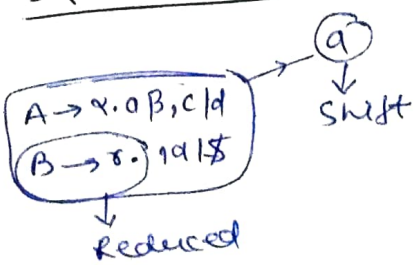
	a	b	$	S	A
0	S_{36}	S_{47}			2
1					
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

union

Deleting duplicates after finding union.

LR(1) items

SR conflict → Dependson look ahead ← RR conflict



A → α. , a
B → α. , a

↳ Both are reduced and final but due to same look ahead both get placed in same column

If a grammar is not CLR(1) then it must not be LALR(1).

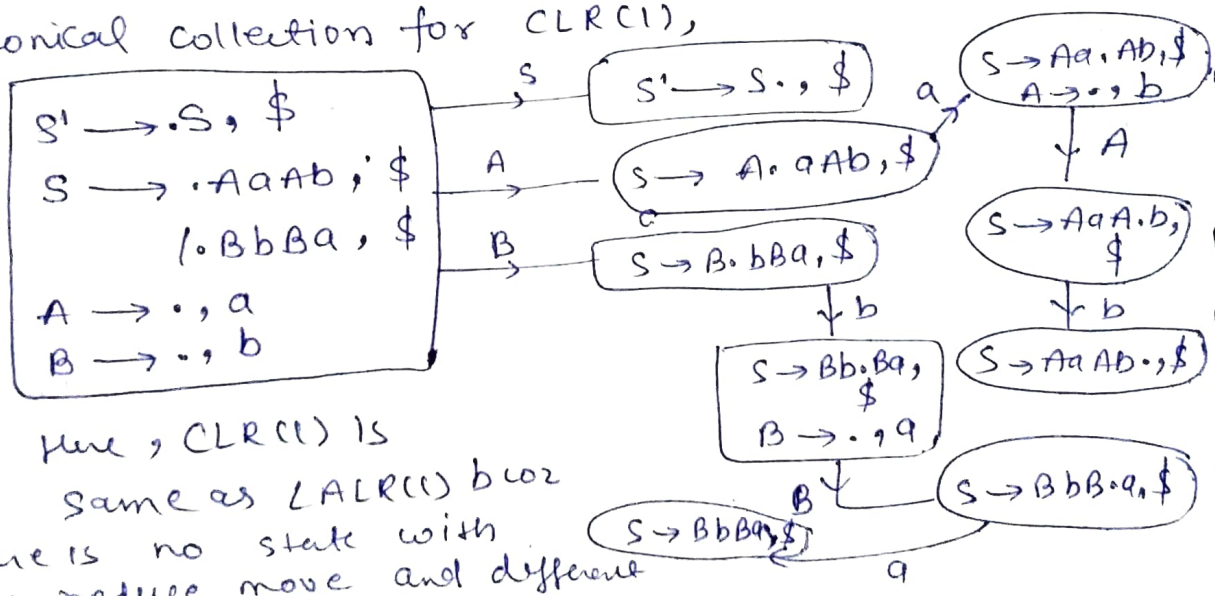
If a grammar is CLR(1) then it is not necessarily LALR(1)

If there is no SR conflict in CLR(1) then it must not be in LALR(1), but if there is no RR conflict in CRL(1) it is not necessary that there is no RR conflict in LALR(1).

Ex $S \rightarrow AaAb \mid BbBa$
$A \rightarrow \epsilon$
$B \rightarrow \epsilon$

① LL(1) ✓
② LR(0) ✗
③ SLR(1) ✗
④ CLR(1) ✓
⑤ CLR(2) ✓

Canonical collection for CLR(1),



→ Here, CLR(1) is same as LALR(1) because there is no state with same reduce move and different look aheads.

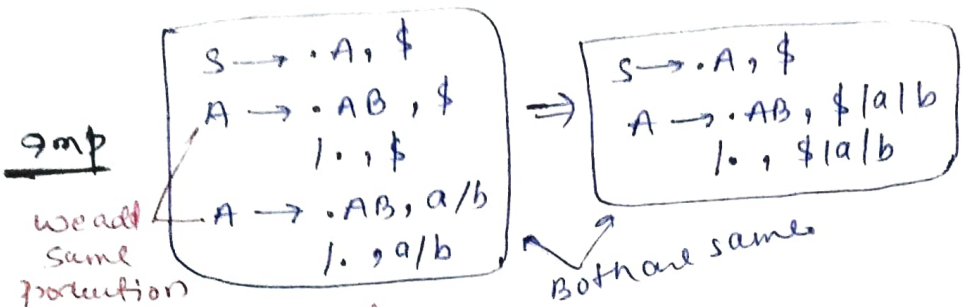
→ On terminal it is shift and on variable it is goto.

Example of CLR(1) and LALR(1) and comparison of all parsers.

Grammar
$S \rightarrow A$
$A \rightarrow ABE$
$B \rightarrow aB \mid b$

Closure (S → .A , $)

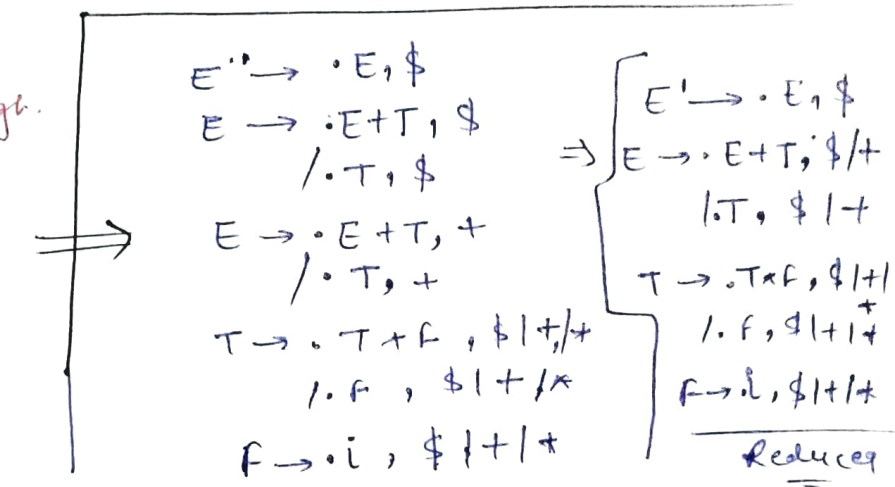
↓



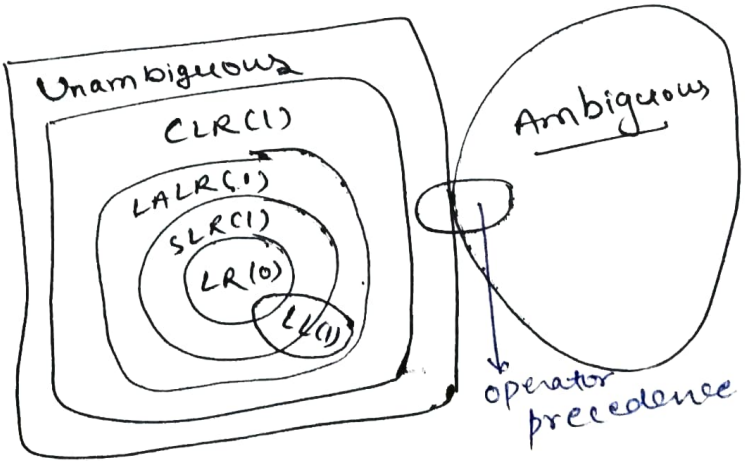
Imp

We add same production twice bcoz we get different look aheads, but in case of 2 R(0) only one production is enough.

② E → E+T | T
T → T*F | F
F → i



Power of Parsers



* If grammar is LL(1) then it must be LALR(1).

* In SLR(1) and LALR(1)
① Goto moves will always identical.
② Shift moves
③ Reduce entries in tables may be different.
④ Error entries in table may be different.
⑤ Both have same no. of states.

$S \rightarrow CC$

$C \rightarrow cC/d$

a) LL(1)

b) SLR(1) but not LL(1)

c) LALR(1) but not SLR(1)

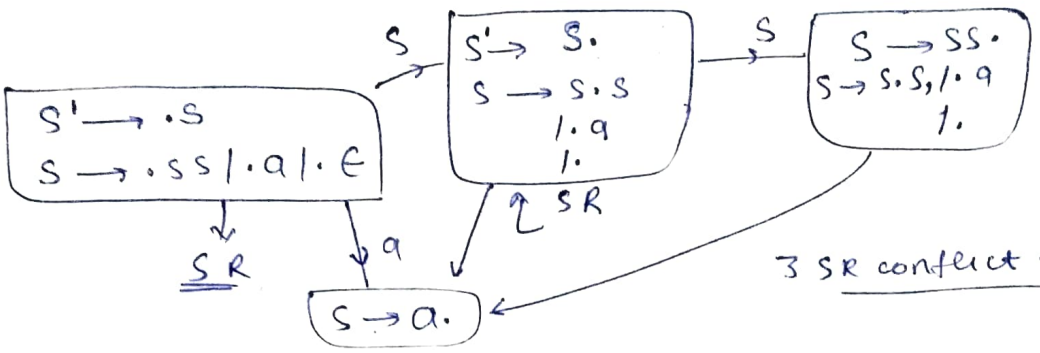
d) CLR(1) but not in LALR(1)

→ Grammar is LL(1), LR(0), SLR(1), CLR(1) and LALR(1) all of them.

find the number of SR and RR conflicts in dfa with LR(0) items.

$S \rightarrow SS/a/\epsilon$

Both (SR and RR)



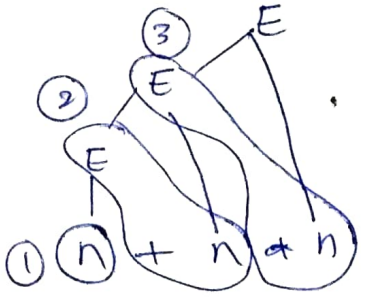
3 SR conflict + 1 RR conflict

→ Total 4 conflicts are generated

Q $E \rightarrow E+n / E*n / n.$

For the string $n+n*n$, the handles in right sequential form of reduction are-

Right sequent form: Bottom-up parsing.



Handles means the element that is being reduced.

→ firstly n is reduced so first handle is 'n'

then we reduced (E+n) to E → 'E+n' is second handle

then we reduced (E*n) to E → 'E*n' is another handle

Option a) $n_1, E+n, E+n+n$

b) $n_1, E+n, E+E*n$ ✗

c) $n_1, n+n, n+n+n$

d) $n_1, E+n, E*n$ ⟶ correct one

Q $S \rightarrow (S) \mid a$

SLR(1) : n_1, [CLR(1) or LR(1)] : n_2, LALR(1) : n_3

n_1, n_2, n_3 are no. of states then,

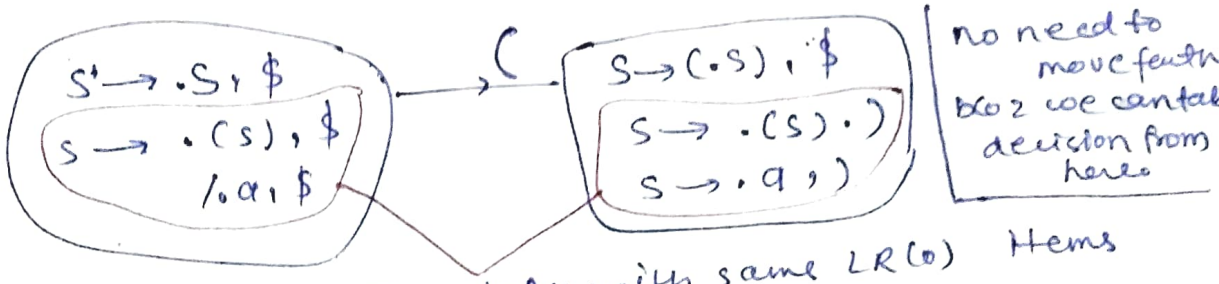
a) $n_1 < n_2 < n_3$

b) $n_1 = n_3 < n_2$ ✓

c) $n_1 = n_2 = n_3$

d) $n_1 \geq n_3 \geq n_2$

Soln



Two states with same LR(0) Items but different look aheads in LR(1). So they will reduce to combine in LALR(1) table so

Size of LR(1) table > size of LALR(1).

$n_2 > n_3$

and size of LALR(1) = size of SLR(1)

$n_3 = n_1$

$$n_1 = n_3 < n_2$$

→ Ambiguous grammar can't use in parsing procedure bcoz in canonical structure all the final states will get conflicts but if somehow we can remove those conflicts meaningfully then we can use that grammar for parsing.

YACC: Yet Another Compiler Compiler → O/P [LALR(1)]

↳ It resolve both SR and RR conflicts

→ SR conflict resove by shift move

→ RR conflict resolve by first reduce

Syntax direct translation (SDT)

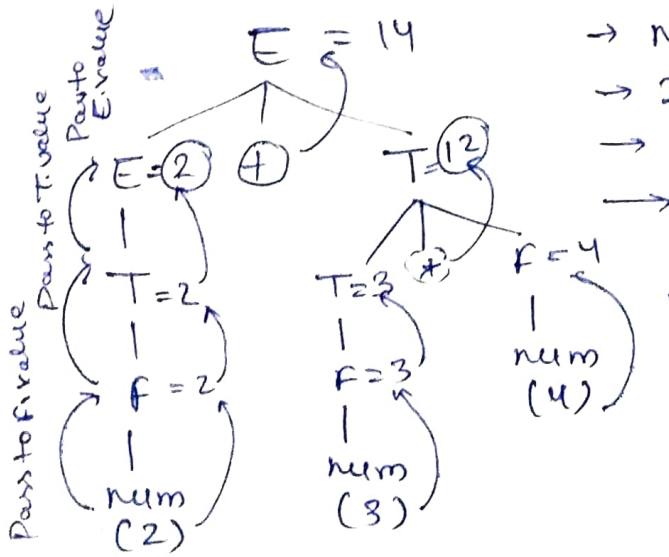
Syntax directed translations examples

Grammar + {Semantic rules} = SDT

SDT for evaluation of expression

$E \rightarrow E+T$	{ E.value = E.value + T.value }	(1)
$/ T$	{ E.value = T.value }	(2)
$T \rightarrow T*F$	{ T.value = T.value * F.value }	(3)
$/ F$	{ T.value = F.value }	(4)
$F \rightarrow num$	{ F.value = num.value }	(5)

Take a expression 2+3*4



→ num → token

→ 2, 3, 4 → lexical values

→ value → attribute

→ E, T, F → variables

→ In SDT every variable can have any number of attributes from 0 to n.

→ Bottom-up Parsing.

② $E \rightarrow E+T \mid T$ { Printf ("+"); } ①
{ } ②

$T \rightarrow T*F$ { Printf ("*"); } ③

$\mid F$ { } ④

$F \rightarrow num$ { Printf (num.lval); } ⑤

Production + Semantic rule = semantic action and we give no. to them

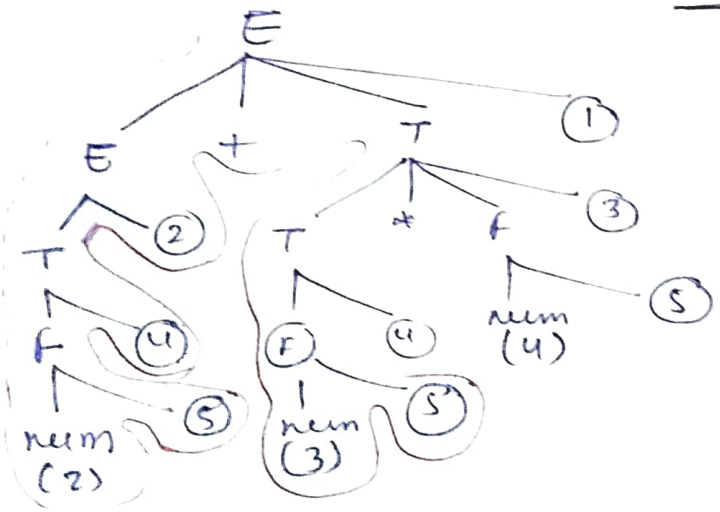
Above SDT is used to convert Infix to Postfix.

2 + 3 * 4 (Infix)

Postfix → 2 3 4 * +

→ Using top-down parser

we perform operation top → down and left to right and whenever we look semantic action we performed it.



o/p 2 3 4 * +

⑤ ④ ⑤ ④ ⑤ ③ ①

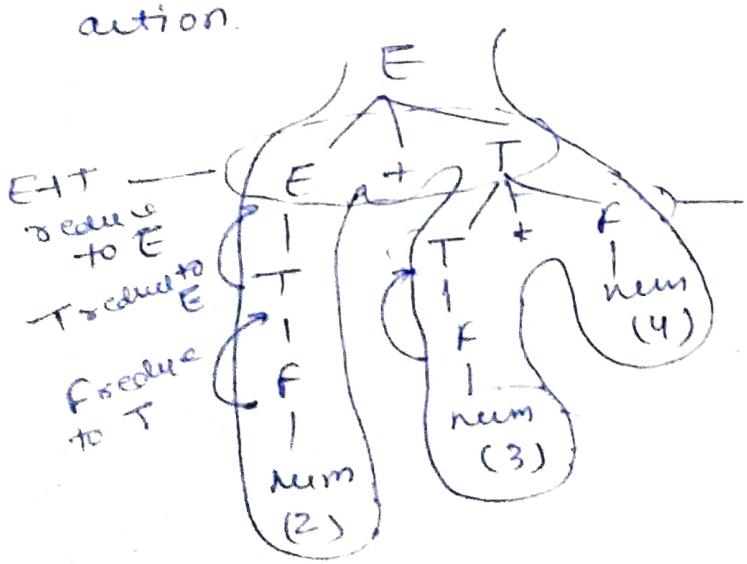
②

semantic action number and corresponding o/p

→ bottom-up parser

Mainly focused on reduction.

for reduction they take production and perform action.



o/p 2 3 4 * +

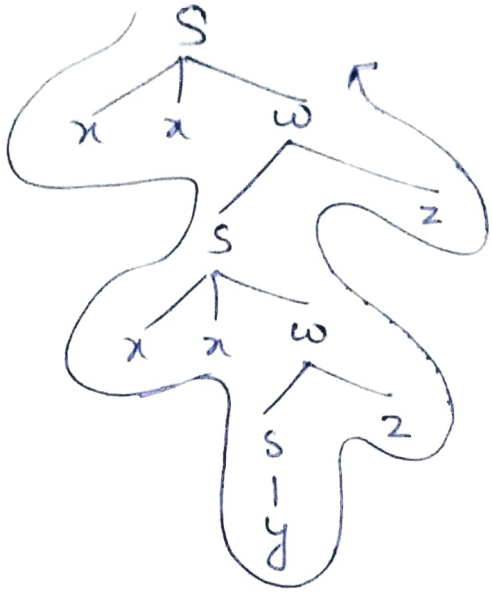
T * F reduce to T

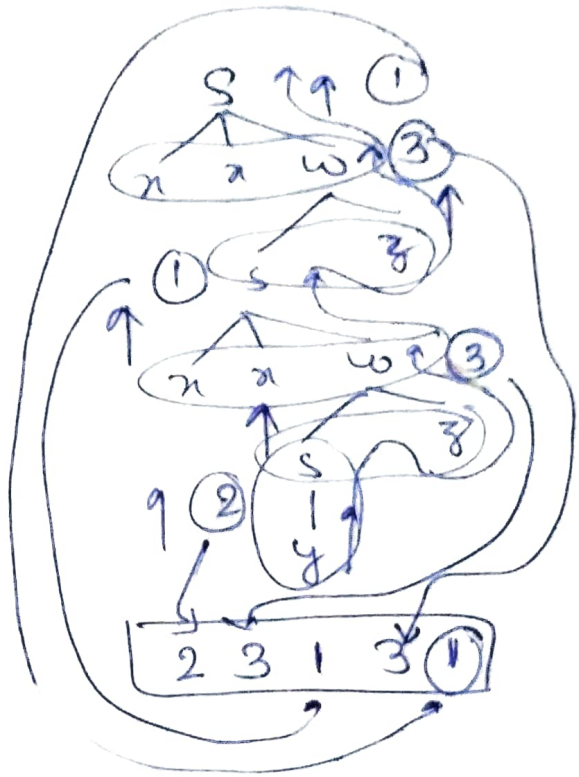
→ we get same output so SDT can be use for both top down and bottom up parsers.

(2) $S \rightarrow xxw$ { Printf(1); }
$\quad / y$ { Printf(2); }
$w \rightarrow Sz$ { Printf(3); }

String : xxxxyzz

→ bottom up parser (LR).





(4) $E \rightarrow E + T$
$\quad / T$
$T \rightarrow F - T$
$\quad / F$
$F \rightarrow 2$
$\quad / 4$

Evaluate : 4 − 2 − 4 * 2

Observation from Grammar

(1) ⊖ have more precedence than ⊕ bcoz it comes at lower level.

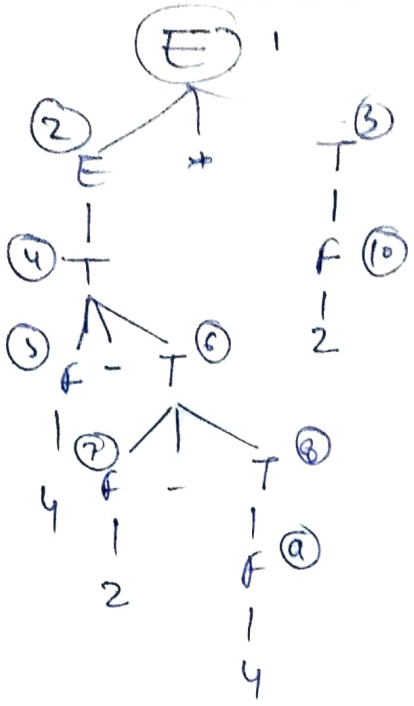
(2) ⊖ have Right to left predecence because Grammar production T is Right recursive.

So $((4 - (2-4)) * 2)$

-2

6

12 Ans.



No. of reductions = 10

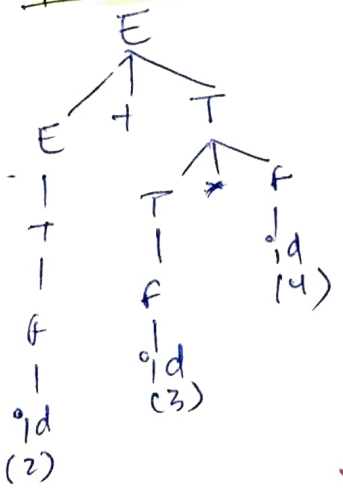
All the non-leafs are reduction

Note → Left associative means left to right say that grammar is left recursive. and same for right associative too.

Example of SDT

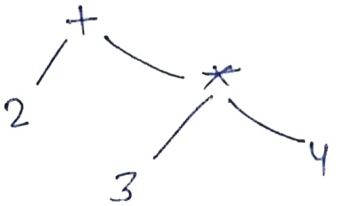
2 + 3 * 4

Concrete syntax tree or parse tree



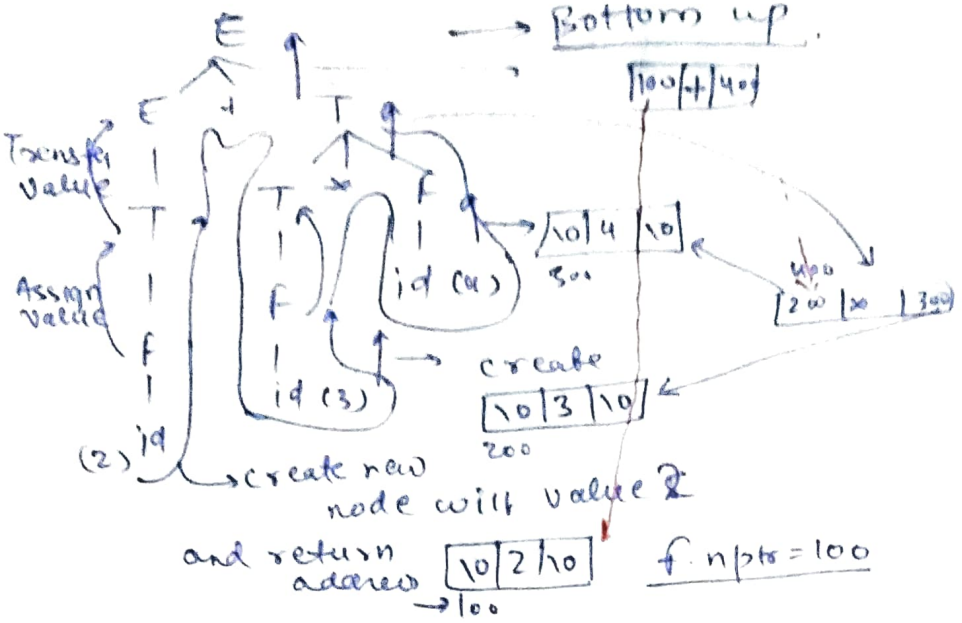
Abstract syntax tree or parse tree

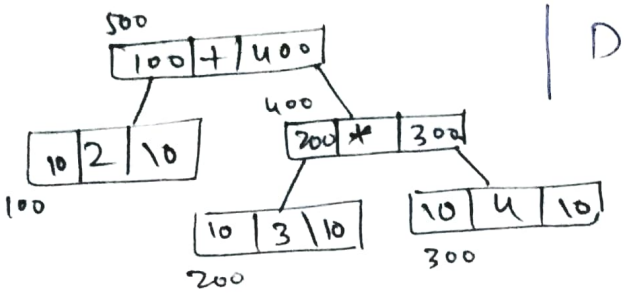
Expression tree



STD to built abstract syntax tree

$E \rightarrow E + T$	{ E.nptr = mknode (E1.nptr, '+', T.nptr); }
$\mid T$	{ E.nptr = T.nptr; }
$T \rightarrow T * f$	{ T.nptr = mknode (T1.nptr, *, f.nptr }
$\mid f$	{ T.nptr = f.nptr }
$f \rightarrow id$	{ F.nptr = mKnode (null, id.name, null); }





Data structure of abstract parse tree

STD for type checking

E → E1 + E2 { if((E1.type == E2.type) && (E1.type = int)) then E.type = int else error; }

| E1 == E2 { if ((E1.type == E2.type) && (E1.type = int / boolean) then E.type = boolean else error; }

| (E1) { E.type = E1.type; }

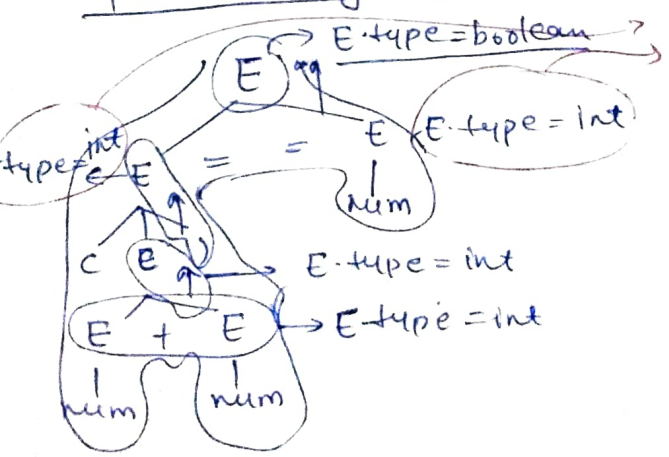
| num { E.type = int; }

| True { E.type = boolean; }

| false { E.type = boolean; }

⇒ Both the the E1 and E2 are same as E but just for separation btw LHS and RHS we make this.

operation string (2+3) == 8



So top most E will get E.type = boolean

	Count 1's	Count 0's
$N \rightarrow L$	{ N.count = L.count }	→ same
$L \rightarrow L_1 B$	{ L.count = L_1.count + B.count }	→ same
$/ B$	{ L.count = B.count }	→ same
$B \rightarrow 0$	{ B.count = 0 }	{ B.count = 1 }
$/ 1$	{ B.count = 1 }	{ B.count = 0 }

N → Number L → List of bits B → Bits

Convert binary to decimal using above grammar:-

MSB 1011 LSB

1 → 1x2+0 → 2x2+1 → 5x2+1 → 11

Calculate or count of bits before LSB, multiply it with 2 and add LSB.

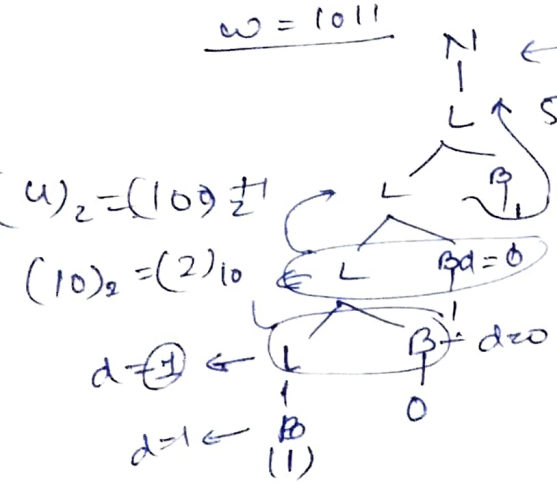
$N \rightarrow L$	{ N.dval = L.dval }
$L \rightarrow L_1 B$	{ L.dval = L_1.dval * 2 + B.dval }
$/ B$	{ L.dval = B.dval }
$B \rightarrow 0$	{ B.dval = 0 }
$/ 1$	{ B.dval = 1 }

L

1100 | 1

L_1 B

w = 1011



Binary no. contain decimal

$(11.01)_2 \rightarrow (3.25)_{10}$

$2^{-2} \times 1$

$2^{-1} \times 0$

$(11)_2 \rightarrow (3)_d$

$(.11)_2 \rightarrow \frac{3}{2^2} = 0.75$

$N \rightarrow L_1 . L_2$ { N.val = L_1.dval + (L_2.dval / $2^{L_2.count}$) }

$L \rightarrow L_1 B$ { L.count = L_1.count + B.count ; L.dval = L_1.dval + B.dval }

$/ B$ { List.count = B.count , L.dval = B.dval }

$B \rightarrow 0$ { B.count = 1 , B.dval = 0 }

$/ 1$ { B.count = 1 , B.dval = 1 }

Both L_1 and L_2 are same

SDT to generate 3-address code

S → id = E { gen (id.name = E.place) }

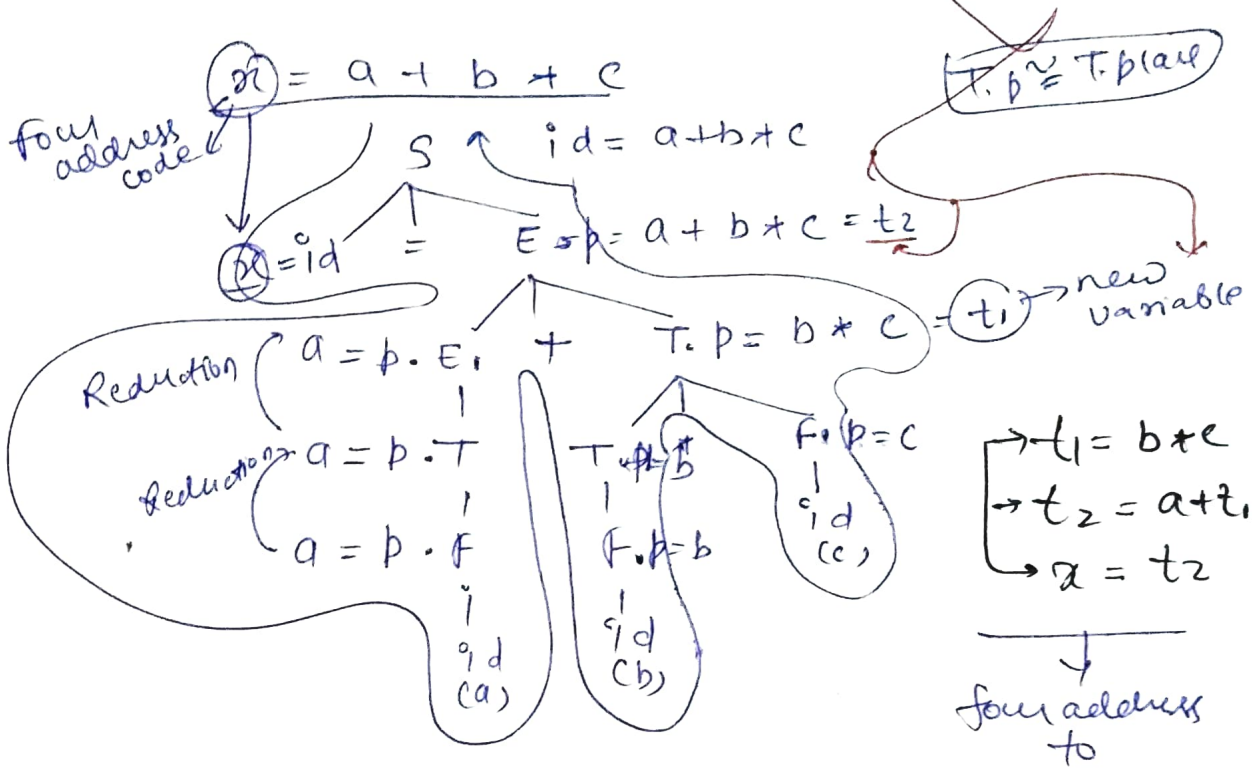
E → E1 + T { E.place = newtemp() ; gen (E.place = E.place + T.place) }

| T { E.place = T.place }

T → T1 * F { T.place = newtemp() ; gen (T.place = T1.place * F.place) }

| F { T.place = F.place }

F → id { F.place = id.name }



Classification of SDT and attributes

Synthesised attribute

Parent ← A → BCD → child

A.att = f (B.att , C.att , D.att). means parent is taking attribute from children.

Inherit Attribute

A → B C D

C.att = f (A.att , B.att , D.att) here 'C' can be replaced with B and D only means child in inhering attribute from parent.

S – attributed SDT

① Use only synthesized attributes

② Semantic actions are placed at right end of productions.

$A \rightarrow BCC \{ \}$

↓

also called as postfix SDT

③ Attributes are evaluated using bottom up parsing

L-attributed SDT

① Can have both kind of attribute but inherit only from parent or left siblings

Ex $A \rightarrow XYZ$

$\{Y.S = A.S, Y.S = X.S, Y.S \neq Z.S\}$

② Placed anywhere on RHS

$A \rightarrow B \{ \}$
$\quad / C \{ \} D$
$\quad / \{ \} E$

③ Traverse parse tree depth first, left to right.

Ex $A \rightarrow LM \{ \underbrace{L.i = A.i}_{\text{① Inherit}}, \underbrace{M.i = L.S}_{\text{② L attribute}}, \underbrace{A.S = M.S}_{\text{③ L attribute}} \}$

Not S

combiningly is it L attribute SDT

$A \rightarrow QR \{ \underbrace{R.i = A.i}_{④ L}, \underbrace{Q.i = R.i}_{⑤ \text{Not-L}}, \underbrace{A.S = Q.S}_{⑥ L} \}$

Due to ① it is not S attributed and due to ⑤ it is not L attributed. So it is neither L attributed or neither S attributed

SDT to store type information in symbol table

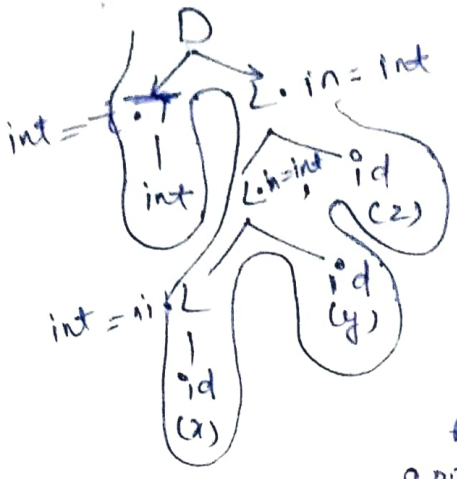
$D \rightarrow TL \{ L.in = T.type \}$

$T \rightarrow int \{ T.type = int \}$
$\quad / char \{ T.type = char \}$

$L \rightarrow L_1, id \{ L_1.in = L.in, addtype(id.name, L.in); \}$
$\quad / id \{ addtype(id.name, L.in) \}$

Above grammar is L-attributed grammar.

Ex → int x, y, z



(evaluate first visit)

Visit on each id

x	int
y	int
z	int

When you came across any inherited attribute evaluate first time when you visit it and across synthesized attribute evaluate last time when you visited.

All the L are inherited attribute so apply the action on first visit.

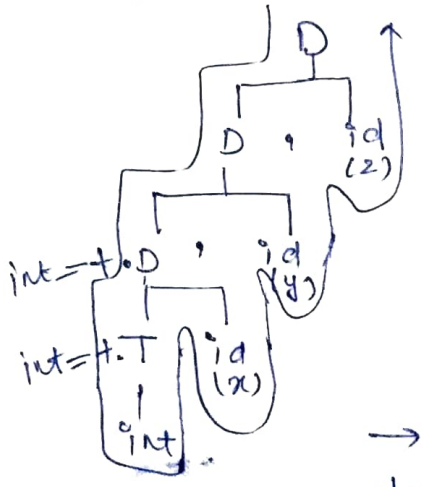
S → attributed SDT (Only synthesized attributes)

D → D_1 , id { addtype (id.name , D_1.type), D.type = D_1.type }
/ T, id { addtype (id.name , T.type), D.type = T.type }

T → int { T.type = int }
/ char { T.type = char }

Both D and D_1 are same

Ex → int x, y, z



x	int
y	int
z	int

On each visit to id we add T.type and id.name in table.

→ In S-attribute we will do action on last visit to the attribute.

⇒ To convert a SDT from L attributed to S-attributed we need to change grammar and semantics action both.