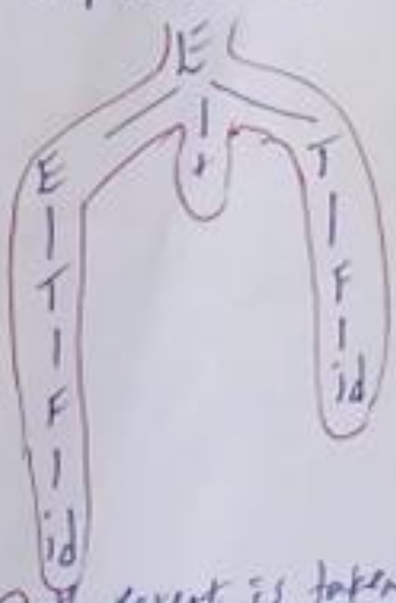
Types of SDT

S - Attributed SDT

- Based on Synthesized Attribute
- Use Bottom up Parsing
- Semantic Rules always written at Rightmost position in RHS

Example :-
Grammar
$E \to E+T \mid T$
$T \to F$
$F \to id$
input string :- id+id



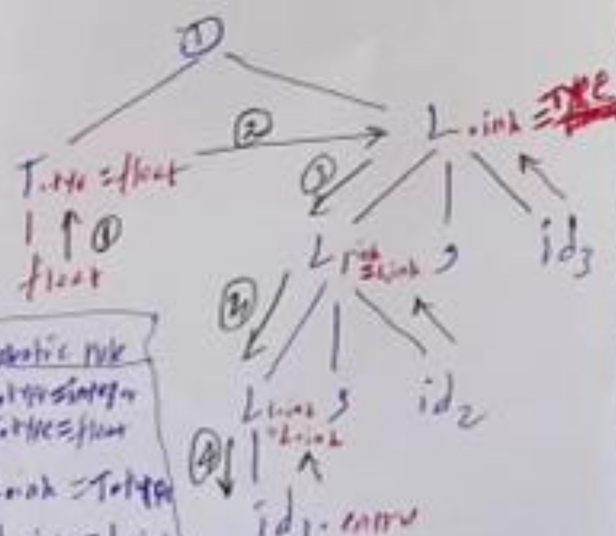
(*) The parent is taken the value from children it is called S-attributes

L - Attributed SDT

- Based on Both Synthesized and inherited attributes
- Top Down parsing
- Semantic Rules anywhere in RHS
- no cyclic dependency

D: Declaration
T: Data type (int, float)
L: List of identifiers or identifier

Example :-
Grammar
$D \to TL$
$T \to int \mid float$
$L \to L_1, id \mid id$
input string :- float id,id,id



Production	Semantic rule
T → int	T.type = integer
T → float	T.type = float
D → TL	L.inh = T.type
L → L₁,id	L₁.inh = L.inh Addtype(id.entry, L.inh)
L → id	[illegible] (id.entry, L.inh)

(*) The children is taken value from parent called L-attributes.

Add type()

- id.entry is a lexical value that points to the symbol table.
- L.inh is the type being assigned to every identifier in the list
- The function installs L.inh as the type of corresponding identifier.

Conflicts in Syntax Analysis

* Bottom-up (LR-family)
  1) shift-reduce conflict
  2) reduce-reduce conflict
  3) conflict from insufficient lookahead (SLR / LALR vs LR(1))

* Top-down (LL-family)
  4) first-first conflict
  5) first-follow - "
  6) left-recursion "
  7) top-down ambiguity "

* Grammar level ambiguity
  8) inherent ambiguity conflict

* Lexer-parser interface
  9) token ambiguity

LALR(1), final table

States	Action			Goto	
	c	d	$	S	C
I_0	s36	s47		1	2
I_1			Accept		
I_2	s36	s47			5
I_{36}	s36	s47			89
I_{47}	r3	r3	r3		
I_5			r1		
I_{89}	r2	r2	r2		

10 reduce
⇓ 7 States

LALR (1)

CLR(1) / LALR(1) → look a head value

⇓

LR(1) items

⇓

LR(0) items + look a head value

$E \to BB$
$B \to cB / d$ } ⇒ $E' \to \cdot E, \$$
$E \to \cdot BB, \$$
$B \to \cdot cB / \cdot d, c/d$ } LR(1) items.

⇒ $I_3, I_6 \Rightarrow I_{36}$
⇒ $I_4, I_7 \Rightarrow I_{47}$
⇒ $I_8, I_9 \Rightarrow I_{89}$
} same production & look a head value is different

LALR (1) parsing table

State	Action			Goto	
	c	d	$	1	2
I_0	s36	s47			
I_1			Accept		
I_2	s36	s47			5
I_{36}	s36	s47			89
I_{47}	r_3	r_3			
I_5			r_1		
I_{36}	s36	s47			89
I_{47}			r_3		
I_{89}	r_2	r2			
I_{89}			r_2		

CLR(1) Parsing Table

State	Action			Goto	
	c	d	$	E	B
I_0	S3	S4		1	2
I_1			Accept		
I_2	S6	S7			5
I_3	S3	S4			8
I_4	r3	r3			
I_5			r1		
I_6	S6	S7			9
I_7			r3		
I_8	r2	r2			
I_9			r2		

CLR(1) and LALR(1)

Canonical Collection of

LR(1) items => LR(0) item + Look a head

LR(0)	SLR	CLR & LALR
⇓	⇓	⇓
reduce is written in full row	reduce is written in follow of l(P)	reduce is written only on look a head

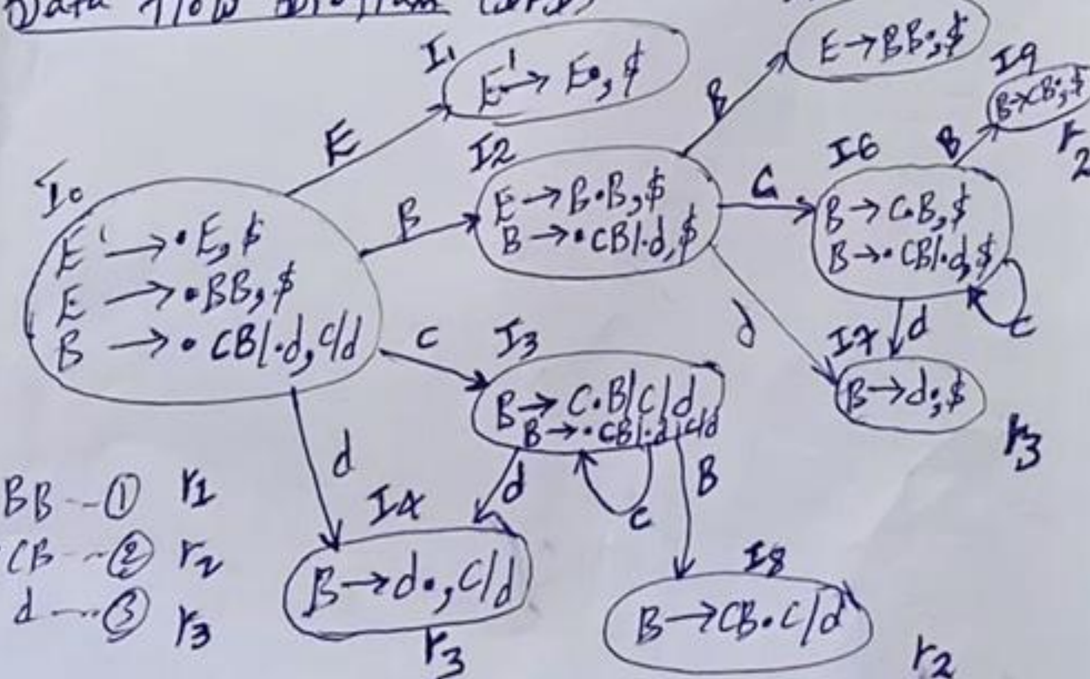
Example :-

$E \rightarrow BB$
$B \rightarrow cB \mid d$

* Augment grammar & LR(1) items

$E' \rightarrow \cdot E, \$$ ← look a head item
$E \rightarrow \cdot BB, \$$ ← look a head item
$B \rightarrow \cdot cB \mid \cdot d, c/d$ ← look a head item

Data Flow Diagram (DFD)

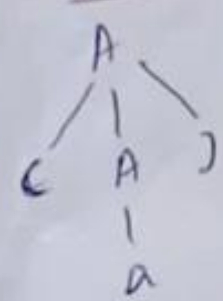


$E \rightarrow BB$ --- ① r_1
$B \rightarrow cB$ --- ② r_2
$B \rightarrow d$ --- ③ r_3

i/p string parsing:- (a)$

step	parsing stack	i/p	Action
1	$0	(a)$	Shift 2
2	$0(2	a)$	Shift 3
3	$0(2a3	)$	Reduce r_2 $A \rightarrow a$
4	$0(2A4	)$	Shift 5
5	$0 $\boxed{(2A4)}$ 5	$	Reduce r_1 $A \rightarrow (A)$
6	$0A1	$	Accept

Parse tree



$\Rightarrow (a)$

SLR (1) Parsing

LR (0), SLR (1) → LR(0) items
CLR (1), LALR(1) → LR(1) items

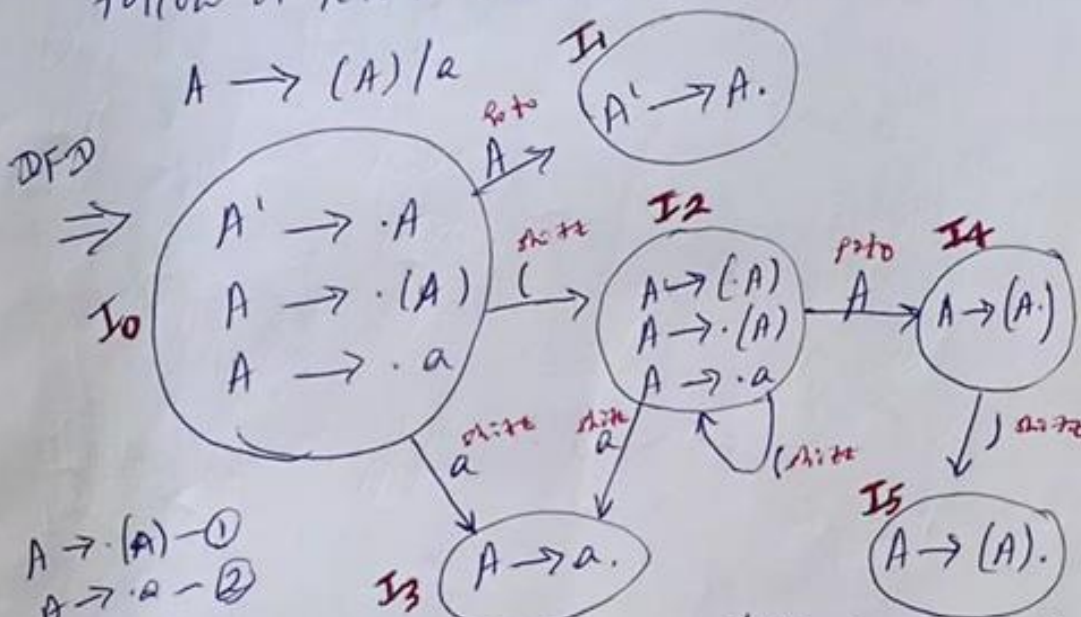
⇓

Simple LR

⇒ Works on smallest class of grammar
⇒ Few no: of states
⇒ Simple & fast to reconstruct

* In SLR we place the reduce move only in the follow of left hand side not to entire row.

A → (A)/a



A → ·(A) — ①
A → ·a — ②
r2

Follow (A) = {), $}

Parsing Table

State	Action				goto
	a	(	)	$	A
0	s3	s2			1
1				Accept	
2	s3	s2			4
3			r2	r2	
4			s5		
5			r1	r1	

r → reduce

step-7:- Construct parsing Table LR(0)

states	Action (terminals)			Goto	
	a	b	$	A	S
I_0	S3	S4		2	1
I_1	Accept	Accept	Accept		
I_2	S3	S4		5	
I_3	S3	S4		6	
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

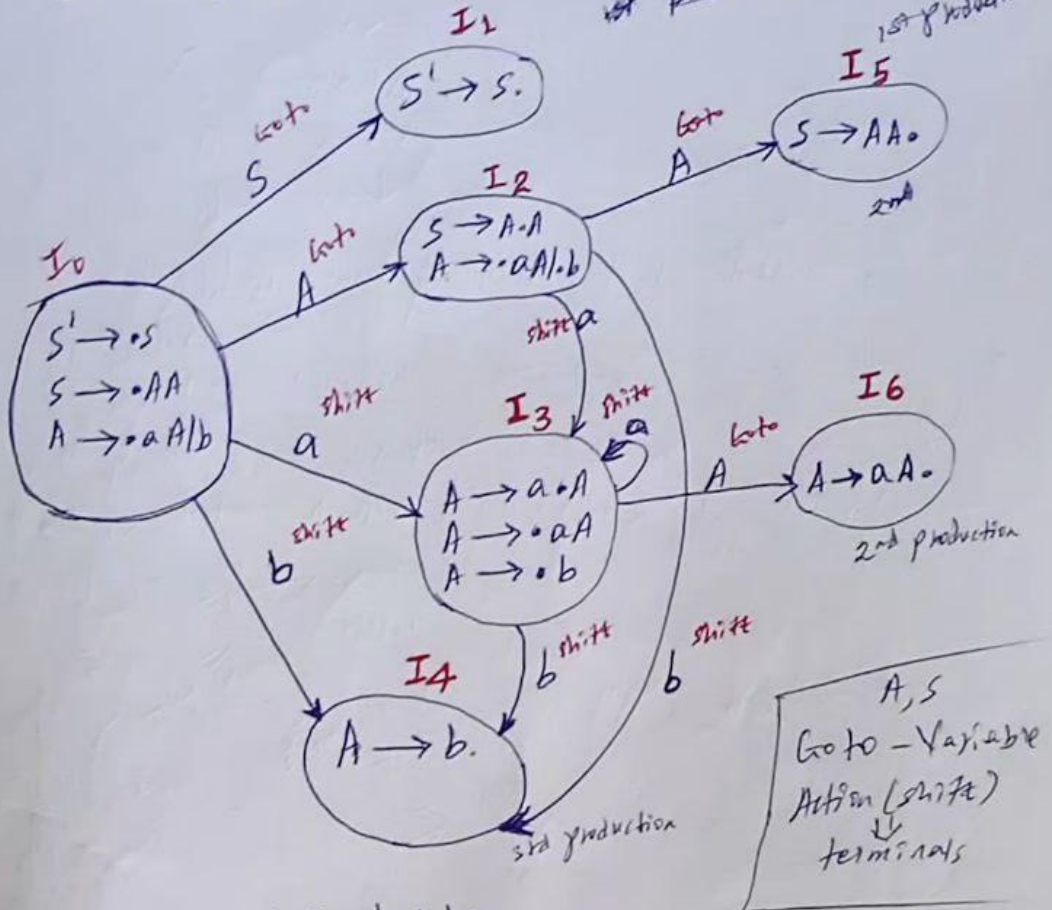
I4, I5, I6 all contains final states

$S \rightarrow AA$ — ①
$A \rightarrow aA$ — ②
$A \rightarrow b.$ — ③

LR (0) parsing

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

Drawing DFA { Contains 7 states I0 to I6 }



Step 6 :- LR (0) table

* If a state is going to some other state on a terminal it is correspond to a shift move.
* If a state is going to some other state on a Variable it is correspond to goto move
* If a state contain the final item in the particular row then write the reduce node completed.

S^*

$S' \longrightarrow \cdot S$
$S \longrightarrow \cdot AA$
$A \longrightarrow \cdot aA$
$A \longrightarrow \cdot b$

{ Add Agument production to I_0 state & compute closure

I_0 = closure { $S' \rightarrow \cdot S$ }

all productions starting with S into I_0 state b/s "•" is followed non terminal. So I_0 state becomes

$I_0 = S' \rightarrow \cdot S$
$S \rightarrow \cdot AA$

all productions starting with "A" in modified I_0 state b/s

State

$I_1 = Goto(I_0, S) = closure(S' \to \cdot S.) = S' \to S.$

Here, the production is reduced so close the stat

State

$I_2 = Goto(I_0, A) = closure(S \to A \cdot A)$

Add all productions starting with A into I_2 states

b/s "." follow

Non terminal. So, the I_2 state becomes,

$$I_2 = \begin{array}{l} S \to A \cdot A \\ A \to \cdot aA \\ A \to \cdot b \end{array}$$

$Goto(I_2, a) = closure(A \to a \cdot A)$

states

I_0 :- $S' \to \cdot S$
$S \to \cdot AA$
$A \to \cdot aA$
$A \to \cdot b$

goto (I_0, S)

I_1 :- $S' \to S.$

goto (I_0, A)

I_2 :- $S \to A \cdot A$
$A \to \cdot aA$
$A \to \cdot b$

goto (I_2, a)

I_3 :- $A \to a \cdot A$
$A \to \cdot aA$
$A \to \cdot b$

goto (I_2, b)

I_4 :- $A \to b.$

goto (I_2, A)

I_5 :- $S \to AA.$

goto (I_3, A)

I_6 :- $A \to aA.$

LR(0) Parsing :-

Various steps involved in parsing

→ for the given input string write context free grammer
→ Check ambiguity of the grammer
→ Add augment production in the given grammer
→ create Canonical Collection of LR(0) items
→ Draw a data flow diagram (DFA)
→ Construct LR(0) parsing table.

E.g :-

$S \rightarrow AA$
$A \rightarrow aA \mid b$

$$\left.\begin{array}{l} \boxed{S' \rightarrow S} \\ S \rightarrow AA \\ A \rightarrow aA \\ A \rightarrow b \end{array}\right\} \text{Augment production}$$

Step-3 :- Add Augment production

$$\left.\begin{array}{l} S' \rightarrow S \\ S \rightarrow AA \\ A \rightarrow aA \mid b \end{array}\right\} \text{given grammer.}$$

Step-4 :- Create Canonical Collection of LR(0) items

* An LR(0) items is a production G with dot a some position the right hand side of production.
* LR(0) items is useful to indicate that how much of there has been scanned up to a given point in the process of parsing

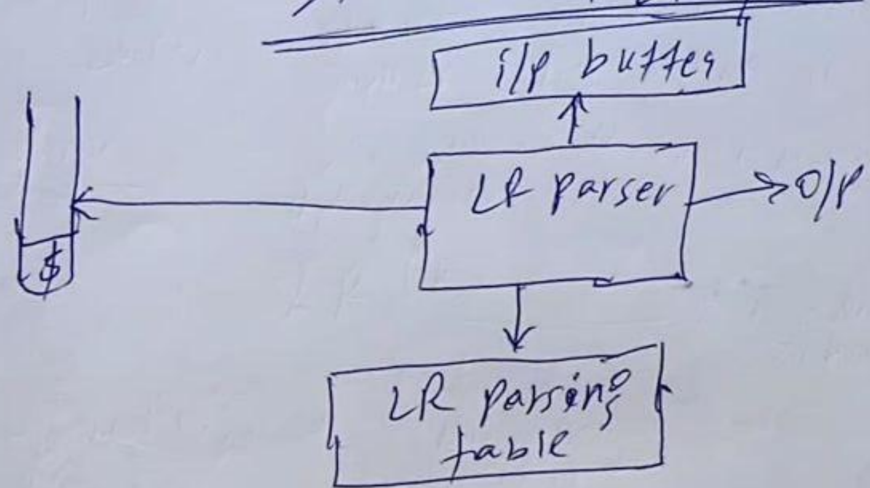
$$\boxed{\begin{array}{l} S' \rightarrow \cdot S \text{ — LR(0) item} \\ S \rightarrow \cdot AA \\ A \rightarrow \cdot aA \mid \cdot b \end{array}}$$

* In LR(0), we place the reduce node in entire row.

3) LALR (1) :- Look A head LR parser.

* Works on intermediate size of grammer
* no of states are same as SLR(1)

Structure of LR Parser



LR(0) –
SLR(1) –
LALR(1) –
} the only difference is parsing table

* To construct LR(0) and SLR(1) tables we use Canonical collection of LR(0) items.
* To construct LALR(1) & CLR(1) tables we use Canonical Collection of LR(1) items.

Parsers - Compiler Design

LR Parser

Least powerful — most powerful

- LR(0)
- SLR(1) Simple LR
- LALR(1) Look a head LR
- CLR Canonical LR

LR parser :- ⇒ Non recursive shift reduce bottom up parser.

LR (K) — look a head symbol

- L: Left to right scanning input string
- R: Construction of right most derivation in reverse

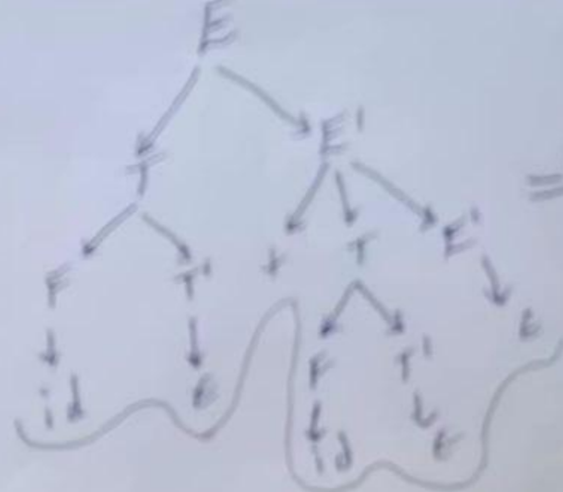
1) SLR (1) :- Simple LR Parser

* Work on smallest class of grammer
* Few no of states
* Simple & fast Construction.

2) CLR (1) :- LR parser

* Works on Complete set of LR (1) grammer
* large no of states
* slow Construction

Step-4:- parse tree



$\Rightarrow id + id$

step-2 :- parsing table

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$					

step-3 :- stack Implementation

i/p = id+id

stack	i/p	Action
E$	id+id$	$E \to TE'$
TE'$	id+id$	$T \to FT'$
FT'E'$	id+id$	$F \to id$
~~id~~T'E'$	~~id~~+id$	pop id
T'E'$	+id$	$T' \to \epsilon$
E'$	+id$	$E' \to +TE'$
~~+~~TE'$	~~+~~id$	pop +
TE'$	id$	$T \to FT'$
FT'E'$	id$	$F \to id$
~~id~~T'E'$	~~id~~$	pop+
T'E'$	$	$T' \to \epsilon$
E'$	$	$E' \to \epsilon$
$	$	Accept

Construction of predictive parser LL(1)

Example 2 :-

$$E \rightarrow E+T / T$$
$$T \rightarrow T*F / F$$
$$F \rightarrow (E) / id$$

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' / \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' / \epsilon$$
$$F \rightarrow (E) / id$$

1. first follow
2. parsing table
3. stack implementation
4. parse tree

step - 1 :- First Function

first (F) = { (, id }
first (T') = { *, ϵ }
first (T) = { (, id
first (E') = { +, ϵ }
first (E) = { (, id }

Follow function

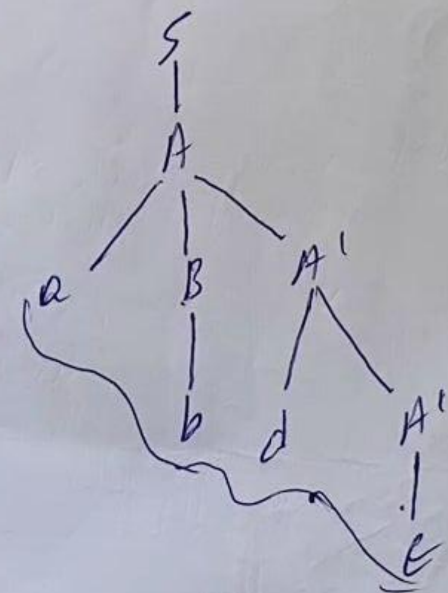
Follow (E) = { \$,) }
Follow (E') = { \$,) }
Follow (T) = { first (E') - ϵ $\cup$ Follow (E) }
= { +, \$,) }
Follow (T') = { +, \$,) }
Follow (F) = { +, *,), \$ }

Step 3 :- Stack Implementation by using parse table

Input string: abd$

Stack	input	Production
$S\$$	$abd\$$	$S \rightarrow A$
$A\$$	$abd\$$	$A \rightarrow aBA'$
$\cancel{a}BA'\$$	$\cancel{a}bd\$$	pop a
$BA'\$$	$bd\$$	$B \rightarrow b$
$\cancel{b}A'\$$	$\cancel{b}d\$$	pop b
$A'\$$	$d\$$	$A' \rightarrow dA'$
$dA'\$$	$d\$$	pop d
$A'\$$	$\$$	$A' \rightarrow \epsilon$
$\$$	$\$$	Accept {the input is properly parsed}

Step 4 :- Generate parse tree using implementation following top down approach.



string abdε