Unit-IV (Phases of 2nd, 3rd & 4th)

Syllabus

I- Syntax Analysis :

Introduction to types of Passing Techniques
Top-down parsing.
Recursive decent Parsing.
Predictive parsing : LL(1).
Bottom-up Parsing : SLR, CLR, LALR

II Semantic Analysis (SA)

Introduction
syntax directed definition (SDD).
syntax directed Translation
Attributes, Types of attributes
Bottom-up evaluation of attributes.

III- Intermediate code Generation (ICG)

Types of Intermediate code
Types of 3 address codes
Evaluation of 3-address code.

Unit-IV:

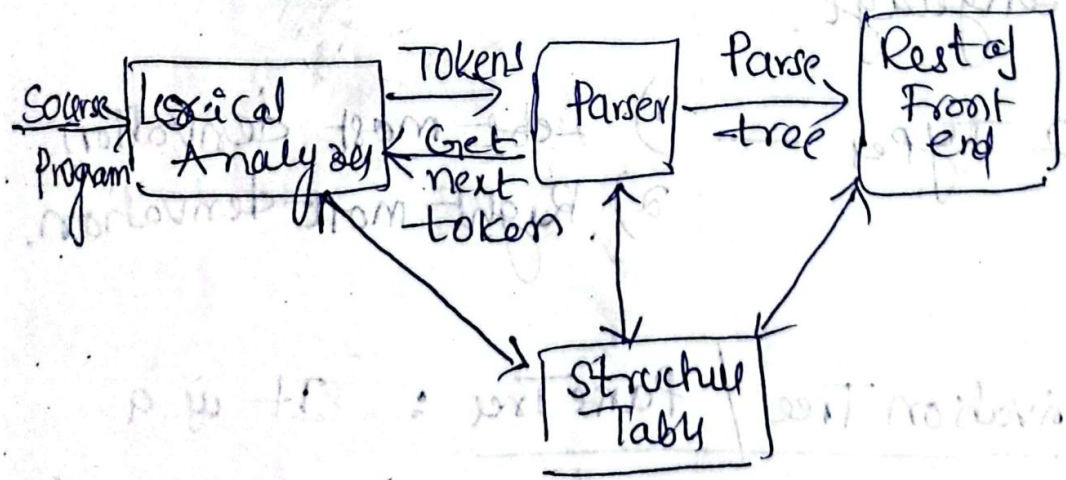
Chapter-III. Syllabus :. (II phase of compiler)

Syntax Analysis : Types of parsing, Recursive Decent Parsing, Predictive Parsing, Bottomup parsing SLR, CLR & LALR.

Syntax Analysis is the second phase of the compiler.

It can also be called syntax verification & parsing & Parse tree generator.

The parser construct the parse tree and passes to the rest of the compiler for further processing.



Context Free Grammar

VI

CFG has 4 components $G = \{V, T, P, S\}$

V — Set of Non-Terminals / Variables in uppercase letters

T — Set of Terminals, $\{V \cap T = \emptyset\}$ Represented in Lowcase / digits

P — Production Rules

S — Start symbol.

Derivation: The process of deriving a string / sentence from a grammar is called derivation.

It generates strings/sentence of Language.

2 Types — 1) Left most derivation
2) Right most derivation.

Derivation Tree / Parse Tree: It is a Graphical representation for the derivation of the production rules for a given CFG.

Left most DT

Derivation from left to right —

Q: $E \to E+E$
$E \to E*E$
$E \to a/b/c$

derive the string $a+b*c$

Terminals = (+, *, /, a, b, c)

NonTerminal = E

Parse tree

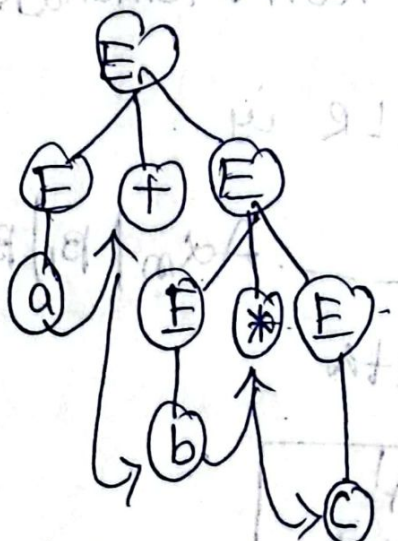
$E \to E+E$

$a+E$

$a+E*E$

$a+b*E$

$a+b*c$



Read left to right

$a+b*c$

Right most DT

Derivation from right to left —

$E \to E+E$

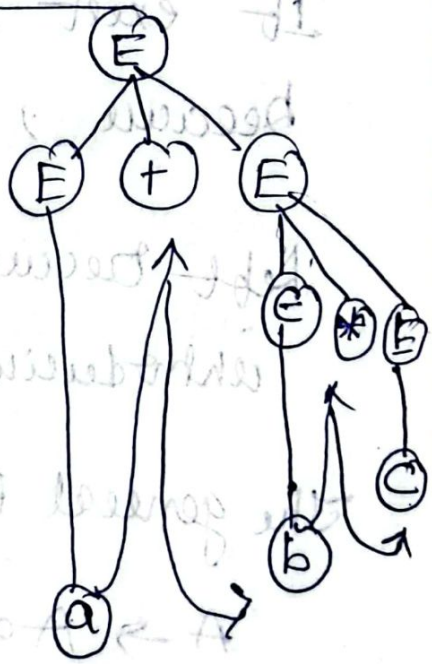
$E \to E+E*E$

$E \to E+E*c$

$E \to E+b*c$

$E \to a+b*c$

Parse tree



Read from Left to Right —

$a+b*c$

Ambiguous Grammar :

A CFG is said to be ambiguous if there exist more than one derivation tree for given input string.
i.e more than one left most derivation or more than one right most derivation

Removing left recursion (LR)

Left recursion occurs during parsing
It exist, remove from grammar because ; it can create infinite loop

Left recursion can be eliminated by introducing new non-terminal.

The general form of LR is

$A \rightarrow A\alpha_1 / A\alpha_2 \dots A\alpha_m / \beta_1 / \beta_2 \dots$

can be replace with

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' / \varepsilon$$

D.L. Bhargavi

Ex: $E \to E+T / T$

$A \quad \alpha \quad \beta$

$A \to A\alpha / \beta$ then

Sol: $\begin{cases} A \to \beta A' \\ A' \to \alpha A' / \varepsilon \end{cases} \Rightarrow \begin{matrix} E \to TE' \\ E' \to E+T / \varepsilon \end{matrix}$

Left Factoring (LF)

Left Factoring transforms the grammar to make it useful for Top-down parsers or predictive parsers

or

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for top down parsers.

Grammar

$A \to \alpha\beta_1 + \alpha\beta_2 + \cdots / \gamma_1 / \gamma_2 \cdots$

The equalance Left Factoring Grammer is

$$\boxed{\begin{matrix} A \to \alpha A' / \gamma \\ A' \to \beta_1 / \beta_2 / \cdots / \beta_n \end{matrix}}$$

Ex1: $S \to iEts / iEtses / a$

$E \to b$

Soln: $S \to iEts / iEtses / a$

(common prefix = iEts)

$A \to \alpha A' / \gamma$

$A' \to \beta_1 / \beta_2 - \beta_n$

Left Factoring ⟹

$S \to iEtsS' / a$

$S' \to \epsilon / es$

$E \to b$

Ex: 2 : $A \to aAB / aA$

$B \to bB / b$

Soln:

$A \to aAB / aA$ ⟹ (Left Factoring) $A \to aA'$

$A' \to AB / A$

$B \to bB / b$ ⟹ (Left Factoring) $B \to bB'$

$B' \to B / \epsilon$

Ex: 3) $A \to aAB / aBc / aAc$

Soln: $A \to aA'$

$A' \to AB / Bc / Ac$

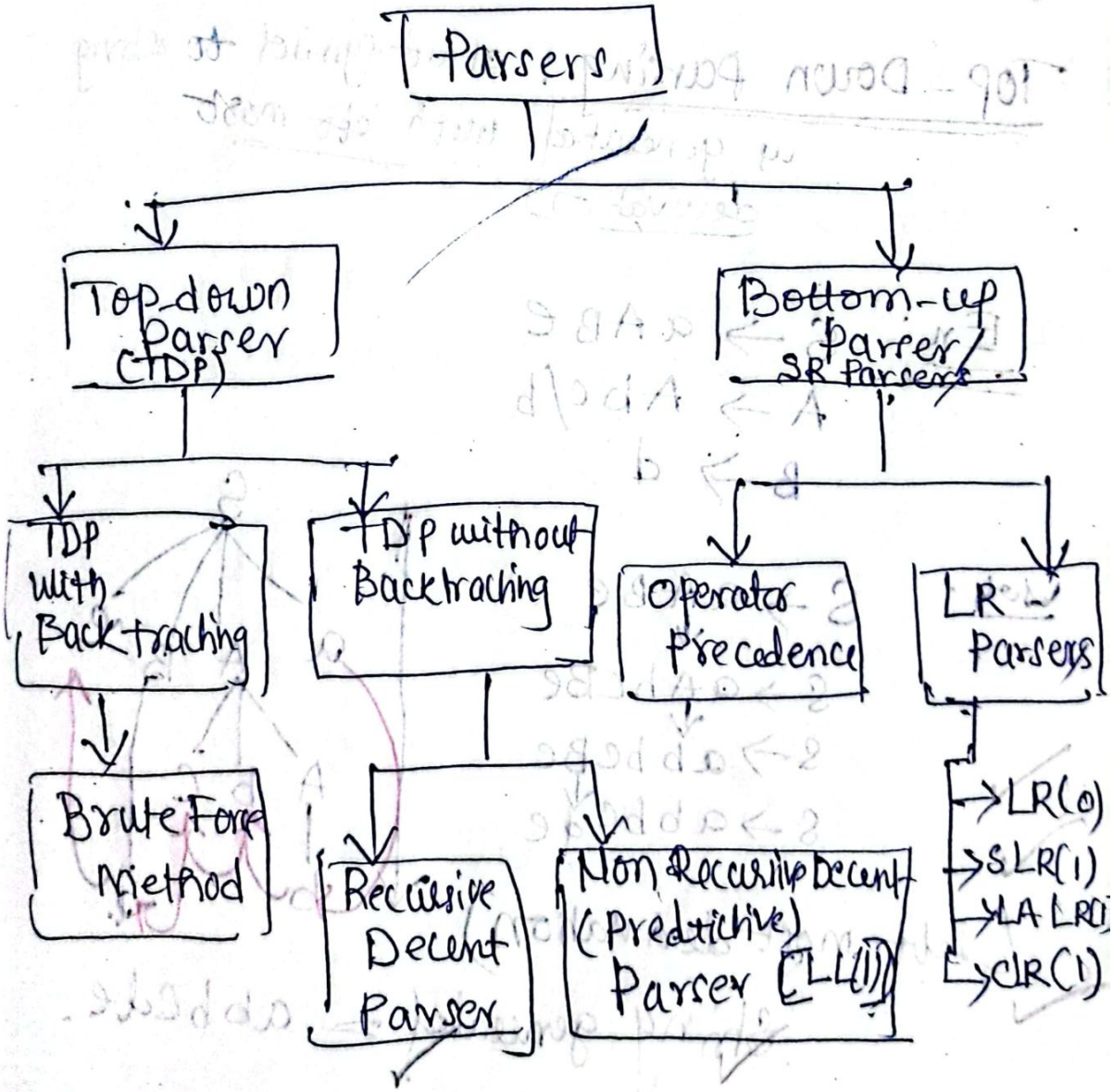
Dr. K. Bhargavi

Types of Parsing :

The task of Parser is to determine if and how the input can be derived from the start symbol within the rules of a Formal Grammar.

2 types of Parsers :

1. Top-down Parsing
2. Bottom-up Parsing



Grammar	Topdown Parser	Bottom up Parser
Ambiguous	✗	✗
UnAmbiguous	✓	✓
LR	✗	✓
RR	✓	✓
Non-Deterministic	✗	✓
Deterministic	✓	✓

Top-Down parsing: start symbol to string is generated with left most derivation

Ex: $S \rightarrow aABe$

$A \rightarrow Abc/b$

$B \rightarrow d$

Soln:

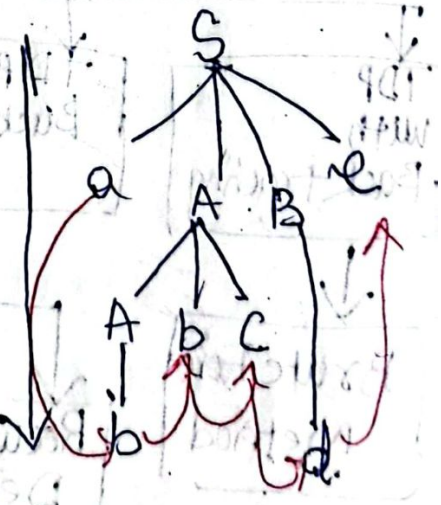
$S \rightarrow aABe$

$S \rightarrow aAbcBe$

$S \rightarrow abbcBe$

$S \rightarrow abbcde$

(left most derivation)



String generated = abbcde.

Bottom up parse tree : Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. String considered 'a' start symbol generated by using Right most derivation.

Ex.1: $S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

Right most derivation of given Grammar is

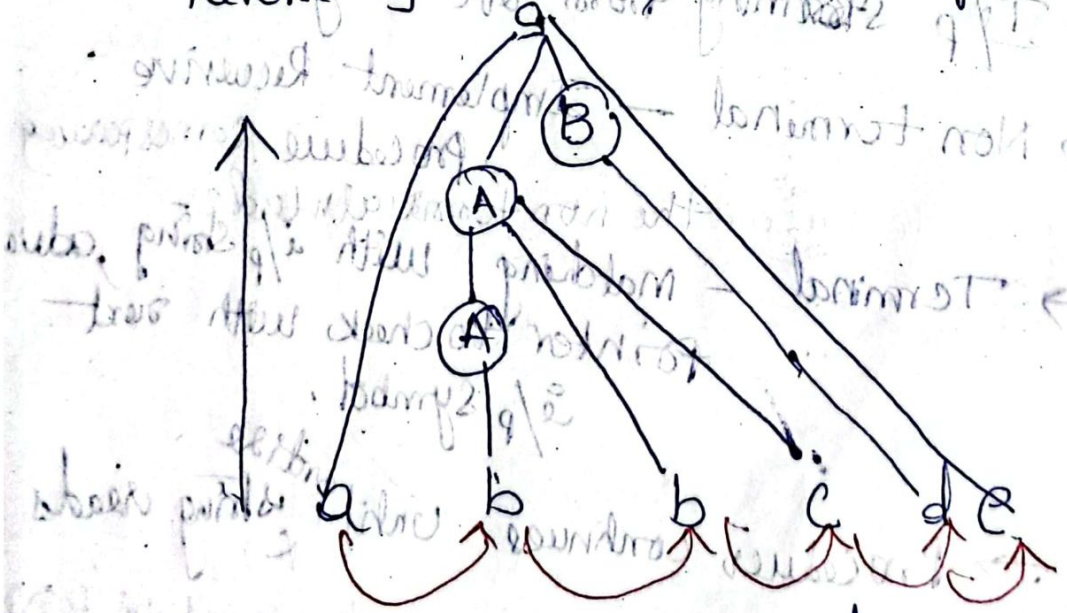
$S \rightarrow aABe$

$\rightarrow aAde$

$\rightarrow aAbcde$

$\rightarrow abbcde$

Reverse of RMD is Bottom up parser



String $\Rightarrow$ abbcde

Recursive Decent parser (RDP): (Topdown parser)

A Recursive decent parser is a topdown Parser.

This parser is simple to implement and is suitable for LL(1) grammar, where decision can be made based on a single lookhead token.

→ RDP uses recursion to analyse the syntax of a programing language based on a predefined grammar.

→ starts from the start symbol of the grammar & recursively calls function for each Non-terminal symbol, matching the against the production rule.

* RDP is used in lexical analysis (scanning) & syntax analysis (Parsing) Phase of compiler.

i.e 1 & 2 phase of compiler.

Recursive Decent Parser (RDP)

The parser uses collection of recursive Procedures for parsing the given input string is called RDP.

→ CFG is used to build recursive routine
→ RHS of Production rule is converted into Progsam.
→ Parsing table is not constructed
→ First & Follow is not used.

→ Procedure is associated with each non-terminal of the Grammar
→ I/p scanning from left to right
→ Non terminal — Implement Recursive Procedure corresponding the non-terminal made
→ Terminal — matching with i/p string, advance pointer to check with next i/p symbol.
→ Procedure continues until entire string reads
→ Left Recursion then not allowed for R.D.P.
→ NO back tracking

Recursive decent parser (RDP).

Q1). construct the Recursive decent Parser for the following grammar.

$E \rightarrow iE'$

$E' \rightarrow +iE' / \in$.

Solu:

Given grammar is

$E \rightarrow iE'$

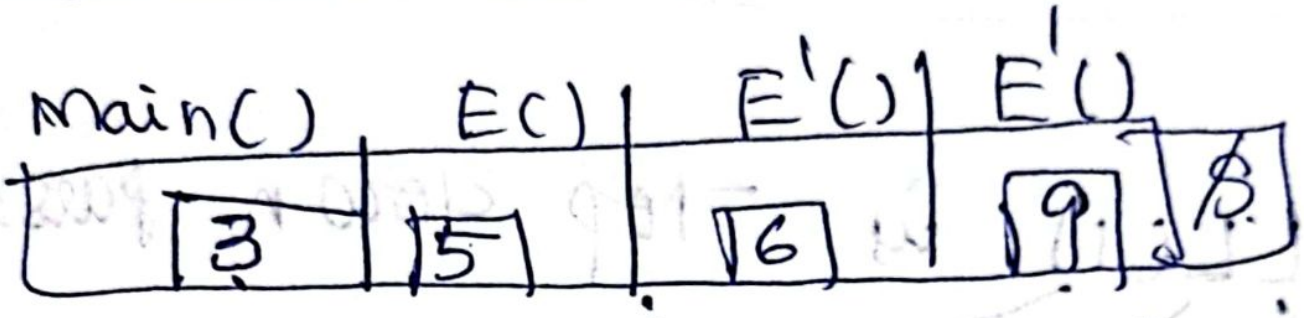
$E' \rightarrow +iE' / \in$

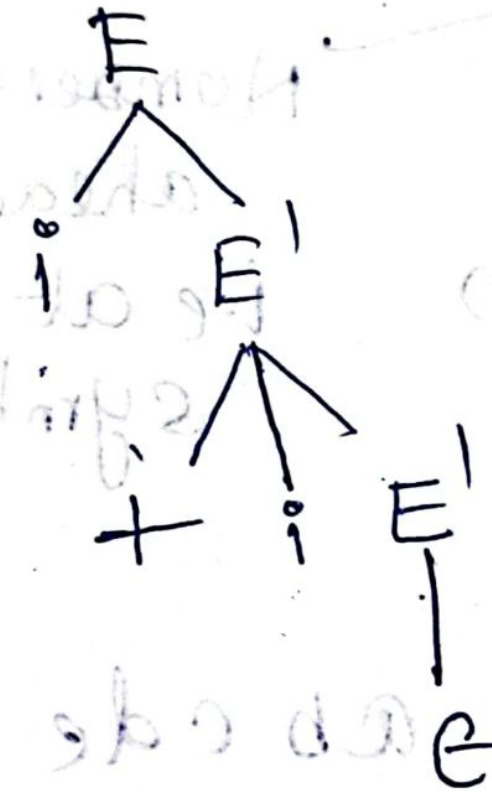
```
Main()
{
  E()
if input == '$'.
   Printf("Parsing successfully done");

}.
```

```
E()
{
  if (input == i)
     input++;
      E'();
   }.
E'()
  {
    if (input == '+').
     {
        input++;
      if (input == 't')
         input++;
           E'();
       }
    else
        return;
    }:
```

Consider the input string & construct RDP

$w = i + i \$$.





RDP (Recursive decent Parser)

(Q1) Construct the recursive decent parser for the grammar

$S \rightarrow 01S'$

$S' \rightarrow 0S1SS' / \epsilon$ & genrate the string 001101 or 01

or

$S \rightarrow S0S1S \mid 01$

Sol: Left recursion having

Removing it

$\therefore S \rightarrow 01S'$

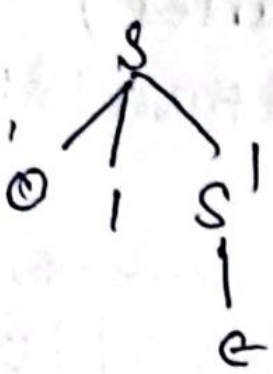
$S' \rightarrow 0S1SS' / \epsilon$

①
```
main()
{
  S();
if (input == '$')
{ Printf ("Parsing successful");
  }
}
```

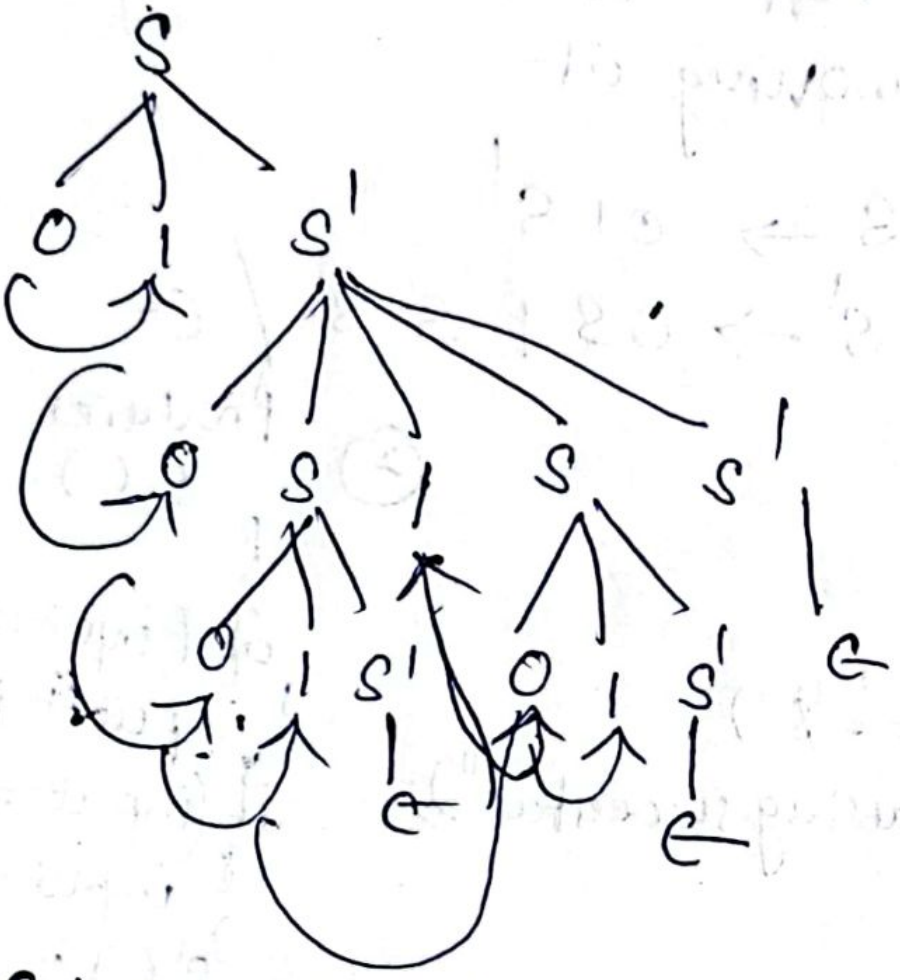
② Produres
```
S()
{
if (input == '0')
{input++
if (input == '1')
{input++
}S();
}
```

③
```
S'()
{
if (input == '0');
{ input++;
if input S();
if (input == '1');
  input++;
    S();
else
  return; // ε
}
```

(1)



01 generated.



01001101 ⇒ no. of 0's = No. of

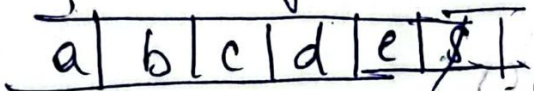
LL(1) Parser or Predictive Parser

LL(1) is Top down parser.

- Left-right Scan the input
- Left most derivation (LMD)
- Number of look ahead i.e at a time one symbol look / use

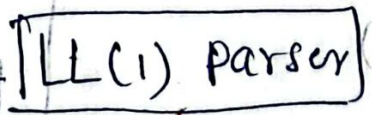
Ex: w = abcde

string scan from left to right



end of the string always represented with $

end of stack symbol



Stack DS used for Procedure of Parsing.

First() & Follow() :

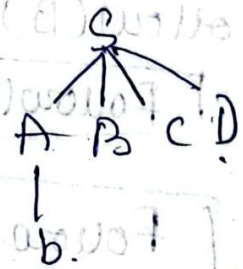
Ex:1

$S \rightarrow A\,B\,C\,D$
$A \rightarrow b$
$B \rightarrow c$
$C \rightarrow d$
$D \rightarrow e$

Find first & Follow functions for above production Rules.

Soln: Find out first & Follow fo LHS side

First(S) =



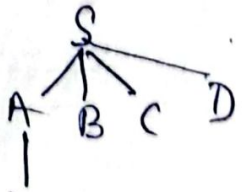
if it is terminal → consider it
if it is non-terminal → check for concern PR rodite the first of it.

∴ First (S) = b.

Ex2

$S \rightarrow A\,B\,C\,D$
$A \rightarrow b/\epsilon$
$B \rightarrow C$

First(S) =



if $b \Rightarrow$ First(S) = b
$\epsilon \rightarrow S \rightarrow \not\epsilon\, B\,C\,D$

In this case First(S) = C

First (A) = b

First (B) = c

First (C) = ϕ

First (D) = e

Follow() : S → ABCD A | B | C | D

Follow(S) = {$}

Follow(A) = First(BCD)

Follow(A) = {c}

S → ABC

First(BC)

First(B) =

Follow(B) = First(CD)

Follow(B) = {d}

Follow(c) = First(D) = {e}

Follow(D) = Follow(S) = $

Always i/p symbol followed by $.

Follow() never contain null or ϵ

* * *

Follow Rules :

Terminal — write as it is

Non-Terminal — write its First element.

Last element — write follow of LHS.

First()	Follow()
First(S) = b	Follow(S) = {$}
First(A) = b	Follow(A) = {c}
First(B) = c	Follow(B) = {d}
First(C) = d	Follow(C) = {e}
First(D) = e	Follow(D) = {$}

Construct the LL(1) table/predictive Parse table

	b	c	d	e	$
S	S→ABCD				
A	A→b				
B		B→c			
C			C→d		
D				D→e	

{ Columns — Terminal
  Rows — Non Terminal

Predictive Parser (LL(1))

- Top down parser
- No Backtracking
- LL(1) Parser

I/P → CFG → O/P → Parse Tree

* Steps for LL(1) parser

1. Elimination of Left Recursion
2. Elimination of left Factoring
3. Find First() and Follow()
4. Construct parse Table
5. Parsing input string
6. construct parse tree

Q 4. [illegible]

(iii)

$S \to iEtSS' / a$

$S' \to eS / \epsilon$

$E \to b.$

(ii)

$S \to iEtS / iEtSeS / a$

$E \to b$

Remove left Factoring

Construct first(), follow() & Construct Pause table of LL(1).

Soln : (i) No left recursion & left factoring

(ii) First(S) = {i, a}

First(S') = {e, ∈}

First(E) = {b}.

Follow(S) = Follow of starting symbol always $ & Check on RHS of S, followed by First of S,

First(S') = {e, ∈}.

Null of ∈ not allowed in Follow() Substitute ∈ in PR.

$S \to iEtS$ ↓ last element

⇒ Follow(S)

Follow(S) = {$, e}.

Follow(S) = follow(S') = {e, $}

Follow(E) = {t}

	First	Follow
S	{i, a}	{e, $}
S'	{e, ∈}	{$, e}
E	{b}	{t}

(b) LL(1) Parse table :

	i	a	e	b	t	$
S	S→iEtSS'	S→a				
S'			S'→eS S→∈			S→∈
E				E→b		

Column → Terminals

Rows → Non Terminals

** If any entry on the consisting more than one entry, then we can say that, it is not in LL(1) Parser Grammar

Note: The above one having multiple entries.

Q5: (3M) Check LL(1) parser for the following Grammar.

$S \rightarrow aABb$

$A \rightarrow c/\epsilon$

$B \rightarrow d/\epsilon$.

and also check validity of string "acd"

Soln:

(1) No. Left Recursion having in the given grammar.

(2) Calculation of First():

First(S) = {a}

First(A) = {c, ∈}

First(B) = {d, ∈}

Follow():

Follow(S) = {$}

Follow(A) = First(B)

= {d, ∈} but ∈ not allow substute ∈ in

$S \rightarrow aA(B)b \Rightarrow (b)$

Follow(A) = {d, b} * *

Follow(B) = {b}

(3) Parse Table :

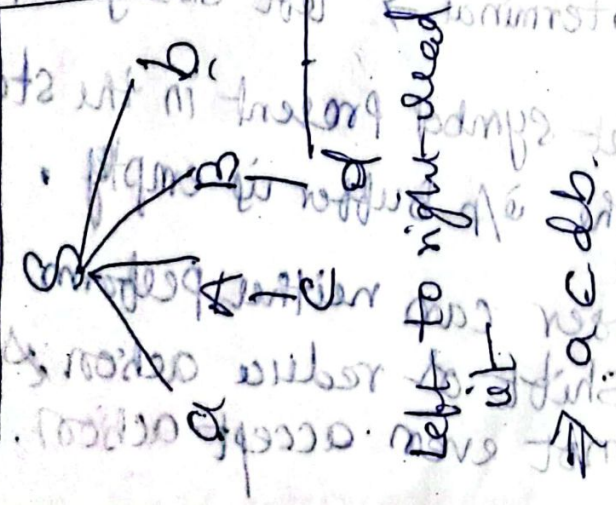
	a	b	c	d	$
S	S→aABb				
A		A→ϵ	A→c.	A→ϵ	
B		B→ϵ		B→d.	

Result table does not contain multiple entries thus given grammar LL(1).

(4) String validation

Stack	Input	Action
$S	acdb$	S→aABb.
$bBAa.	acdb$	pop('a')
$bBA	cdb$	A→c.
$bBc	cdb$	pop(c)
$bB	db$	B→d.
$bd	db$	pop(d)
$b	b$	pop(b)
$	$	Accepted.

LL(1) Parse Tree



LR Parsers (Bottom-up Pa

Bottom up parsing starts from left nodes of tree & works in upward direction till it reaches root node,

* LR is one type of Bottom up Parser.

LR Parsers are non-Recursive, Shift redu bottom-up parser.

LR Parsers are also known as LR(K) Parse

where

L → left to right scanning of i/p Stri

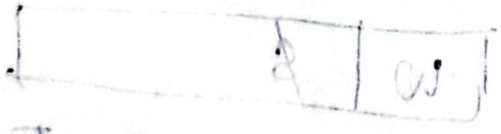
R → construction of right most derivation in reverse

k → No. of look heads needed for decision.

LR(0) < SLR(1) < LALR(1) < CLR(1
Simple LR look ahead LR canoni LR

CLR(1) is most LR powerful parser.

→ LR parsing is Non-Backtracking shift-reduce parser & make use of DPA



Steps for constructing the LR Passing table

**Check the ambiguity of the Grammar

1. write the agumented grammar
2. LR(0) collection of items to be found.
3. Defining two (2) functions:
   a) Goto (List of non-terminals)
   b) Action (List of terminals)
4. construct parse table with Step-3 (goto and action)

Augment Grammar helps, when to stop the parser and acceptance of grammar.
We use dot (.) to scan & process symbol after dot.

Terminal (T)
NonTerminal (V)

$F \rightarrow .xyz$ → Ready to scan to x
$F \rightarrow x.yz$ → x is scanned & Ready to scan y.
$F \rightarrow xyz.$ → All symbols scanned.

SLR(1) Parser :

look ahead.

Simple LR(1) Parser.

Scan from left to Right — Right most derivation

Prerequisite : LR Parser.

SLR is Simple LR Parser. It is the smallest class of grammar having few no. of states.

SLR(1) is similar to LR(0) Parsing.

In LR(0)	SLR(1)
1. There is a chance of conflicts shift-reduce	1. conflict are Sloved here.
2. Entering "reduce" corresponding to all the terminals (i.e Action part) The conflicts in LR(0) are Solved with SLR(1).	2. In SLR(1) "reduce" corresponding to FOLLOW of LHS of the reduced non-terminal.
3. Zero lookahead	3. one lookahead.

Steps for SLR(1) Parser :

1. Writing Augmented grammar
2. LR(0) collection of items to be found
3. Find FOLLOW of LHS of production
4. Define 2 functions
   ① Goto — list of non terminals
   ② Action — list of terminals

In compiler design SLR(1) stands for Simple left to right rightmost derivation Parser with 1 symbol look ahead, a type of bottom-up parsing method. It is simple and efficient parser

SLR(1) — consider the next i/p symbol (look ahead) to determine the parsing action

- Simple
- Scans i/p Left to Right
- Rightmost derivation in reverse

[Shift
[Reduce

Shift — Push the current i/p symbol to stack & move to the next state

Reduce — If current stack symbol match in production replace with non terminal symbol of production & push into stack, then move into corresponding state

Eg 2: Construct the SLR(1) parsing table for the given grammar

$S \to AA$

$A \to aA / b$

Soln:

1. Constructing agumented grammar.

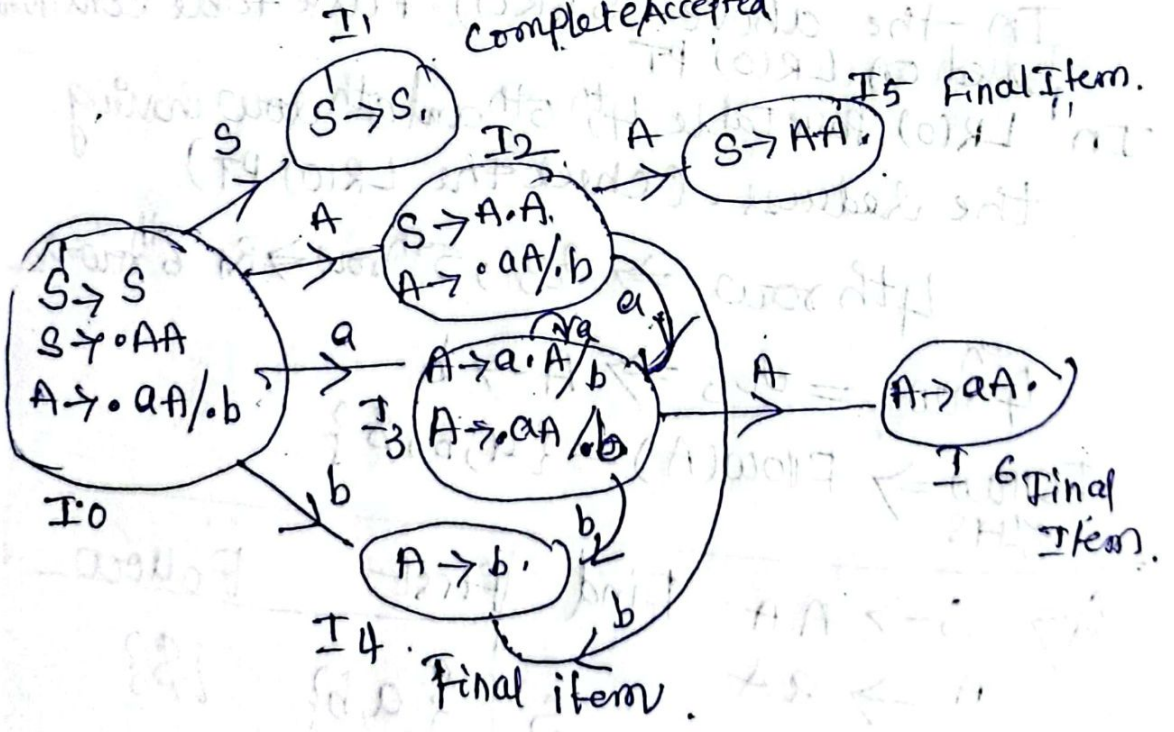
$S' \to .S$. — 0

$S \to .AA$ — 1

$A \to .aA$ — 2

$A \to .b$ — 3

2. Computing LR(0) items / conoical collection of LR(0) items



Steps : Construct SLR(1) Parse Table

State	Action Part			Goto Part	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S3	S4			5
3	S3	S4			6
4	R3	R3	R3		
5			R1		
6	R2	R2	R2		

In the above SLR(1) Parse table constructed on LR(0) PT.

In LR(0) Parse Table 4th, 5th and 6th row having the reduces. (Check the LR(0) PT)

4th row ⇒ R3, 5th row ⇒ R1, 6th row

4th row = R3 ⇒ A → b

Follow ⇒ Follow(A) = {a, b, $}
of LHS

G ⇒ S → AA
A → aA
A → b

Find	First	Follow
S	{a, b}	{$}
A	{a, b}	{a, b, $}

∴ place r_3 in a, b, $ columns.

5th row = r_1

$S \rightarrow AA$

Follow LHS = Follow(S) = $ ⇒ place r_1 in only in $ columns.

6th row = r_2 ⇒ $A \rightarrow aA$

Follow LHS = Follow(A) = {a, b, $}

∴ place R_2 in a, b & $ columns.

2Q: Create or construct the SLR(1) Parsing table for the following grammar.

$E \rightarrow T+E/T$

$T \rightarrow id$

Or Check the Given grammar is in LR(0) or not. (Answer till Step 2)

Solu:

$E \rightarrow T+E$

$E \rightarrow T$

$T \rightarrow id.$

Identify the Non-terminal of LHS from Production rule i.e E. & write the

1) ~~Agumented Grammar~~

$E \rightarrow .E$ — (0)

$E \rightarrow .T+E$ — (1)

$E \rightarrow .T$ — (2)

$T \rightarrow .id$ — (3)

$E' \rightarrow .E$

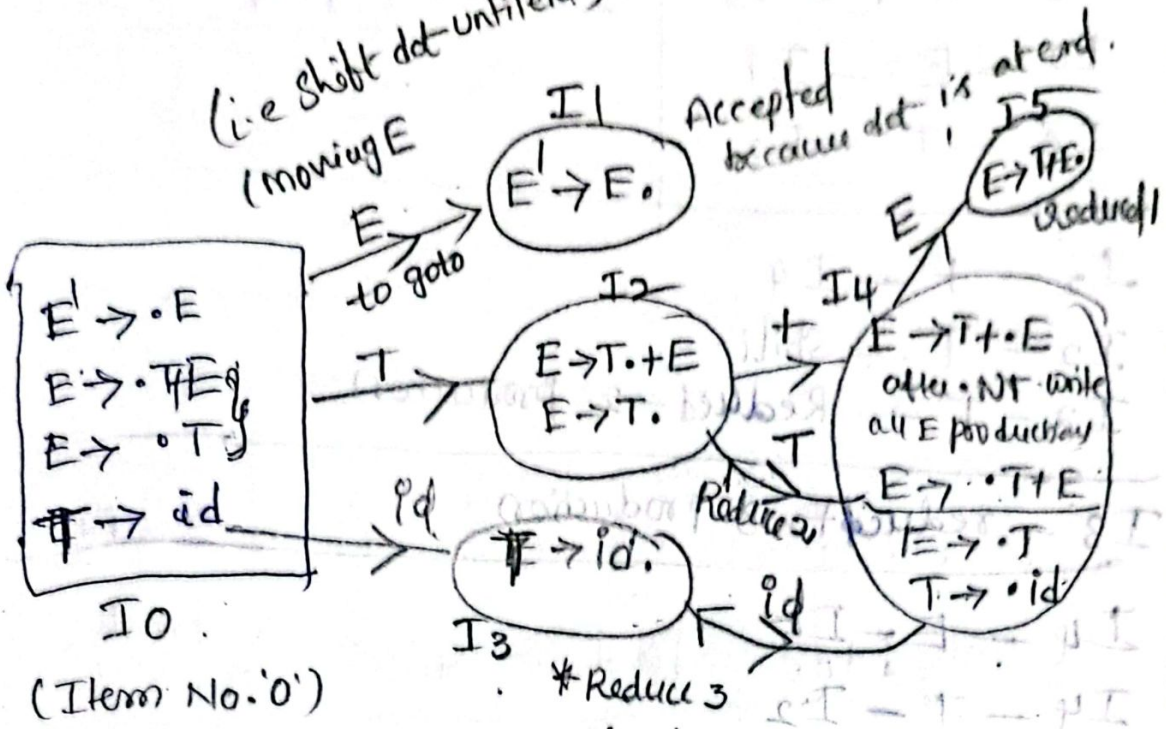
$E \rightarrow .T+E$

$E \rightarrow .T$

$T \rightarrow .id.$

After dot (.)
if
Terminal → Write as
Non-Terminal → open the concern production Rule.

(2) Canonical items of LR(0) :



(3).

LR(0) Parse Table. :.

State	Action Part			Goto Part	
	id	+	$	E	T
0 (I0)	S3	—	—	1	2
1 (I1)	—	—	Accept	—	—
2 (I2)	r2	S4/r2	r2	—	—
3 (I3)	r3	r3	r3	—	—
4 (I4)	S3	—	—	5	2
5 (I5)	r1	r1	r1	—	—

I_0 — id — S_3 I_0 — E — I_1 I_0 — T — I_2	I_1 — Accepted.

I_2 — + — I_4
I_2 — T — Shift.
I_2 — T — Reduced — 2 production.

I_3 — reduced to 3 production.

I_4 — E — I_5
I_4 — T — I_2
I_4 — Id — Shift I_3.

I_5 - Reduced to 1 Production.

* Note

If LR(0) Parse Table having the following then it is not in LR(0) Parser

1) Shift — Reduce conflict
2) Reduce — Reduce conflict.

In this example LR(0) Parse table consisting Shift - Reduce conflict

i.e S_4/R_2 ∴ Not in LR(0) Parser.

Soln:

Canonical Items of SLR(1) = LR(0)

Only the difference is Parse Table.

States	Action					GOTO	
	id	+	$		E	T
0 (I_0)	S3	—	—		1	2
1 (I_1)	—	—	Accept			
2 (I_2)	—	S4	R_2		—	—
3 (I_3)	—	R_3	R_3		—	—
4 (I_4)	S3	—	—		5	2
5 (I_5)	—	—	R_1		—	—

Follow(E) = $

Follow(T) = Follow(E) + +

= {+, $}

Consider "Reduce" from LR(0).

2nd, 3rd & 5th Rows containing Reduce.

2nd Row = r_2

i.e $E \to T$

LHS is E, ∴ Follow(E) = {+, $}

∴ Write r_2 only under $ Column

3rd row = r_3.

$T \Rightarrow id$

LHS $\Rightarrow T \Rightarrow$ Follow(T) = {+, $}

Write r_3 under +, $.

5th row = r_1.

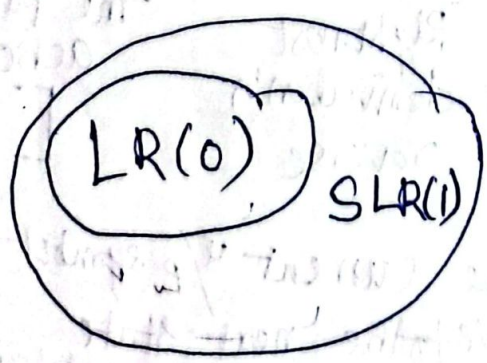
$E \to T+E \Rightarrow$ LHS E $\Rightarrow$ Follow(E) = $

∴ Write r_1 under $

Remaining Goto part is same as LR(0).

SLR(1) Not going to have conflicts

i.e



CLR(1) :- Canonical left Right (1) Parser.

CLR refers to canonical lookahead.

→ Type of Bottom up Parser.

→ CLR Parsing uses canonical collection of LR(1) items to build the CLR(1) parsing table.

→ CLR(1) parsing table produces the more number of states compare to SLR(1).

→ In CLR(1) we place the reduce node only in the lookahead symbols.

CLR(1) ⇒ Using canonical Items of LR(1).

Canonical — scans from left to Right — RMD — lookahead.

LR(1) Items = LR(0) items + a look head.

Note: In CLR(1) & LALR(1) using Single look a head..

Steps for CLR(1) :

1) write the agument grammar & add the look a head.
2) Find canonical collection of LR(1) items.
3) construct the CLR(1) Parse table. by considering look a head symbols. (Place Shift/Reduce in lookahead symbol)

Steps

1. For given input string write a context free Grammar.
2. Check the ambiguity of Grammar
3. Add augment: production in the given Grammar & add lookahead.
4. Create canonical collection of LR(1) items
5. Draw a data flow diagram.
6. Construct CLR(1) parsing table.

(Q) **Example:** Construct the CLR(1) Parse table for the following Gramm

$S \rightarrow AA$

$A \rightarrow aA / b$

Soln: $S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b.$

① Add Augment production.
Insert dot (·) symbol at the 1st pos
for every production in G.
And also add the lookahead i.e ($)
starting symbol of G

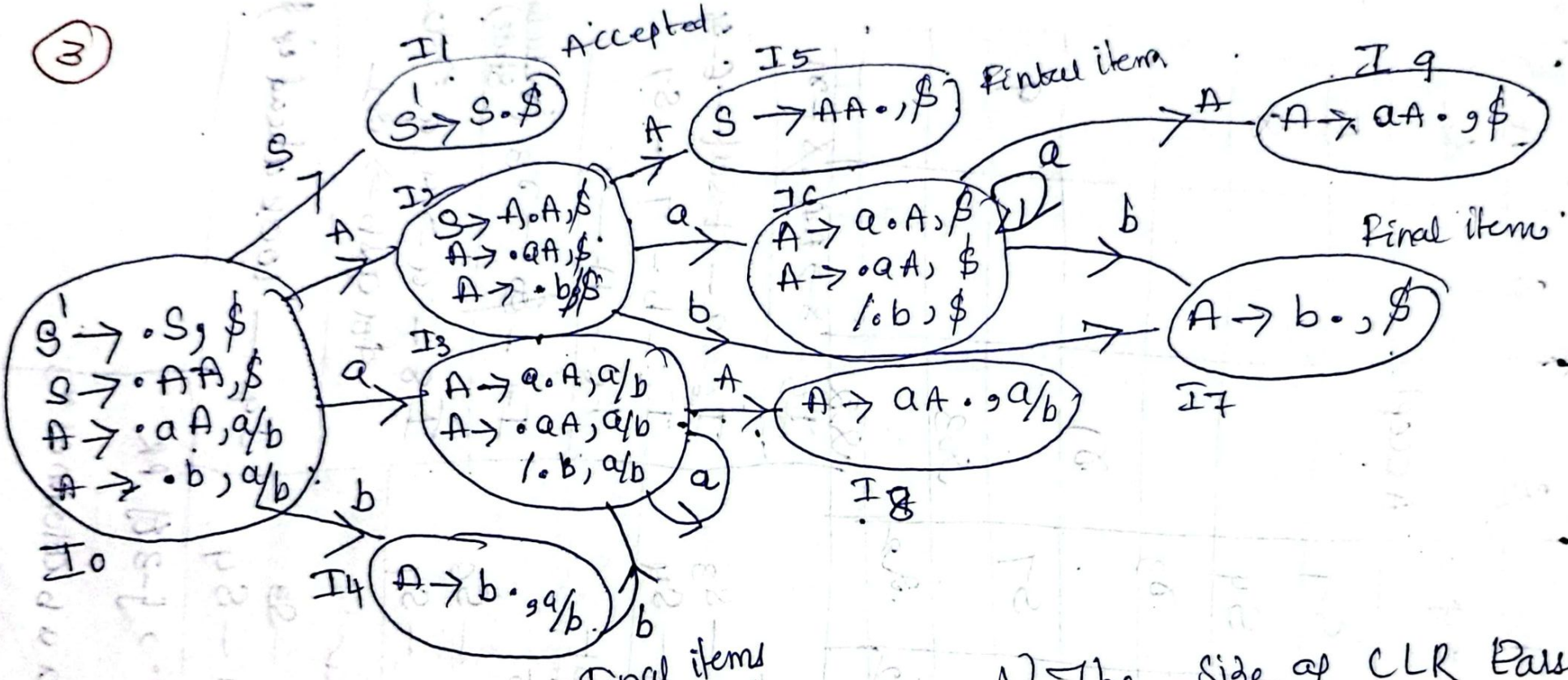
$S' \to \cdot S, \$$ — ⓪ (· $ is a lookahead)
$S \to \cdot AA, \$$ — ① (production 0)
$A \to \cdot aA$; a/b — ② [Note : $S \to \cdot A(A, \$)$
$A \to \cdot b$ / a/b — ③ After A, it is A, $.
First(A) = {a, b}.
∴ for A lookahead is {a, b}
Add in A' products

$S \to \cdot AA, \$$
After dot consider A. Find out.

② Note : LR(1) = LR(0) + alook ahead.
Initial symbol, a look head is $.
1. $S' \to .S, \$$ (Initial state lookahead is $)
2. $S \to .AA, \$ \to$ (look head for previous one).
3. $A \to .aA, a/b$ → ∴ $
4. $A \to .b, a/b \to$ (First(A, $) ⇒ {a, b}

here same symbol of LHS side so same look a head for 4 production

(3)



NOTE: 1) The size of CLR Parse table is quite large compared to other. So, it occupies more space in memory.

2) CLR avoids conflicts in Parsing table.

④ CLR parse table

States	Action – Terminals			Goto – N	
	a	b	$	S	A
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

$I_0 - S - 1$
$I_0 - A - 2$
$I_0 - a - 3 - S_3$
$I_0 - b - 4 - S_4$

I_1 – Accept

$I_2 - A - 5$
$I_2 - a - I_6 - S_6$
$I_2 - b - I_7 - S_7$

$I_3 - A - 8$
$I_3 - a - I_3 - S_3$
$I_3 - b - I_4 - S_4$

$I_4 \rightarrow A \rightarrow b\cdot$, $f - 3$ times, PR
lookhead ⇒ a/b
are place a & b columns as r_3

$I_5 - S \rightarrow AA\cdot, \$, r_1$

$I_6 - a - $ It self $- 6 - S_6$
$I_6 - b - 7 - S_7$
$I_6 - A - 9$

$I_7 - \$ - r_3$ (reduction state)

$I_8 - a \rightarrow a, a \Rightarrow r_2$
for a & b ⇒ r_2

I_9 – look ahead is $

LALR(1)

⇒ LALR(1) parser is lookahead LR Parser.
It is the most powerful parser which can handle large classes of Grammar.

⇒ LALR(1) uses canonical items of LR(1)

LALR(1) parsing is same as the CLR(1) Parsing, only difference is the parsing table.

In LALR(1), combines the similar states of CLR parsing table into one single state.

Construction of LALR(1) Parser

1. Construct all cannonical LR(1) states.
2. Merge those states that are identical if the lookaheads are ignored.
   i.e 2 states beging merged must have the same number of items & items have the same core.
3. The goto function for the new LALR(1) state is the union of merged states.
4. Action & goto entries are constructed from the LALR(1) states as for the Canonical LR(1) parser.

Q1) Construct the LALR(1) Parse table for the given grammar.

$S \rightarrow AA$

$A \rightarrow aA/b$

Solution: $S \rightarrow AA$

$A \rightarrow aA$

$A \rightarrow b$.

⇒ Follow the same answer given for CLR(1).

Write the CLR(1) Parse Table.

⇒ Check CLR(1) Parse table, having same production & different lookahead.

⇒ Then write the LALR(1) Parse table.

LALR(1)

Check is there any two(2) states which have the same production but different lookahead, from CLR(1) Parse Table.

Identify that

⇒ $I_3, I_6 \Rightarrow I_{36}$

⇒ $I_4, I_7 \Rightarrow I_{47}$

⇒ $I_8, I_9 \Rightarrow I_{89}$.

Consider I_3 & I_6.

Then column 3 becomes 36 & remove 6th column.

Check shifts related to 3 & 6 add there also.

Ex: S_3 & S_6

Then becomes S_{36}

I_4 & I_7

⇒ 4th column became 47 & do similarly for shifts

I_8 & I_9 ⇒ 8th column became 89. & do similarly for shifts if exists.

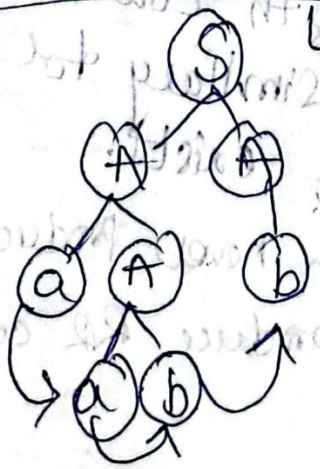
Merger states can never produce shift-reduce conflicts. It can produce RR conflict.

∴ LALR(1) Parsing table

States	Action Part			Goto Part	
	a	b	$	S	A
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		

* The no. of rows has been reduced from 10 to 7 rows
* 36 row and the same data twice. So delete 1 row.
* We combine two 47 rows in to one by combing each value in the singl 47 row.

Generate aabb & Draw the LALR Parse tree for a abb



aabb

UNIT-4(Chapter 2 & 3)

Semantic Analysis: Introduction to Syntax Directed Definition, Syntax Directed Translation, Attributes, Types of Attributes, Bottom-up evaluation of attributes.

Intermediate code generation: Types of Intermediate codes, Types of Three address codes

Introduction to Syntax-Directed Definitions and Translation Schemes

1. When we associate semantic rules with productions, we use two notations:
   1. Syntax directed definition (SDD)
   2. Translation Scheme (SDT)

A. Syntax-Directed Definitions:

- give high-level specifications for translations
- hide many implementation details such as order of evaluation of semantic actions.
- We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.

B. Translation Schemes:

- indicate the order of evaluation of semantic actions associated with a production rule.
- In other words, translation schemes give a little bit information about implementation details.

Syntax-Directed Definition

A syntax-directed definition (SDD) is a context-free grammar with attributes and rules.

Syntax Directed Definition = CFG + Semantic rules

- Attributes are associated with grammar symbols and rules are associated with productions.
- These rules are set by the grammar of the language and evaluated in semantic analysis.

Example:

- If X is a symbol and α is one of its attributes, we write $X.\ \alpha$ to denote the value of a at a particular parse-tree node labeled X. Attributes may be numbers, types, table references or strings

E → E + T ***E. value =E. value + T. value***

- Example: E → E +T
- The above CFG production has no semantic rule associated with it and it cannot help in making any sense of the production
- A parser builds parse trees in the syntax analysis phase.
- The plain parse-tree is of no use for a compiler, as it does not carry any information of how to evaluate the tree.
- The productions of context-free grammar that makes the rule so the languages do not accommodate how to interpret them

Syntax-Directed Definition -- Example

Production	**Semantic Rules**
$L \rightarrow E$ **n**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ **digit**	F.val = **digit**.lexval

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyser is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Attributes

- Each Grammar symbol is associated with set of attributes
- Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions
- Attributes can be
  - String
  - Number
  - Type
  - Memory Location

•The value of the attribute at a parse tree node is defined by a semantic rule associated with the production used at the node

Types of Attributes

- Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions.
- Based on the way the attributes get their values.
- They divided into two(2) categories:
    1. Synthesized attributes
    2. Inherited attributes

Synthesized attributes

These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:

$S \rightarrow ABC$

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

1. **It can be evaluated during a single bottom-up traversal of parse tree.**
2. **Synthesized attributes can be contained by both the terminals or non-terminals.**

Synthesized attributes

CFG	Semantic Actions
$E \rightarrow E + T$	{ E. value = E. value + T. value}
$E \rightarrow T$	{E. value = T. value}
$T \rightarrow T * F$	{T. value = T. value * F.value}
$T \rightarrow F$	{T.value = F.value}
$F \rightarrow num$	{F. value = num.lexicalvalue}

Synthesized attribute is used by both S-attributed SDT and L-attributed SDT.

Inherited attributes

In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

$S \rightarrow ABC$

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

1. **It can be evaluated during a single top-down and sideways traversal of parse tree.**
2. **Inherited attributes can't be contained by both, It is only contained by non-terminals.**
3. **Inherited attribute is used by only L-attributed SDT.**

Inherited attributes

Unlike synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,

Annotate the parse tree for the computation of inherited attributes for the given string int a, b, c.

Solution

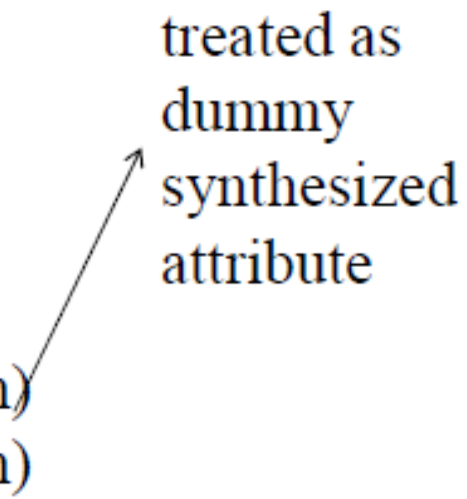
Step1. The Syntax Directed Definition for the given grammar.

Productions	Semantic Rules
S ⟶ T L	L.in .T.type
T ⟶ int	T.type = int
T ⟶ float	T.type = float
T ⟶ char	T.type = char
T ⟶ double	T.type = double
L ⟶ L1, id	L1.in = L.in
L ⟶ id	Id.entry = L.inh

Example Attribute Grammar with Synthesized & Inherited Attributes

Simple Type Declaration

Production	Semantic Rule
$D \rightarrow T\ L$	L.inh = T.type
$T \rightarrow$ **int**	T.type = '*integer*'
$T \rightarrow$ **float**	T.type = '*float*'
$L \rightarrow L_1$ **, id**	L_1.inh = L.inh;
	addtype(**id**.entry, L.inh)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.inh)



Synthesized: T.type, **id**.entry
Inherited: L.inh

Parse Tree

Production Rules

$D \rightarrow T\ L$

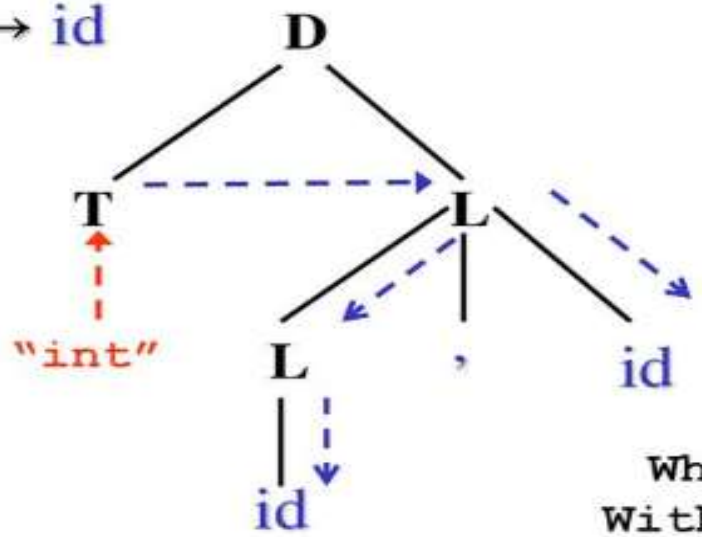
$T \rightarrow int$

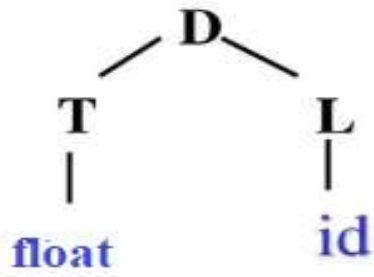
$T \rightarrow float$

$L \rightarrow L\ ,\ id$

$L \rightarrow id$

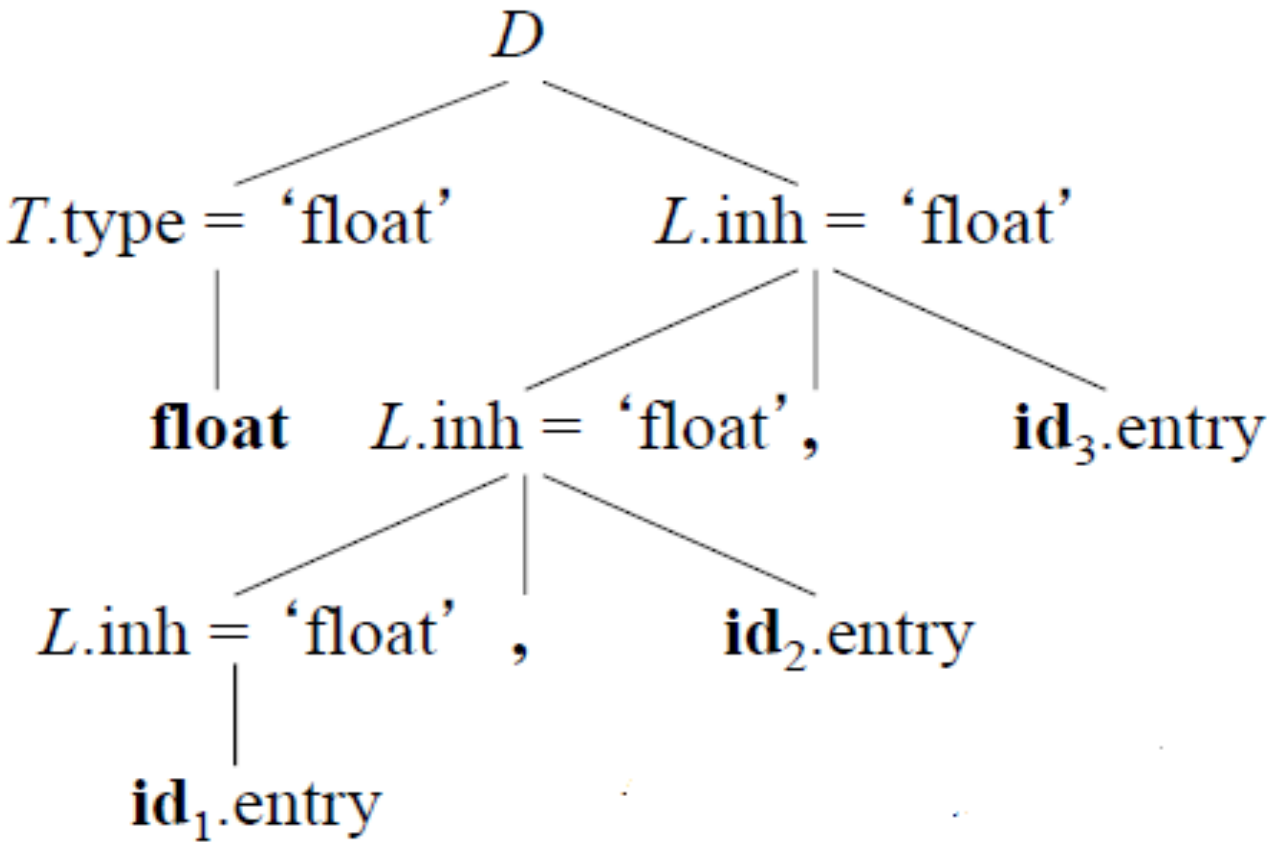
$D \Rightarrow TL \Rightarrow int\ L$
$\Rightarrow int\ L\ ,\ id$
$\Rightarrow int\ L\ ,\ id\ ,\ id$
$\Rightarrow int\ id\ ,\ id\ ,\ id$





Where is Type Information
With respect to Identifiers?

Construct the annotated parse tree for
float id1,id2,id3 using previous production



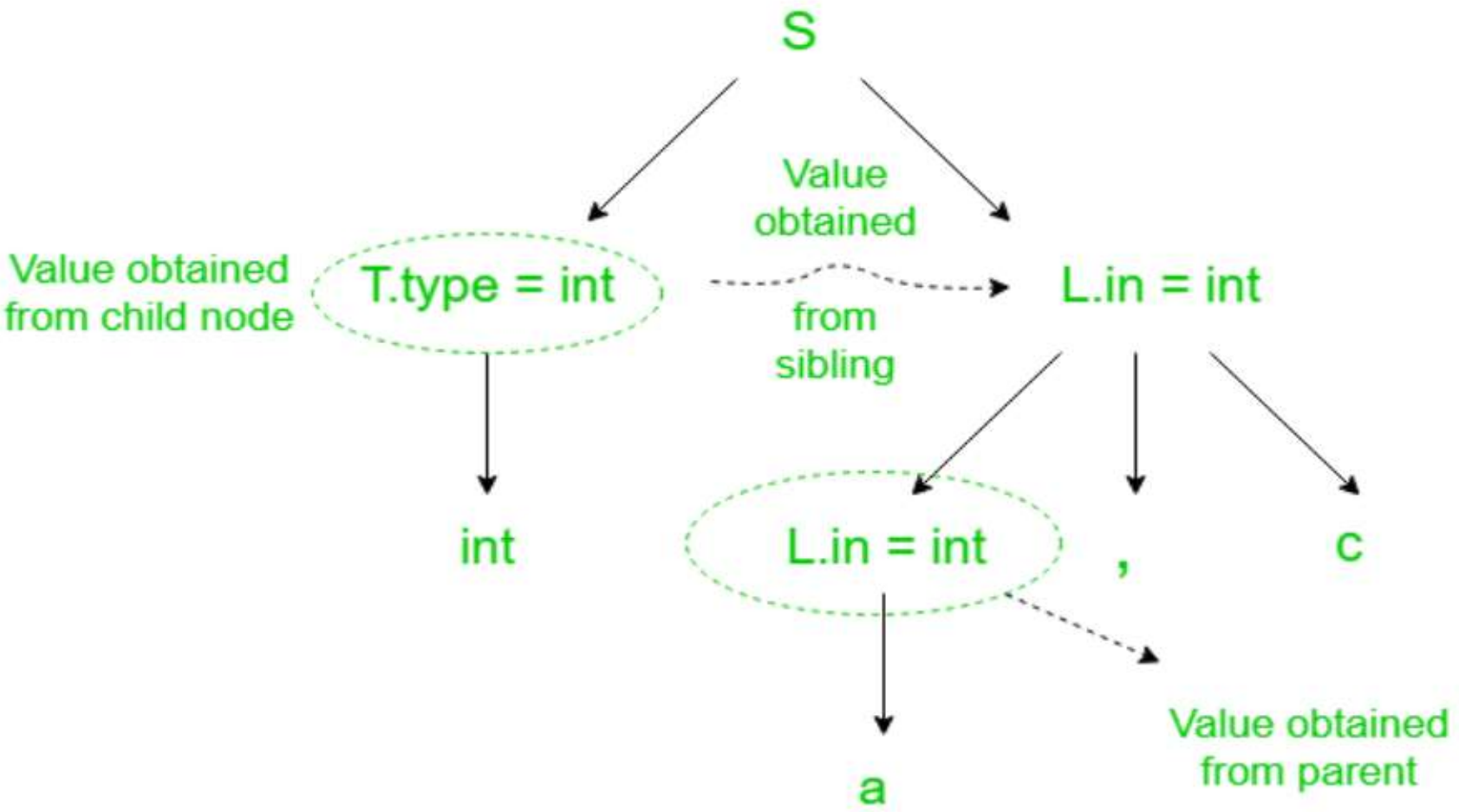
Annotated Parse Tree

1. A parse tree showing the values of attributes at each node is called an **annotated parse tree**.

2. Values of Attributes in nodes of annotated parse-tree are either,
   - initialized to constant values or by the lexical analyzer.
   - determined by the semantic-rules.

3. The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.

4. Of course, the order of these computations depends on the dependency graph induced by the semantic rules.



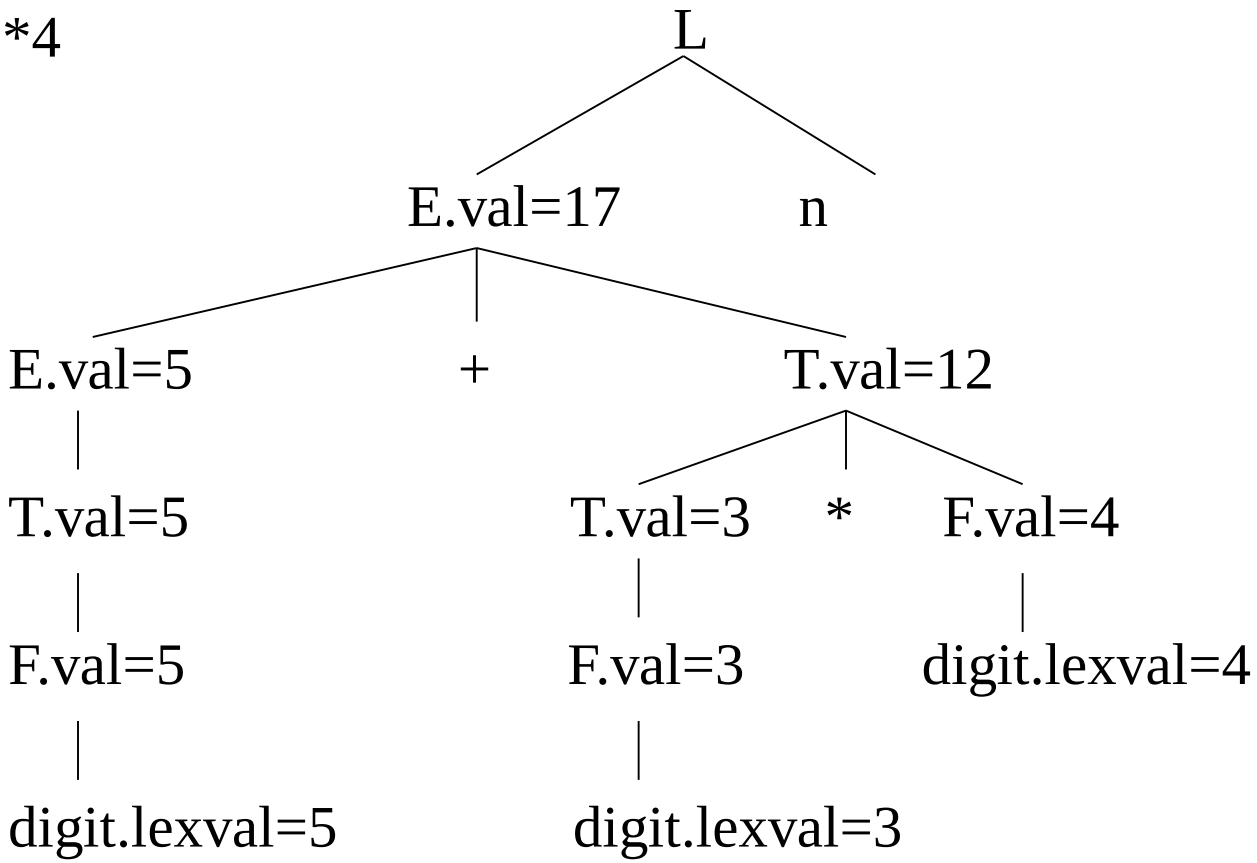
Annotated Parse Tree

Annotated Parse Tree -- Example

Production	**Semantic Rules**
$L \rightarrow E$ **n**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ **digit**	F.val = **digit**.lexval

Annotated Parse Tree -- Example

Input: 5+3*4



Evaluation order/ Implementation of SDD's

- Evaluation order for SDD includes how the SDD(Syntax Directed Definition) is evaluated with the help of attributes, dependency graphs, semantic rules, and S and L attributed definitions

Evaluation order/ Implementation of SDD's :

- Dependency graphs are a useful tool for determining an evaluation order for the attribute instances in a given parse tree.
- While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.
- Dependency graphs define two important classes of SDD's:
  - » S-attributed SDD's
  - » L-attributed SDD's

Dependency Graph

- The basic idea behind dependency graphs is for compiler to look for various kinds of dependence among statements to prevent their execution in wrong order i.e. the order that changes the meaning of the program.
- This helps it to identify various parallel executions of the components in the program.
- For rule $X \rightarrow YZ$
- the semantic action is given by $X.x = f(Y.y, Z.z)$
- then synthesized attribute is X.x and X.x depends upon attributes Y.y and Z.z

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

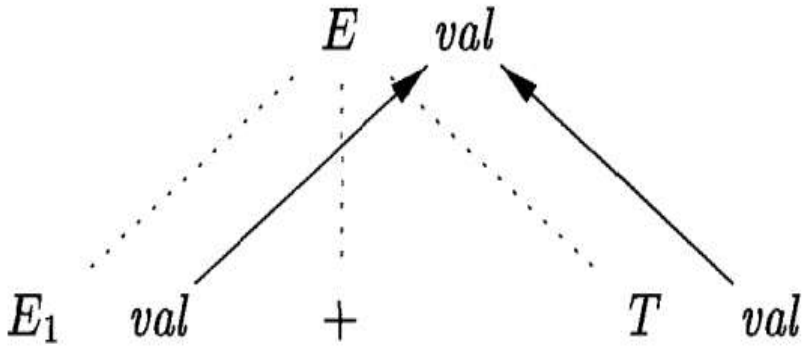


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

Syntax Directed Translations

- Syntax Directed Translation has augmented rules to the grammar that facilitate semantic analysis.
- In stream is the result obtained by evaluating the semantic rules.
- Semantic analyzer receives AST (Abstract Syntax Tree) from its previous stage (syntax analysis).
- Semantic analyzer attaches attribute information with AST, which are called Attributed AST.

Attributes are two tuple value, <attribute name, attribute value>

For example:

int value = 5;

<type, “integer”>

<present value, “5”>

For every production, we attach a semantic rule.

Syntax Directed Translation

- syntax-directed definition and translation schemes, we parse the input token stream, build the parse tree, and then traverse the tree as needed to evaluate the semantic rules at the parse tree nodes.

The conceptual view of syntax directed translation

In Syntax Directed Translations the Productions with semantic actions embedded with production bodies.(Actions are specified)

Eg: E→E1+T { print‘+’ }

- The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order.

- SDT involves passing information bottom-up and/or top-down to the parse tree in form of attributes attached to the nodes.

- Syntax-directed translation rules use
  1. lexical values of nodes,
  2. constants &
  3. attributes associated with the non-terminals in their definitions.

Construct Syntax Directed Translation for the given grammar (for 2*3+ 4)

E → E + T

E → T

T → T * F

T → F

F → num

Solution:

1. Context free grammar + semantic actions are said to be syntax Definition Translation

2. Abstract Syntax Tree for 2 * 3 +4

3.Constructing Parse Tree for the above grammar

4. Annotated Parse Tree

Step 1: 1. Syntax-Directed Translation (SDT)
This is a context-free grammar + semantic actions. SDT is a way to attach meaning (semantics) to the syntactic structure of a program.

Example with semantic actions:

Semantic actions are to be written in between curly braces.

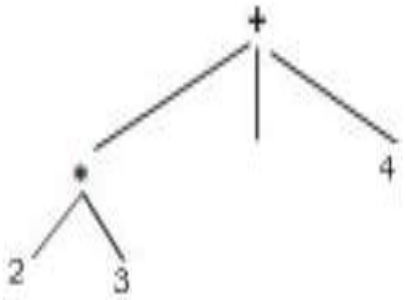
Contest Free Grammar	Semantic Actions
$E \rightarrow E + T$	{ E. value = E. value + T. value}
$E \rightarrow T$	{E. value = T. value}
$T \rightarrow T * F$	{T. value = T. value * T. value}
$T \rightarrow F$	{T. value = F. value}
$F \rightarrow num$	{F. value = num. lexical value}

Step 2 -Abstract Syntax Tree (AST) for 2 * 3 + 4

An AST omits the intermediate grammar rules and focuses only on operations and operands. It shows the essential computation.

Expression: 2 * 3 + 4

Operator Precedence: * has higher precedence than +

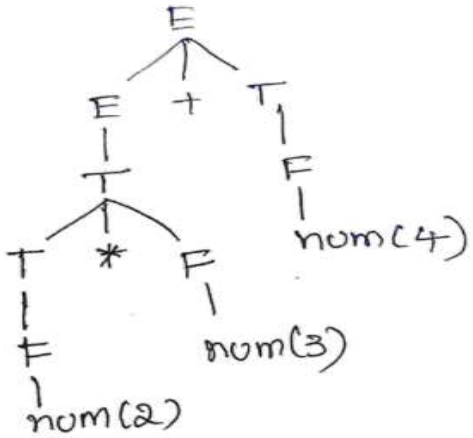


Abstract Syntax Tree

Step 3: Constructing Parse Tree for the above grammar.

This uses the actual grammar rules (productions) to derive the expression 2 * 3 + 4.

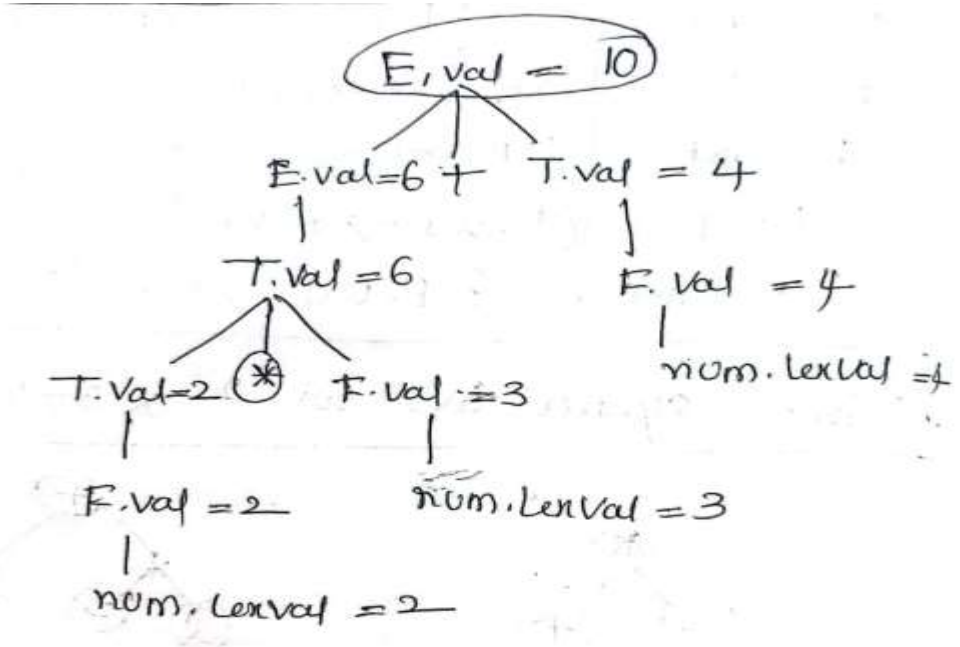
Parse Tree for 2 * 3 + 4:



Step 4: Annotated Parse Tree / Decorated Parse Tree

An annotated parse tree is a parse tree where each node is decorated with the result of semantic actions (like computed values).

Let's compute values bottom-up for 2 * 3 + 4.



- The use of SDT's to implement two important classes of SDD's:

1. The underlying grammar is LR-parsable, and the SDD is S-attributed.
2. The underlying grammar is LL-parsable, and the SDD is L-attributed.

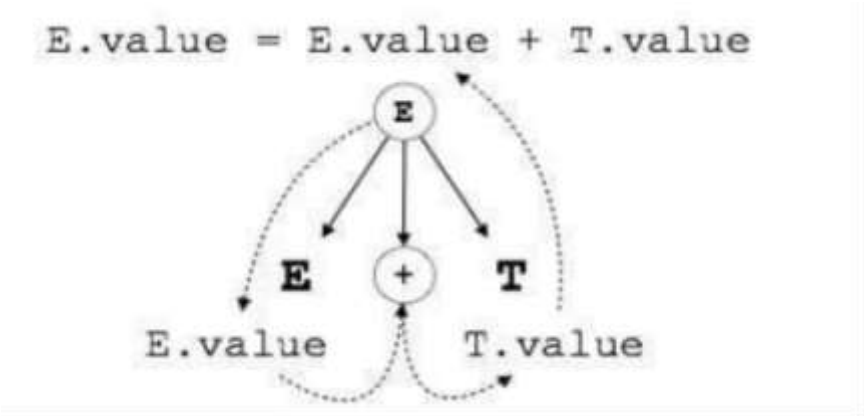
Two SDD's considered for constructing syntax trees for expressions.

The first, an **S-attributed** definition, is suitable for use **during bottom-up parsing.**

second, **L-attributed,** is suitable for use during **top-down parsing.**

S-attributed SDT

- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
- S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS.

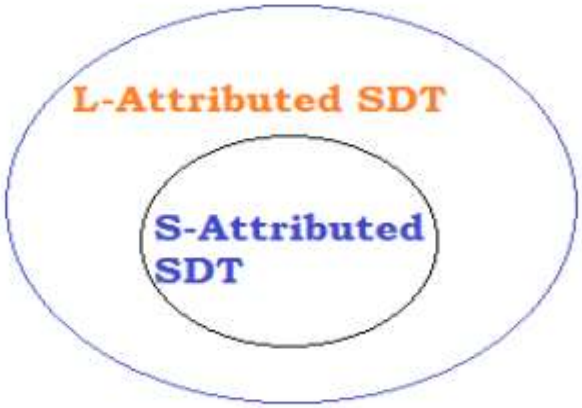


L-Attributed SDT

- If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.

Example : S->ABC,

- Here attribute B can only obtain its value either from the parent – S or its left sibling A but It can't inherit from its right sibling C. Same goes for A & C –
- A can only get its value from its parent & C can get its value from S, A, & B as well because C is the rightmost attribute in the given production.
- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner



- If a definition is S-attributed, then it is also L-attributed
- L-attributed definition encloses S-attributed definitions.

S-Attributed SDT Vs L-Attributed ADT

- Based on Synthesized attributes(i.e get values from the attribute values of their child nodes)
- Use Bottom up parsing.
- Sematic rule always written at rightmost position in RHS.

- Based on Synthesized and Inherited attributes (I.A- can take values from parent and/or siblings)
- Use Top-Down parsing.
- Sematic rules anywhere in RHS.

Applications of Syntax-Directed Translation

- The main application in this section is the construction of syntax tree.
- A syntax tree is condensed form of a Parse Tree.
- Useful for representing the expressions and statements
- The child node represents the meaningful component of the construct.

Consider the grammar that is used for Simple desk calculator. Obtain the Semantic action and also the annotated parse tree for the string 3*5+4n.

$L \rightarrow En$

$E \rightarrow E1+T$

$E \rightarrow T$

$T \rightarrow T1*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow digit$

Solution : Syntax-directed Definition for A simple desk calculator

Production	Semantic Rule
$L \rightarrow E$ **n**	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
$F \rightarrow$ **digit**	$F.\text{val} =$ **digit**.lexval

Note: all attributes in this example are of the synthesized type

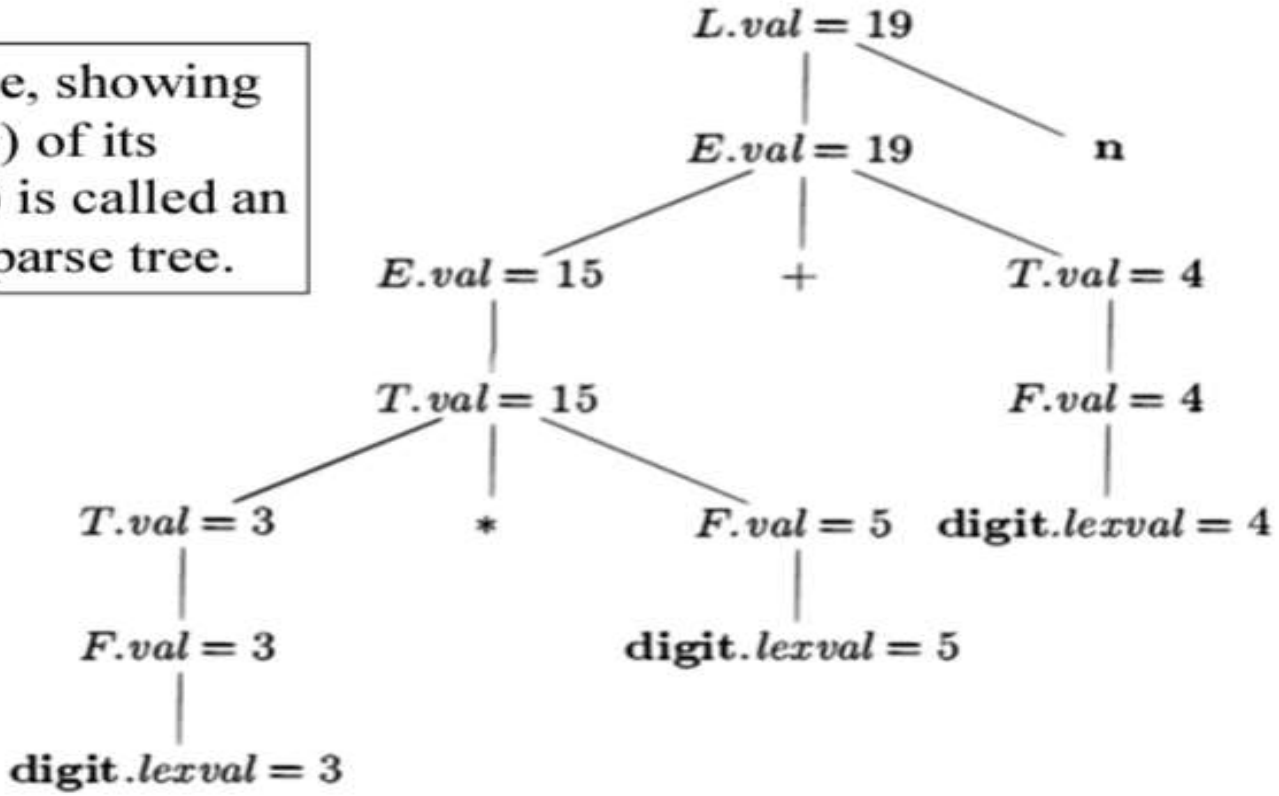
An SDD with only synthesized attributes is called S-attributed.

An SDD without side effects is called an attribute grammar

Solution : Syntax-directed Definition for A simple desk calculator

Annotated Parse Tree for 3 * 5 + 4 n

A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.



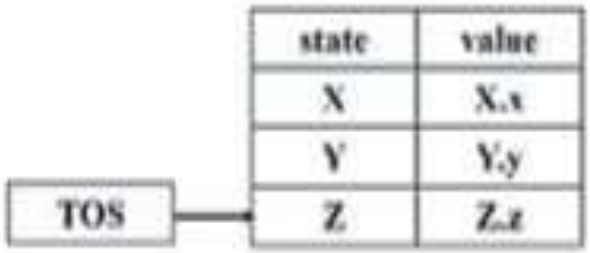
Bottom up Evaluation of Attributes

Explain in detail about Bottom-Up Evaluation of S-Attributed Definitions.

- **Syntax-directed definitions with only synthesized attributes**(S-attribute) can be evaluated by a **bottom up parser** (BUP) as input is parsed
- In this approach, the parser will keep the values of synthesized attributes associated with the grammar symbol on its stack.
- The stack is implemented as a pair of state and value.
- When a reduction is made ,the values of the synthesized attributes are computed from the attribute appearing on the stack for the grammar symbols.
- Implementation is by using an LR parser (e.g. YACC)

Bottom up Evaluation of Attributes

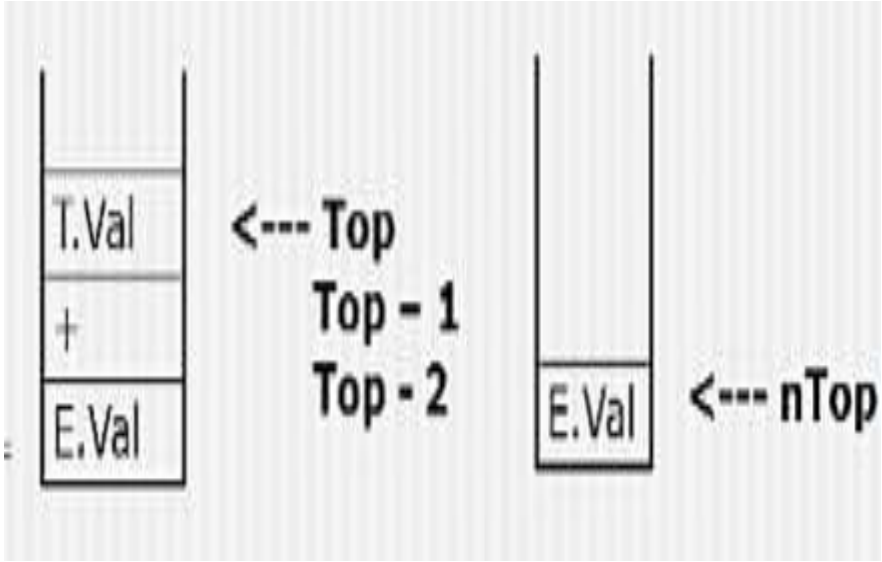
- e.g. A => XYZ and A.a := f (X.x, Y.y, Z.z)



Bottom up Evaluation of Attributes

Production	Semantic Rules
$L \rightarrow E$ **n**	*print* (*E.val*)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val + F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ **digit**	$F.val :=$ **digit**.*lexval*

- Implementation of Desk Calculator.



Bottom up Evaluation of Attributes

Synthesized Attributes on the Parser Stack

- If attribute values are placed on the stack as described, it is implement the semantic rules for the desk calculator.
- Example – Desk Calculator

Production	Semantic Rule	Code
L -> E **newline**	print(E.val)	print val[top-1]
E -> E1 + T	E.val := E1.val + T.val	val[newtop] = val[top-2]+val[top]
E -> T	E.val := T.val	/*newtop==top, so nothing to do*/
T -> T1 * F	T.val := T1.val x F.val	val[newtop] = val[top-2]*val[top]
T -> F	T.val := F.val	/*newtop==top, so nothing to do*/
F -> (E)	F.val := E.val	val[newtop] = val[top-1]
F -> **digit**	F.val := number.lexval	/*newtop==top, so nothing to do*/

Production	Value Stack Implementation
L → E n	print(E.val)
E → E + T	val(ntop) = val(top – 2) + val(top)
E → T	no action
T → T * F	val(ntop) = val(top – 2) * val(top)
T → F	no action
F → (E)	val(ntop) = val(top – 1)
F → digit	no action

Bottom up Evaluation of Attributes

INPUT	state	val	PRODUCTION USED
3*5+4n	_	_	
*5+4n	3	3	
*5+4n	F	3	F → digit
*5+4n	T	3	T → F
5+4n	T *	3 _	
+4n	T * 5	3 _ 5	
+4n	T * F	3 _ 5	F → digit
+4n	T	15	T → T * F
+4n	E	15	E → T
4n	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	F → digit
n	E + T	15 _ 4	T → F
n	E	19	E → E + T
	E n	19 _	
	L	19	L → E n

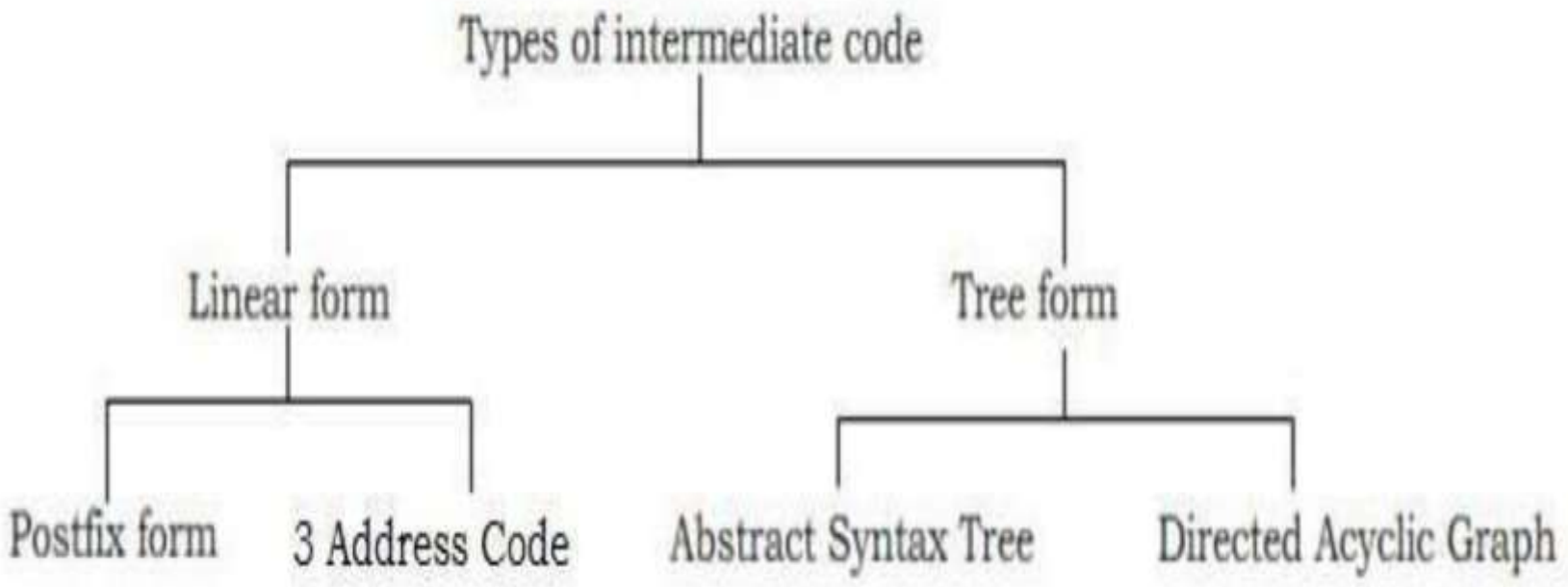
Intermediate code generation: Types of Intermediate codes, Types of Three address codes.

Intermediate code generation

- In the analysis-synthesis model of a compiler,
  - the front end of a compiler translates a source program into an independent intermediate code,
  - then the back end of the compiler uses this intermediate code to generate the target code.

- A three-address statement is an abstract form of intermediate code.
- Intermediate code is used to translate the source code into the machine code. Intermediate code lies between the high-level language and the machine language.
- Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Types of Intermediate code



Intermediate Code Representations

The following are commonly used intermediate code representations:

1) Postfix Notation

2) Syntax Tree

3) Three-Address Code

Postfix Notation: Also known as reverse Polish notation or suffix notation.

•In this notation, the operator is placed after operands,

e.g., *a* + *b*. Postfix notation positions the operator at the right end, as in *ab* +.

Example 1: The postfix representation of the expression

(a + b) * c is : ab + c *

(a – b) * (c + d) + (a – b) is : ab – cd + *ab -+

2. **Syntax Tree:** A syntax tree serves as a condensed representation of a parse tree.

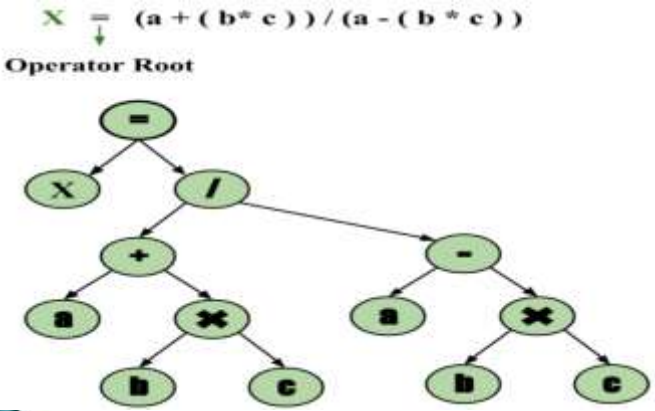
syntax tree is usually used when represent a program in a tree structure.

•The operators are represented as parent and intermediate nodes, identifiers and numbers are represented as leaf nodes.

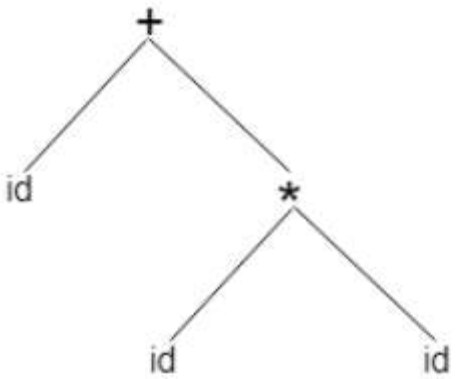
•Creating a syntax tree involves strategically placing parentheses within the expression.

•The syntax tree not only condenses the parse tree but also offers an improved visual representation of the program's syntactic structure

•**Example:** x = (a + b * c) / (a – b * c)



sentence id + id * id would have the following syntax tree:



3. Three-Address Code: A three address statement involves a maximum of three references, consisting of two for operands and one for the result.

•A sequence of three address statements collectively forms a three address code.

•The typical form of a three address statement is expressed as

$x = y\ op\ z$, where x, y, and z represent memory addresses.

•Each variable *(x, y, z)* in a three address statement is associated with a specific memory location.

Example: The three address code for the expression a = b + c * d:
First, convert this statement into Three Address code

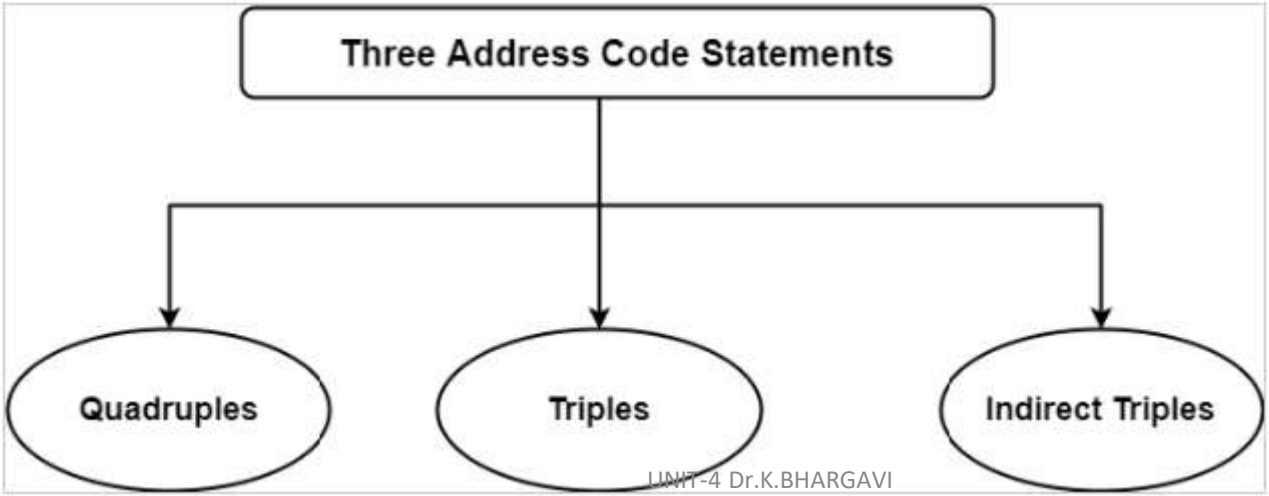
∴ Three Address code will be

t1 = c ∗ d
t2 = b + t1
a = t2

.t1 , t2 are temporary variables.

Implementation of Three Address Code

- There are 3 representations of three address code.
  1. Quadruple
  2. Triples
  3. Indirect Triples



Quadruple

- It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.
- **Advantages**
  - Easy to rearrange code for global optimization.
  - One can quickly access value of temporary variables using symbol table.
- **Disadvantage**
  - Contain lot of temporaries.
  - Temporary variable creation increases time and space complexity.

Example a = b + c * d

Example a = b + c * d

Quadruple

Location	Operator	arg 1	arg 2	Result
(0)	*	c	d	t1
(1)	+	b	t1	t2
(2)	=	t2		a

The content of fields arg 1, arg 2 and Result are pointers to symbol table entries for names represented by these entries.

UNIT-4 Dr.K.BHARGAVI

Triples

- This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used.
- So, it consist of only three fields namely op, arg1 and arg2.
- **Disadvantage**
  - Temporaries are implicit and difficult to rearrange code.
  - It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also.
  - With help of pointer one can directly access symbol table entry.

Triples

Triple

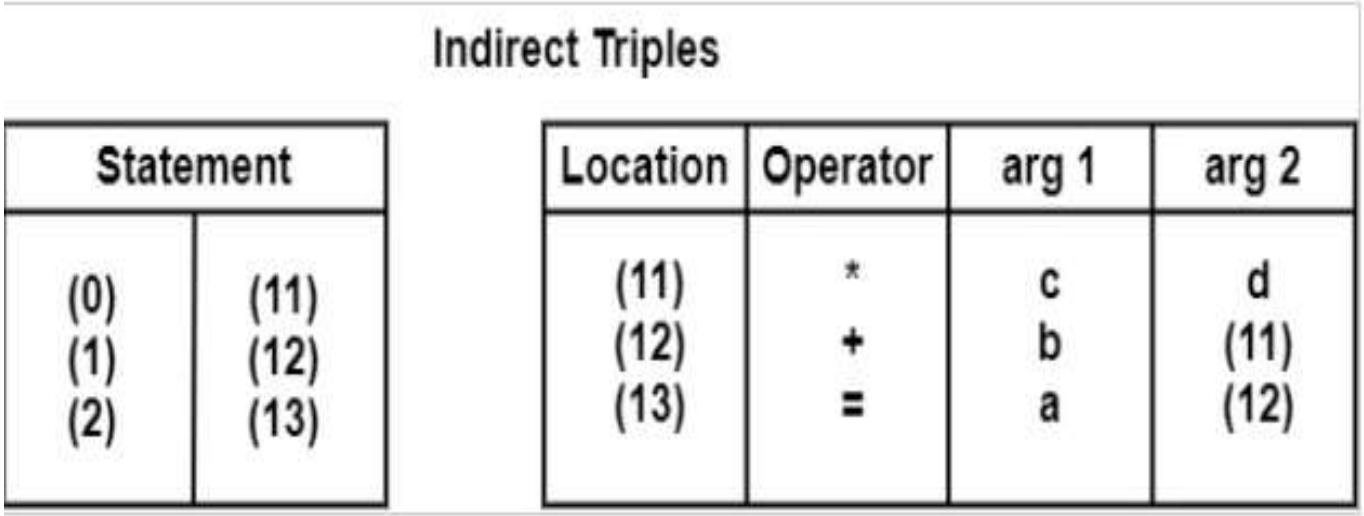
Location	Operator	arg 1	arg 2
(0)	*	c	d
(1)	+	b	(0)
(2)	=	a	(1)

Here (0) represents a pointer that refers to the result c * d, which can be used in further statements, i.e., when c * d is added with b. This result will be saved at the position pointer by (1). Pointer (1) will be used further when it is assigned to a.

Indirect Triples

- The indirect triple representation uses an extra array to list the pointers to the triples in the desired order. This is known as indirect triple representation.
- Its similar in utility as compared to quadruple representation but requires less space than it.
- Temporaries are implicit and easier to rearrange code.

Indirect Triples



In this, it can only need to refer to a pointer (0), (1), (2)which will further refer pointers(11), (12), (13) respectively & then pointers (11), (12), (13) point to triples that is why this representation is called indirect triple representation.

6.2.5 Exercises for Section 6.2

Exercise 6.2.1: Translate the arithmetic expression $a + -(b + c)$ into:

a) A syntax tree.

b) Quadruples.

c) Triples.

d) Indirect triples.

Exercise 6.2.2: Repeat Exercise 6.2.1 for the following assignment statements:

i. `a = b[i] + c[j].`

ii. `a[i] = b*c - b*d.`

iii. `x = f(y+1) + 2.`

iv. `x = *p + &y.`

Example:

- Construct Quadruples ,Triples, Indirect triples for expression

a = b * – c + b * – c.