Date
Page No.

Unit-2

#) Syntax Analysis

- It is second phase of compiler design process in which the given input string is checked for the confirmation of rules and structure of the formal Grammer.
- Syntax Analysis in Compiler design process comes after the lexical Analysis phase. It is also known as Parse Tree
- It verifies the grammatical mistake of the source code.
- To verify the syntax of code, the language must be defined by the CFG.

#) Role of Parser

- Parser is responsible for carrying out Syntax Analysis
- Syntax Analysis works hand in hand with lexical Analysis
- Each time the Parser makes a request for a token the lexer passes it.

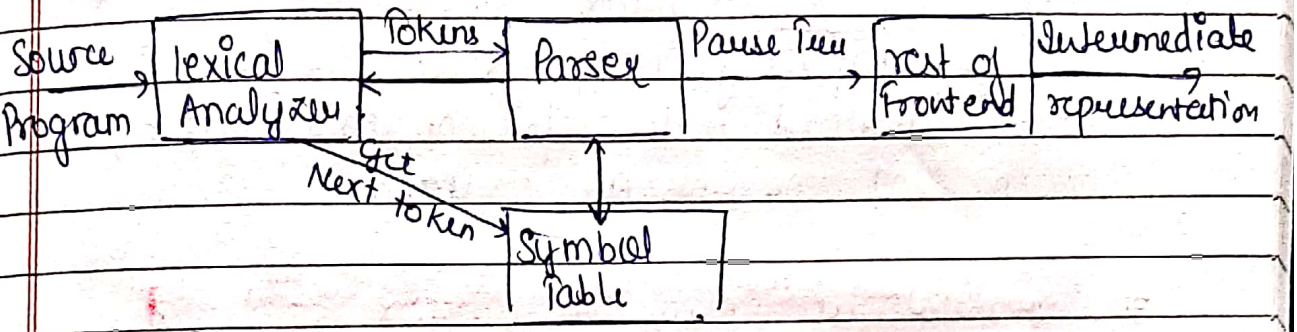
Main objective of Parser :-

1) To identify the programming language constructs & determine whether the input is valid or not. If the input is valid, then it will produce a parse tree.
2) If the input is invalid. then the parser will report an error and hence it will not produce any parse tree.

Star
Date
Page No.

Functions :-

- It verifies the structure generated by the tokens based on the Grammar.
- It constructs the Parse Tree
- It reports the errors
- It performs error recovery

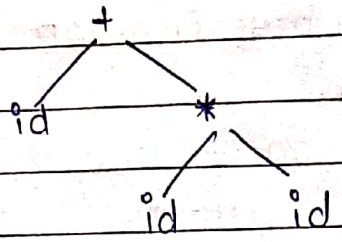


#) Abstract Syntax Tree (AST)

- It is a kind of tree representation of the abstract syntatic structure of source code written in a programming language.
- Each Node of tree denotes a construct occurring in the source code.
- AST are mainly used in compiler to check code for their accuracy.
- If the generated tree has errors, the compiler prints an error message.
- It often serves as an intermediate representation of the program through several stages that the compiler requires and has a strong impact on the final output of the compiler.

Flow chart :-

id + id * id



- These are more denser than Parse Tree.
- They do not provide every characteristic information from the real syntax.
- It is compact form of Parse Tree.
- Each interior node represent an operator, each leaf represents an operand.

#) Terminologies related to Syntax Analysis

① Context-free Grammers →

- A CFG, naturally, describes the hierarchial
- A CFG can be mathematically described as

$G = \{V_N, V_T, P, S\}$

where V_N → Set of Non-terminal Symbol
V_T → Set of Terminal symbol
P → Production
S → Start Symbol

V_N are syntatic variables that denote set of strings
V_T → terminals are basic symbols from which strings are formed

P → Production of a Grammar Specify the manner in which terminals and non-terminals can be combined to form strings.
S → It is the point from where Production Begins.

(2) Derivation →
The sequence of intermediatary strings that are Generated to expand the start symbol, if a Grammar has to arrive at a desired string is known as derivation.
There are two type :-
1) left Most derivation
2) Right Most derivation

⇒ left Most Derivation
- The left Most Non Terminal Symbol is always replaced at each Step.
- The Intermeditary strings produce during LMD is known as left Sentential Form.

⇒ Right Most Derivation
- The RMD Non terminal is always replaced at each Step.
- Sentential form of an input is scanned and replaced from right to left.
- The Intermeditary strings produce during RMD is known as Right Sentential Form.

Star
Date
Page No.

(3) Ambiguous Grammer

A grammer is said to be ambiguous if it produces more than one parse tree. In other words, if a grammer produces more than one LMD or RMD it is considered to ambiguous.

(4) Left Recursion

A grammer is said to be left Recursive if any of the production is in form of

$$A \rightarrow A\alpha | \beta$$

where A is Non terminal and it derives $A\alpha$ which begins with the same non terminal from which it is derived.

The presence of left Recursion makes it difficult for the parser. Hence it must be eliminated in following way

$$A \rightarrow BA'$$
$$A' \rightarrow \alpha A' | \epsilon$$

(5) Left Factoring -

If more than one grammer production rules has a common prefix string then the top down Parser the string in hand. Left factoring transforms the grammer to make it useful for top down parser.

In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions

#) Limitation of Syntax Analyzers

- It cannot determine if a tokens is valid
- It cannot determine if a token is declared before it is being used
- It cannot determine if a token is initialized before it is being used.
- It cannot determine if an operation performed on a token type is valid or not.

#) Types of Parsing

There are two types of Parsing :-

1) Top down Parsing

Top down Parsing is a method of parsing the input string provided by the lexical analyzer. In top-down Parsing the input string and then generate parse tree of it.

Construction of Parse Tree starts from the root Node i.e. the start symbol of the grammar. Then using leftmost derivation it derives a string that matches the input string.

Star

Date

Page No.

In top-down approach construction of the parse tree starts from the root node and end up creating the leaf nodes. Here the leaf node presents the terminal that match the terminals of the input string.

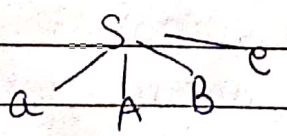
To derive the input string, first, a production in Grammar with a start symbol is applied Now at each step, the parser has to identify which production rule of a non-terminal must be applied in order to derive the input string. The Next step is to match the terminals in the production with the terminals of input string.
In top down parsing, the parse tree is generated from top to Bottom.

Example

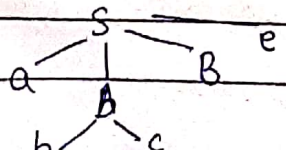
Q.1 $S \rightarrow aABe$ → Start symbol
$A \rightarrow bc$
$B \rightarrow d$
$W \rightarrow abcde$ → Input string

Soln Step1

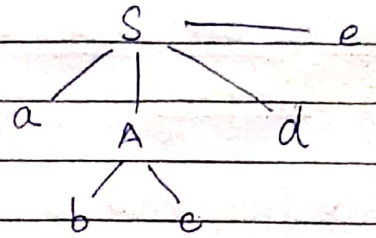
$S \rightarrow a\ ABe$



Step2



Step3



Que2.

Grammer ⇒ $E \to TE'$

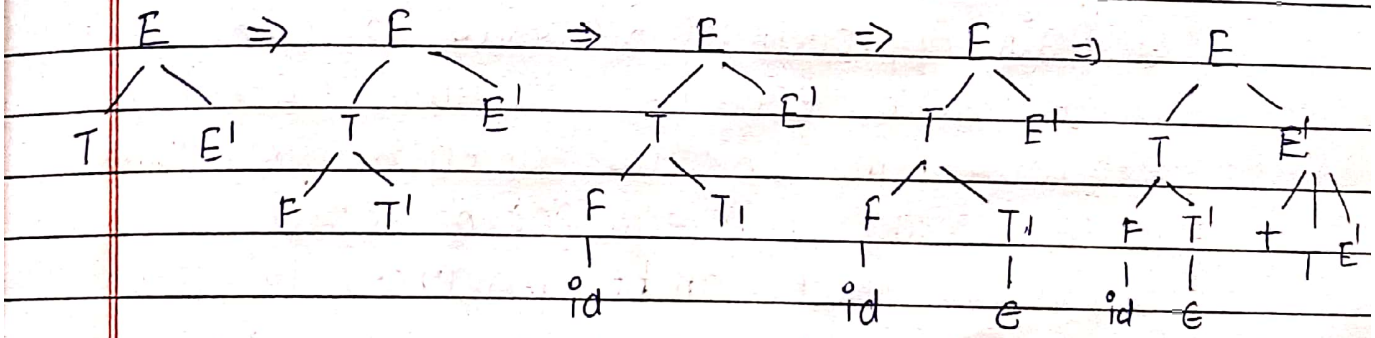
$E' \to +TE' \mid \epsilon$

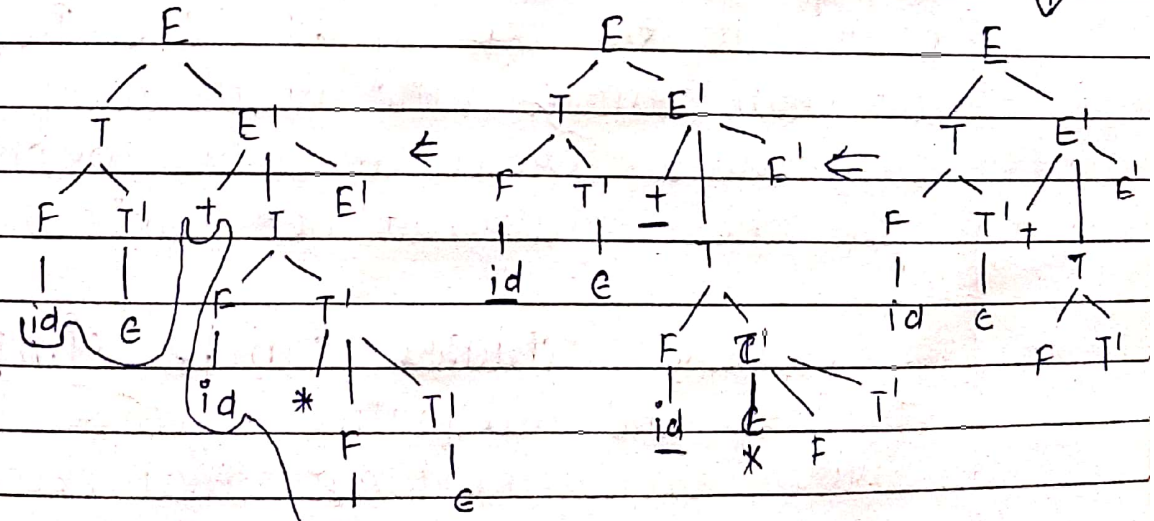
$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to (E) \mid id$

W → id + id * id





Here we got input string id + id * id

Date
Page No.

Classification of top down parser is :-

1) Recursive Descent Parsing

Recursive Descent Parsing / Parser uses the technique of Top-Down Parsing without / with backtracking. It can be defined as a parser that uses the various recursive procedure to process the input string with no backtracking. It can be simply performed using a recursive language. The first symbol of the string of R.H.S of production will uniquely determine the correct alternative to choose.

The Main Approach of Recursive descent passing is to relate each non terminal with a procedure. The objective of each procedure is to read a sequence of input characters that can be produced by the corresponding non terminal and return a pointer to the root of the Parse Tree for the non terminal. The structure of the procedure is prescribed by the productions for the equivalent non terminal.

The recursive procedures can be simply to write and adequately effective if written in a language that executes the procedure call effectively. There is a

procedure for each non terminal in the grammer. It can consider a Global variable lookhead, holding the current input token and a procedure match is the action of recognising the next token in the parsing process and advancing the input stream pointer, such that lookahead points to the next token to be parsed. match() is effectively a call to lexical analyzer to get the next token.

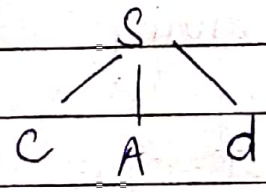
Example

$S \rightarrow cAd$

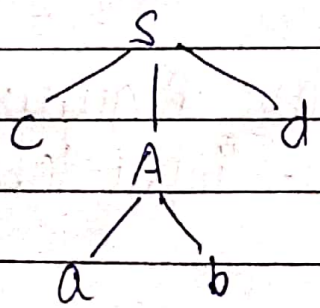
$A \rightarrow ab/a$

To construct a parse Tree top down for the i/p string $w = cad$, begin with a tree consisting of a single node labelled S and the input pointer to c, the first symbol of w.

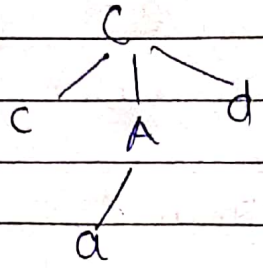
Step1



Step2



Step 3 (Backtracking) $A \to a$



Here this is parse tree for given input string -

#) Shift Reduce Parser

- Shift Reduce Parsing is a process of reducing a string to the start symbol of Grammer
- Shift Reduce Parsing uses a stack to hold the Grammer and an input tape to hold the string

A String $\xrightarrow{\text{reduce to}}$ the starting symbol

- This reduction can be achieved by directly handling the rightmost derivation from the starting symbol to the input string.
- The Shift Reduce Parsing is a type of Bottom-up passing as it Generates a Parse tree from the leaves (Bottom) to the root (up).
- Two data structures are Required to perform Shift Reduce :- input tape (Buffer), Stack

Star
Date
Page No.

Basic operations performed by Shift Reduce Parser

(1) Shift :- This operation involves moving the current symbol from the input buffer onto the stack

(2) Reduce :- when the parser knows the right hand of the handle is at the top of stack, the reduce operation applies the applicable production rules i.e. pops out the RHS of the production rule from the stack and pushes the LHS of production rule onto the stack.

(3) Accept :- After Repeating the Shift & Reduce operations if the stack contains the starting symbol of the input string and the input buffer is empty i.e. Includes the $ symbol, the input string is said to be accepted.

(4) Error :- If the Parser cannot perform the Shift or Reduce operation, also the string is not accepted, then it is said to be error state.

Note:-

* If the priority of incoming operator is more than the priority of in stack operator then Shift action is performed
* If the priority of incoming operator is same or less then Reduce action is performed

Date
Page No.

Example Consider the Grammer

$S \to S+S$

$S \to S-S$

$S \to (S)$

$S \to a$

where input string is $a_1 - (a_2 + a_3)$

Soln →

Stack	Input string	Parsing Actions
$	$a_1-(a_2+a_3)$$	Shift a_1
$$a_1$	$-(a_2+a_3)$$	reduce by $S \to a$
$$S$	$-(a_2+a_3)$$	Shift -
$$S-$	$(a_2+a_3)$$	Shift (
$$S-($	$a_2+a_3)$$	Shift a_2
$$S-(a_2$	$+a_3)$$	reduce $S \to a$
$$S-(S$	$+a_3)$$	Shift +
$$S-(S+$	$a_3)$$	Shift a_3
$$S-(S+a_3$	$)$$	reduce $S \to a$
$$S-(S+S$	$)$$	Shift)
$$S-(S+S)$	$$	reduce $S \to S+S$
$$S-(S)$	$$	reduce $S \to (S)$
$$S-S$	$$	reduce $S \to S-S$
$$S$	$$	Accept

Example Consider the Grammer

$E \to E-E$

$E \to E \times E$

$E \to id$

Parse the input string id - id x id using a Shift Reduce Parser

Soln

Stack	Input String	Parsing Action
$	id - id x id $	Shift id
$ id	- id x id $	reduce E → id.
$ E	- id x id $	Shift -
$ E -	id x id $	Shift id
$ E - id	x id $	reduce id → E
$ E - E	x id $	shift x
$ E - E x	id $	Shift id
$ E - E x id	$	reduce id → E
$ E - E x E	$	reduce E → E x E
$ E - E	$	reduce E → E - E
$ E	$	Accept.

#) Operator Precedence Parsing

- Operator Precedence grammar is kinds of Shift Reduce Parsing Method. It is applied to a small class of operator grammers.
- Operator Precedence Grammer could be either ambiguous or unambiguous.
- Operator Precedence is provided with precedence rules.

Date
Page No.

- Operator Precedence can be only established between the terminals of grammer. It ignores the non terminal.
- A grammer is said to be operator precedence grammer if it has two properties :-
  → No RHS of any production has ∈.
  → No two non terminals are adjacent.
- The Disadvantage of the operator precedence relation table is that if there are 'n' no. of symbols then we require a table of n*n to store them. On the other hand, by using the operator function table, to accomodate n number of symbols we require a table of 2*n
- The Advantage of using operator Precedence grammer is simple and enough powerful for expression in programming languages.
- There are three Operator precedence relations :-
  a ·> b ⇒ means a has higher precedence than b
  a <· b ⇒ means a has lower precedence than b
  a ≐ b ⇒ means a and b both have same precedence

operator relation table

eg) E → E + E | E * E | id

	id	+	*	$
id	×	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	×

id → highest prec.
$ → lowest "
*, + are left ass. so left one has more prec.

- Two id's will never be compared because they will never come side by side.
- Identifier will be given highest precedence compared to any other operator.
- $ has least precedence to any other operator.

Example
$S \to a \mid \uparrow \mid (T)$
$T \to T,S \mid S$

The operator Precedence table can be given as:

	a	↑	(	)	,	$
a				$\cdot>$	$\cdot>$	$\cdot>$
↑				$\cdot>$	$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$<\cdot$	$=$	$<\cdot$	
)				$\cdot>$	$\cdot>$	$\cdot>$
,	$<\cdot$	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$	
$	$<\cdot$	$<\cdot$	$<\cdot$			Accept

Input String = (a,a)

Stack	Input String	Actions
$	(a,a)$	$ <· (, Shift
$(	a,a)$	$(<· a, shift
$(a	,a)$	a ·>, , Pop (a) reduce by S→a, T→S
$(S	,a)$	(<· , , shift
$(T	,a)$	, <· a , shift
$(T,	a)$	a ·>), pop a reduce by S→a
$(T,a	)$	, ·>), pop T,S reduce by T→T,S
$(T,S	)$	

$ (T	)$	(≐) , shift
$ (T)	$	) ⋗ $, pop (T) reduce by S
$ S	$	Accepted.

Example

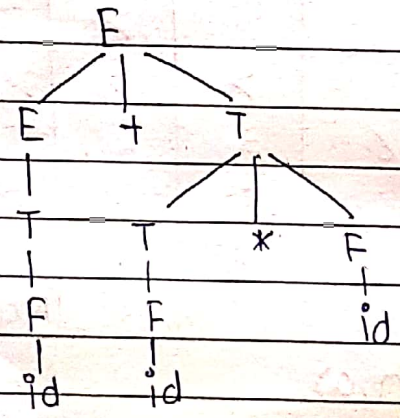
E → E+T / T

T → T* F / F

F → id

Input String → id+id* id

Step Parse Tree :-



Operator Precedence Table

	id	+	*	$
id	x	⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	≐

Date
Page No.

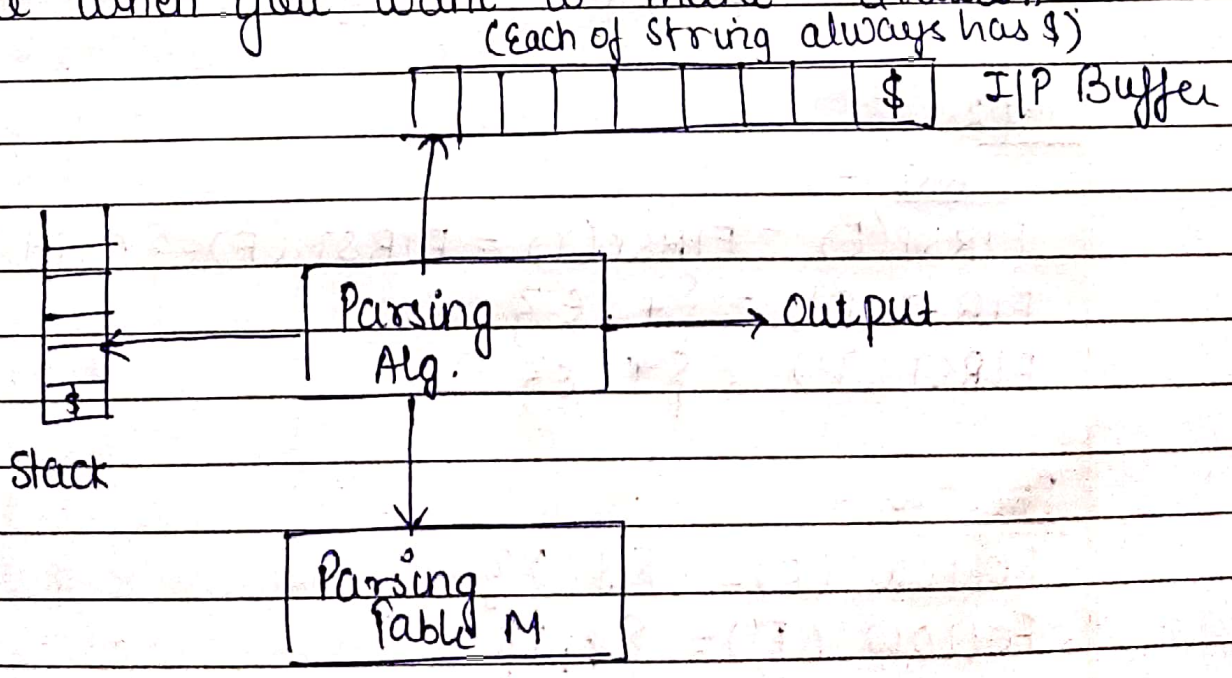
#) **Predective Parser (LL Parser)**

A predective Parser is a recursive descent parser with no backtracking or backup. It is a top down parser that does not require backtracking. At each step, the choice of the rule to be expanded is made upon the next terminal symbol.

Non-Recursive predective parsing or table driven is also known as LL(1) parser.

In LL(1),

- First L represent scanning of input will be done from left to Right manner.
- Second L represent we will use left most derivation tree.
- 1 represents the number of look ahead which means how many symbols are you going to see when you want to make division.



Star
Date
Page No.

Algorithm

1. Remove left Recursion from Grammer if present
2. Calculate First() and Follow() for non-terminal
3. Construct the predective parsing table.
4. Perform Parsing of input string with help of predective Parsing table.

Ex →

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

Step1 Eliminate left Recursion from Grammer

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

Step2

FIRST :-

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
FIRST(E') = {+, ∈}
FIRST(T') = {*, ∈}

FOLLOW :-

FOLLOW(E) = {$,)}
FOLLOW(E') = {$,)}
FOLLOW(T) = {+, $,)}
FOLLOW(T') = {+, $,)}

FOLLOW (F) = {*, +, $,)}

Step 3

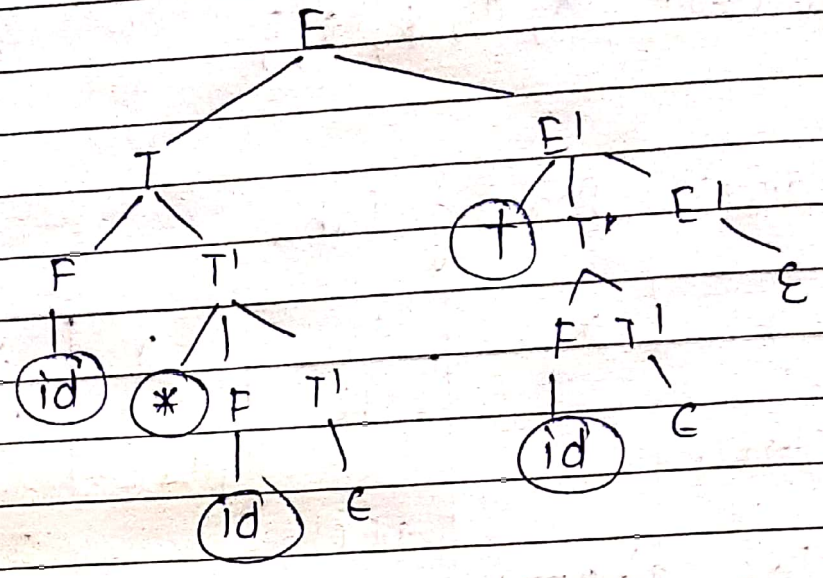
Predective Parsing Table

	+	*	(	)	id	$
E			E→TE'		E→TE'	
E'	E'→+TE'			E'→ϵ		E'→ϵ
T			T→FT'		T→FT'	
T'	T'→ϵ	T'→*FT'		T'→ϵ		T'→ϵ
F			F→(E)		F→id	

Step 4

W = id* id + id

Stack	Input	Output
$E	id*id+id$	
$E'T	id*id+id$	E→TE'
$E'T'F	id*id+id$	T→FT',
$E'T'id	id*id+id$	F→id
$E'T'	*id+id$	T'→*FT'
$E'T'F*	*id+id$	
$E'T'F	id+id$	F→id
$E'T'id	id+id$	
$E'T'	+id$	T'→ϵ
$E'	+id$	E'→+TE'
$E'T+	+id$	
$E'T	id$	T→FT'
$E'T'F	id$	F→id
$E'T'	$	T'→ϵ
$E'	$	E'→ϵ
$	$	



Hence, id * id + id derived.

#) FIRST() & FOLLOW()

First(A) contains all terminals present in first place of every string derived by A.

Rules

1) FIRST(Terminal) = Terminal
2) FIRST (ε) = ε
3) If A → Y1, Y2, Y3 ... Yn
   FIRST(A) = FIRST (Y1)

Eg -

(1) S → ABC | ghi | jkl
A → a | b | c
B → b
D → d

FIRST(D) = {d}
FIRST(B) = {b}
FIRST(A) = {a, b, c}
FIRST(S) = FIRST(A) = {a, b, c}
FIRST(ghi) = {g}
(jkl) = {j}

Star

Date

Page No

(2) $S \to ABC$

$A \to a \mid b \mid \epsilon$ FIRST(A) $= \{a, b, \epsilon\}$

$B \to c \mid d \mid \epsilon$ FIRST(B) $= \{c, d, \epsilon\}$

$C \to e \mid f \mid \epsilon$ FIRST(C) $= \{e, f, \epsilon\}$

FIRST(S) = FIRST($\overset{\epsilon}{A}\overset{\epsilon}{B}C$) $= \{a, b, c, d, e, f, \epsilon\}$

(3) $S \to ACB \mid Cbb \mid Ba$

$A \to da \mid BC$

$B \to g \mid \epsilon$ FIRST(B) $= \{g, \epsilon\}$

$C \to h \mid \epsilon$ FIRST(C) $= \{h, \epsilon\}$

FIRST(A) $= \{d\} \cup$ FIRST(BC)

$= \{d\} \cup \{g, h, \epsilon\}$

$= \{d, g, h, \epsilon\}$

FIRST(S) = FIRST(ACB) $\cup$ FIRST(Cbb) $\cup$ FIRST(Ba)

$= \{d, g, h, b, a, \epsilon\}$

FOLLOW ( )

FOLLOW(A) contains of all terminals present immediate in right of 'A'.

Rules

1. Follow of start Symbol is $

FO(A) = $

2. $S \to ACD$
$C \to a \mid b$
$FO(A) = FIRST(C) = \{a, b\}$
$FO(D) = FO(S) = \{\$\}$

3. $S \to aSbS \mid bSaS \mid \epsilon$
$FO(S) = \{\$, b, a\}$

* Follow never contain ϵ.

Eg. $S \to AaAb \mid BbBa$
$A \to \epsilon$
$B \to \epsilon$

$FO(A) = \{a, b\}$
$FO(B) = \{b, a\}$

Eg. $S \to ABC$
$A \to DEF$
$B \to \epsilon$
$C \to \epsilon$
$D \to \epsilon$
$E \to \epsilon$
$F \to \epsilon$

$FO(A) = FIRST(B) = FIRST(C) = \{\$\}$
$FO(D) = FIRST(E) = FIRST(F) = FO(A) = \{\$\}$

Eg. Find First() & Follow() of given :-
$S \to aBDh$, $C \to bC \mid \epsilon$, $E \to g \mid \epsilon$
$B \to CE$, $D \to EF$, $F \to f \mid \epsilon$

Soln
FIRST(S) $\Rightarrow \{a\}$; FIRST(D) = FIRST(E) ∪ FIRST(F)
$= \{g, f, \epsilon\}$
FIRST(C) $\Rightarrow \{b, \epsilon\}$
FIRST(B) $\Rightarrow \{c\}$; FIRST(F) = $\{f, \epsilon\}$
FIRST(E) $\Rightarrow \{g, \epsilon\}$

Follow() :-

FO(S) = $; FO(B) = {FIRST(D) − ε} ∪ FIRST(h)
$= \{g, f, h\}$

FO(C) = FO(B) = $\{g, f, h\}$
FO(D) = $\{h\}$
FO(E) = FIRST(F) ∪ FO(D) = $\{f, h\}$
FO(F) = FO(D) = $\{h\}$

#) SLR Parser
SLR represent Simple LR parsing. It is also called as LL(0) Parser. It is simple and economical to execute. But it fails to make a parsing table for some classes of grammer. To construct SLR(1) parsing table we use canonical collection of LR(0) item.
SLR(1) :- A grammar having an SLR parsing table is said to be SLR(1).

In the SLR(1) parsing, we place the reduce move only in the follow left hand side.

Steps or Working of SLR parser

SLR parsing can be done if context free grammer will be given. In LR(0) means there is no look ahead symbol.

Write Context free Grammer

⇓

Construct Canonical collection of LR(0) Items
a) Construct Augmented Grammer
b) Find Closure c) Find goto.

⇓

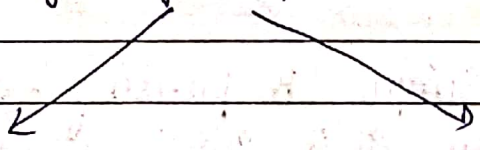
Find FIRST and FOLLOW for Grammer symbols

⇓

Construct SLR Parsing Table

⇓

Parsing of input String



String Accepted

String Not Accepted

Star
Date
Page No.

Ex → r1) E → E+T
r2) E → T
r3) T → T*F
r4) T → F
r5) F → (E)
r6) F → id

Step1 Augmented grammer
E' → E
E → E+T
E → T
T → T*F
T → F
F → (E)
F → id

Step2 Canonical LR(0) collection (closure & goto)

I_0: E' → ·E
E → ·E+T
E → ·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id

goto (I_0, E)
I_1: E' → E·
E → E·+T

goto (I_0, T)
I_2: E → T·
T → T·*F

goto (I_0, F)
I_3: T → F·

goto (I_0, ()
I_4: F → (·E)
E → ·E+T
E → ·T
T → ·T*F
T → ·F
F → ·(E)
F → ·id

goto (I_0, id)
I_5: F → id·

Star

Date

Page No.

goto(I_1, +)

I_6 : E → E+·T

T → ·T*F

T → ·F

F → ·(E)

F → ·id

goto(I_2, *)

I_7 : T → T*·F

F → ·(E)

F → ·id

goto(I_4, E)

I_8 → F → (E·)

E → E·+T

goto(I_6, T)

I_9 → E → E+T·

E → T·*F

goto(I_7, F)

I_{10} → T → T*F·

goto(I_8,))

I_{11} : F → (E)·

SLR Parsing Table

Step 4

state	action						goto		
	+	*	(	)	id	$	E	T	F
0			S_4		S_5		1	2	3
1	S_6					acc			
2	r_2	S_7	r_2	r_2	r_2	r_2			
3	r_4	r_4	r_4	r_4	r_4	r_4			
4			S_4		S_5		8	2	3
5	r_6	r_6	r_6	r_6	r_6	r_6			
6			S_4		S_5			9	3
7			S_4		S_5				10
8	S_6			S_{11}					
9		S_7							
10									
11									

Step3

Fo(E) = {$, +,)}

Fo(T) = {$, +,), *}

Fo(F) = {$, +,), *}

CLR Parsing

The CLR parsing stands for canonical LR parser. It is a more powerful LR parser. It make use of lookhead symbols. This method uses a large set of item called LR(1) items. The main difference between LR(0) and LR(1) item is that, in LR(1) items. its possible to carry more information in a state, which will rule out useless reduction states. This extra information is incorporated into the state by the lookahead symbol.

The General Syntax :-

$$[A \to \alpha . B, a]$$

Where $A \to \alpha . B$ is production and a is a terminal or right end marker $.

LR(1) items = LR(0) items + look ahead

i) Closure

$$A \to \alpha . B\beta , a$$
$$B \to . \gamma , \quad FIRST(\beta a)$$

ii) Reduce

$$A \to \alpha . , a$$

Steps in CLR :-

1. Writing augmented grammer
2. LR(1) collection of items
3. Defining goto & action in CLR Parsing Table.

S → CC
C → cC
C → d

Step1 Augmented Grammer

S' → S
S → CC
C → cC
C → d

Step2 LR(1) Items

$I_0 \Rightarrow S' \to \cdot S, \$$
$S \to \cdot CC, \$$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

goto(I_0, S)
$I_1 \Rightarrow S' \to S \cdot, \$$

goto(I_0, C)
$I_2 \Rightarrow S \to C \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

goto(I_0, C)
$I_3 \Rightarrow C \to c \cdot C, c/d$
$C \to \cdot cC, c/d$
$C \to \cdot d, c/d$

goto(I_0, d)
$I_4 \Rightarrow C \to d \cdot, c/d$

goto(I_2, C)
$I_5 \Rightarrow S \to CC \cdot, \$$

goto(I_2, c)
$I_6 \Rightarrow C \to c \cdot C, \$$
$C \to \cdot cC, \$$
$C \to \cdot d, \$$

goto(I_2, d)
$I_7 \Rightarrow C \to d \cdot, \$$

goto(I_3, C)
$I_8 \Rightarrow C \to cC \cdot, c/d$

goto(I_6, C)
$I_9 \Rightarrow C \to cC \cdot, \$$

Star

Date

Page No.

step3 CLR Parsing Table

S→Represent Shift Action
R→Represent Reduce Action
acc→accept
Blank represent ~~Error~~

state	Action			goto	
	c	d	$	S	C
0	S_3	S_4		1	2
1			acc		
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_3		
8	R_2	R_2			
9			R_2		

#) LALR Parser

LALR Parser is look Ahead LR Parser. It is intermediate in power between SLR and CLR. It is the compaction of CLR parser and hence tables obtained in this will be smaller than CLR parsing table. In LALR parsing, the LR(1) items which have same production but look ahead are combined to form a single set of items.

Steps,

① writing Augmented Grammer
② LR(1) collection of Items to be found
③ Defining 2 functions: goto & action in LALR parsing table.

Example

$S \to CC$
$C \to cC$
$C \to d$

Soln

Step1 Augmented Grammer

~~$S' \to .S, \$$~~	$S' \to S$
~~$S \to .CC, \$$~~	$S \to CC$
~~$C \to .cC, c/d$~~	$C \to cC$
~~$C \to .d, c/d$~~	$C \to d$

Step2

LR(1) items

$I_0 \to S' \to .S, \$$
$S \to .CC, \$$
$C \to .cC, c/d$
$C \to .d, c/d$

goto (I_0, S)

$I_1 \to S' \to S., \$$

goto (I_0, C)

$I_2 \to S \to C.C, \$$
$C \to .cC, \$$
$C \to .d$

goto (I_0, c)

$I_3 \to C \to c.C, c/d$
$C \to .cC, c/d$
$C \to .d, c/d$

goto (I_0, d)

$I_4 \to C \to d., c/d$

goto (I_2, C)

$I_5 \to S \to CC., \$$

goto (I_0, C)

$I_6 \to C \to c.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

goto (I_3, d)

$I_7 \to C \to d., \$$

goto (I_3, C)

$I_8 \to C \to cC., c/d$

goto (I_6, C)

$I_9 \to C \to cC., \$$

LALR(1) collection

Combining I_3, I_6

$I_{36} \to C \to c.C, c/d/\$$
$C \to .cC, c/d/\$$
$C \to .d \quad c/d/\$$

Combining I_4 & I_7

$I_{47} \Rightarrow C \to d\cdot,\ c/d/\$$

I_8 & I_9

$I_{89} \Rightarrow C \to cC\cdot,\ c/d/\$$

LALR Parsing Table from CLR Parsing Table

State	Action			goto	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			acc		
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		

⇒ Ambiguous Grammer

Example: Check whether Given Grammer is ambigtious or not for string 'ibtibtibtaca".

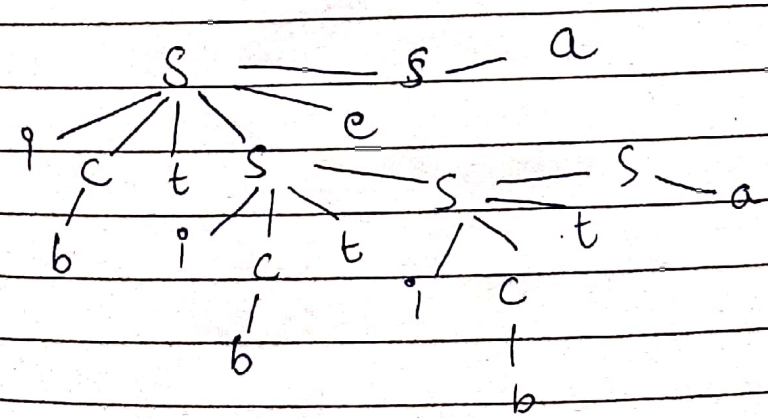
$S \to iCtS$

$S \to iCtSeS$

$S \to a$

$C \to b$

Soln 1)

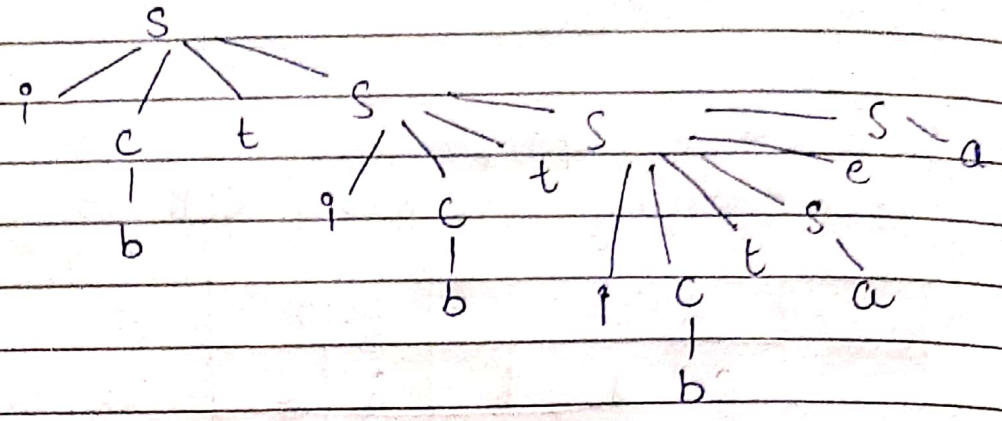


Star

Date

Page No.

a)



Hence,
Given grammer is ambiguous grammer.