CD Unit-II - Compiler design

Compiler Design (Jawaharlal Nehru Technological University, Hyderabad)

UNIT - II

2 Syntax Analysis

2.1 Introduction to Syntax Analysis

Q.1 Explain the term - syntax analysis.

☞ [JNTU : Part A, Marks 2]

Ans. : Syntax analysis is a process which takes the input string w and produces a syntax tree. If there exists any error in the syntax of the programming construct then the syntax error messages are generated.

Q.2 Explain the reasons for separating lexical analysis phase from syntax analysis.

☞ [JNTU : Part A, Marks 2]

Ans. : The lexical analyzer scans the input program and collects the tokens from it. On the other hand parser builds a parser tree using these tokens. These are two important activities and these activities are independently carried out by these two phases. Separating out these two phases has two advantages - Firstly it **accelerates** the process of **compilation** and secondly the **errors** in the source input can be **identified precisely.**

2.2 Context Free Grammar

Q.3 Define a context free grammar.

☞ [JNTU : Part A, May-18, March-16, Marks 3]

Ans. : Definition :

The context free grammar can be formally defined as a set denoted by $G = (V, T, P, S)$ where V and T are set of non-terminals and terminals respectively. P is set of production rules, where each production rule is in the form of

non-terminal → non-terminals

or non-terminal → terminals

S is a start symbol.

For example,

$$P = \{ S \rightarrow S + S$$
$$S \rightarrow S * S$$
$$S \rightarrow (S)$$
$$S \rightarrow 4 \}$$

If the language is 4 + 4 * 4 then we can use the production rules given by P. The start symbol is S. The number of non-terminals in the rules P is one and the only non terminal i.e. S. The terminals are +, *, (,) and 4.

We are using following conventions.

1. The capital letters are used to denote the **non-terminals.**
2. The lower case letters are used to denote the **terminals.**

Q.4 Explain concept of derivation tree.

Ans. : Let G = (V, T, P, S) be a Context Free Grammar.

The derivation tree is a tree which can be constructed by following properties.

i) The root has label S.

ii) Every vertex can be derived from (V U T U ε).

iii) If there exists a vertex A with children R_1, R_2, ... R_n then there should be production $A \rightarrow R_1 R_2 \cdots R_n$.

iv) The leaf nodes are from set T and interior nodes are from set V.

Q.5 Explain in brief about left most and rightmost derivations.

☞ [JNTU : Part A, Dec-17, Marks 3]

Ans. : If the leftmost non terminal in a sentential form is chosen for derivation then that derivation is called leftmost derivation.

If the rightmost non terminal in a sentential form is chosen for derivation then that derivation is called rightmost derivation.

For example - Refer Q.6.

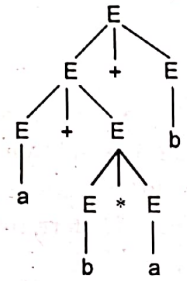
Q.6 *Consider the grammar given below -*

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid a \mid b$

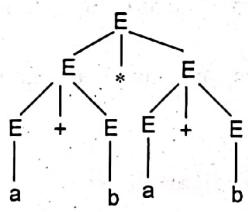
*Obtain leftmost and rightmost derivation for the string a + b * a + b.*

☞ **[JNTU : May-09, April-11, Dec.-11, Marks 8]**

Ans. : Leftmost derivation



Rightmost derivation



Q.7 *Consider*

$S \rightarrow a \mid \hat{} \mid (T)$

$T \rightarrow T, S \mid S$

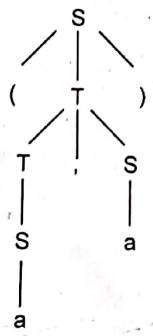
a) Draw parse trees for (a,a) and (a, (a, a))

b) Construct leftmost and rightmost derivation for (a, e) and (a, (a, e))

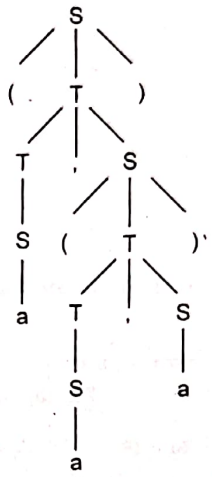
c) What language does the grammar generate.

☞ **[JNTU : Dec.-11, Marks 16]**

Ans. : a) i) Parse tree for (a, a) is



ii) Parse tree for (a, (a, a)) is



b) i) Leftmost and rightmost derivation for (a, e) we will put e = a.

S	S
(T)	(T)
(T, S)	(T, S)
(S,S)	(T, a)
(a, S)	(S, a)
(a, a)	(a, a)

ii) Leftmost and righmost derivation for (a, (a, e)) We will put e = a.

S	S
(T)	(T)
(T, S)	(T, S)
(S,S)	(T, (T))
(S, (T))	(T, (T, S))
(a, (T))	(T, (T, a))
(a, (T, S))	(T, (S, a))
(a, (S, S))	(T, (a, a))
(a, (a, S))	(S, (a, a))
(a, (a, a))	(a, (a, a))

c) This grammar generates all strings with well formedness of parenthesis.

2.3 Writing a Grammar

Q.8 *Write a CFG for the 'while' statement in 'C' language.* **[JNTU : Jan.-10, Marks 8]**

Ans. : The CFG G = {V, T, P, S} where V is a set of nonterminals, T is a set of terminal symbols, P is a set of production rules and S is a start symbol. The set of production rules

P = { S → while (condition) stmt

condition → id relop id

relop → < |> | < =| > =| ! =| = =

S → while (condition) { L }

L → stmt L | stmt

}

V = (S, condition, relop, L);

T = (while, (,), <, >, < =, > =, ! =, = =, id, stmt)

Q.9 *What is the string generated by the grammar* $A \rightarrow (A)A$. **[JNTU : April-11, Marks 8]**

Ans. : Let the production rule for the CFG is

$$A \rightarrow (A)A$$

as there is no terminal symbol generation by this grammar, we can not derive any string from it. Hence we will assume the rule to be

$$A \rightarrow (A)A$$

$$A \rightarrow ( ) A$$

$$A \rightarrow ( ) | \varepsilon$$

The strings that will get generated will be $\{( ), ( )( ), (( )), (( )( )), \ldots\}$

Hence the language denoted by this grammar is for well formedness of parenthesis.

2.4 Parsing Techniques

Q.10 Explain various parsing techniques used in compiler. **[JNTU : Part B, Marks 5]**

Ans. : When the parse tree can be constructed from root and expanded to leaves then such type of parser is called Top-down parser. The name itself tells us that the parse tree can be built from top to bottom.

When the parse tree can be constructed from leaves to root, then such type of parser is called as bottom-up parser. Thus the parse tree is built in bottom up manner.

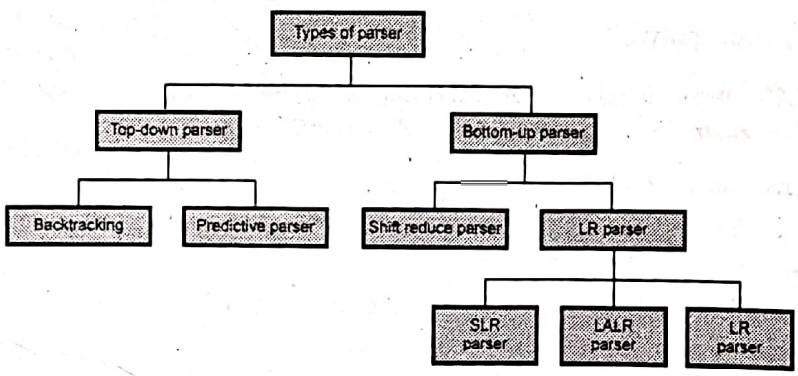


Fig. Q.10.1 Parsing techniques

Q.11 What is the difference between top down parser and bottom up parser ?
☞[JNTU : Part B, March-16, Marks 5]

Ans. :

Sr. No.	Top down parser	Bottom up parser
1.	Parse tree can be built from root to leaves.	Parse tree is built from leaves to toot.
2.	This is simple to implement.	This complex to to implement.
3.	This is less efficient parsing techniques. Various problems that occur during top down technique are ambiguity left recursion.	When the bottom up parser handles ambiguous grammar conflicts occur in parse table.
4.	It is applicable to small class of languages.	It is applicable to a broad class of languages.
5.	Various parsing techniques are 1) Recursive descent parser 2) Predictive parser.	Various parsing techniques are 1) Shift reduce 2) Operator precedence 3) LR parser.

2.5 Top Down Parsing

Q.12 What are problems in top down parsing ?

Ans. : There are certain problems in top-down parsing. In order to implement the parsing we need to eliminate these problems. These problems are -

(1) Backtracking

(2) Left Factoring

(3) Left Recursion

(4) Ambiguity.

Q.13 Write a rule of left factoring a grammar and give example. **☞[JNTU : Part B, Marks 5]**

Ans. : In general if

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2$$

is a production then it is not possible for us to take a decision whether to choose first rule or second. In such a situation the above grammar can be left factored as,

$$A \rightarrow \alpha A'$$

$$A \rightarrow \beta_1 | \beta_2$$

For example : Consider the following grammar.

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

The left factored grammar becomes,

$$S \rightarrow iEtSS' | a$$

$$S' \rightarrow eS | \varepsilon$$

$$E \rightarrow b$$

Q.14 Define left recursive grammar.
☞[JNTU : Part A, Dec.-17, Marks 2]

Ans. : Left Recursion : The left recursive grammar is a grammar which is as given below.

$$A \overset{+}{\Rightarrow} A\alpha$$

Here $\overset{+}{\Rightarrow}$ means deriving the input in one or more steps.

To eliminate left recursion we need to modify the grammar. Let, G be a context free grammar having a production rule with left recursion.

$$\left.\begin{array}{l} A \rightarrow A\alpha \\ A \rightarrow \beta \end{array}\right\} \quad \ldots(1)$$

Then we eliminate left recursion by re-writing the production rule as :

$$\left.\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \\ A' \rightarrow \varepsilon \end{array}\right\} \quad \ldots(2)$$

Q.15 Eliminate left recursion for the following grammar
E-> E+T | T
T ->T*F | F
F-> (E) | id **☞[JNTU : Part A, Nov.-15, Marks 3]**

Ans. : $E \rightarrow E + T \mid T$

We can map this grammar with the rule $A \rightarrow A\alpha | \beta$. Then using equation (2) (in Q.14) we can say,

$$A = E$$

$$\alpha = + T$$

$$\beta = T$$

Then the rule becomes,

$$E \rightarrow TE'$$

$$E' \rightarrow + TE' \mid \varepsilon$$

Similarly for the rule,

$$T \rightarrow T * F \mid F$$

We can eliminate left recursion as

$$T \rightarrow FT'$$

$$T' \rightarrow * FT' \mid \varepsilon$$

The grammar for arithmetic expression can be equivalently written as -

$$E \rightarrow E + T \mid T \qquad E \rightarrow TE'$$

$$E' \rightarrow + TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T \rightarrow T * F \mid F \Rightarrow T' \rightarrow * FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid id \Rightarrow F \rightarrow (E) \mid id$$

Q.16 Discuss in brief about left recursion and left factoring with examples.

[JNTU : Part B, Nov.-17, Marks 5]

Ans. : Left Recursion : Refer Q.14.

Left Factoring : Refer Q.13.

Q.17 Define : Left recursive state the rule to remove left recursive from the grammar. Eliminate left recursive from following grammar.
S → A a | b
A → Ac | Sd | f **[JNTU : Part B, Marks 5]**

Ans. : The left recursive grammar is a grammar which is as given below :

$$A \overset{+}{\Rightarrow} A\alpha$$

To eliminate left recursion we apply following rule.

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$ is a rule then

$$A \rightarrow \beta_1 A' \mid \beta_2 A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \varepsilon$$

If we replace S by Aa|b then there exists only one rule

$$A \rightarrow Ac \mid Aad \mid bd \mid f$$

After eliminating left recursion the grammar becomes

$$A \rightarrow bdA' \mid fA'$$

$$A \rightarrow cA' \mid adA' \mid \varepsilon$$

To summerize

$$S \rightarrow Aa \mid b$$

$$A \rightarrow bdA' \mid fA'$$

$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$

Q.18 Check the following grammar is left recursive or not. Justify your answer. If left recursive then make grammar as non - left recursive.
S → (L) | a
L → L, S | S **[JNTU : Part B, Marks 5]**

Ans. : The following rule is left recursive

$$L \rightarrow L, S \mid S$$

We will use following formula -

If $A \rightarrow A\alpha \mid \beta$ then $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon$. Hence we can eliminate the left recursive rule using above formula

$$\begin{array}{ccccc} L & \rightarrow & L, & S & \mid S \\ \uparrow & & \uparrow & \uparrow & \uparrow \\ A & & A & \alpha & \beta \end{array} \Rightarrow \begin{array}{l} L \rightarrow SL' \\ L' \rightarrow ,SL' \mid \varepsilon \end{array}$$

Q.19 What is backtracking ? Explain it with suitable example. **[JNTU : Part B, Marks 5]**

Ans. : Backtracking is a technique in which for expansion of non-terminal symbol we choose one alternative and if some mismatch occurs then we try another alternative if any.

For example :

$$S \rightarrow xPz$$

$$P \rightarrow yw \mid y$$

Then

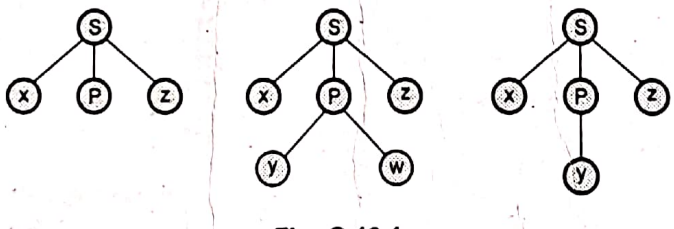


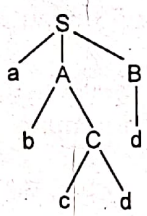
Fig. Q.19.1

If for a non-terminal there are multiple production rules beginning with the same input symbol then to get the correct derivation we need to try all these alternatives. Secondly, in backtracking we need to move some levels upward in order to check the possibilities. This increases lot of overhead in implementation of parsing. And hence it becomes necessary to eliminate the backtracking by modifying the grammar.

Q.20 Define ambiguous grammar. Check whether the grammar S->aAB, A->bC|cd, C->cd, B->c/d is ambiguous or not. ☞**[JNTU : Part B, March-16, Marks 5]**

Ans. : Definition : A grammar G is said to be ambiguous if it generates more than one parse trees for sentence of language L(G).

Consider the string **abcdd**. It can be derived as -



As there is only one possible parse tree that can be generated for deriving the given string, the given grammer is not ambiguous.

Q.21 Consider the following grammar,
E → E + E | E * E | (E) | id
Show that it is ambiguous. Eliminate the ambuigity from this grammar.
☞**[JNTU : May-08, 09, Aug.-08, April-09, Marks 8, Jan.-14, Marks 7]**

Ans. : We will derive the string **id + id * id.**

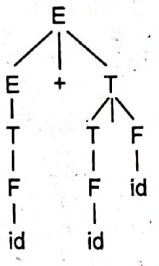
As two different parse trees can be drawn for deriving the same string. Hence given grammar is ambiguous. To eliminate ambiguity, observe that given grammar has left associative operators. Hence we will make use of left recursive production rules to eliminate ambiguity. The unambiguous grammar will be,

E → E + T | T

T → T * F | F

F → (E) | id

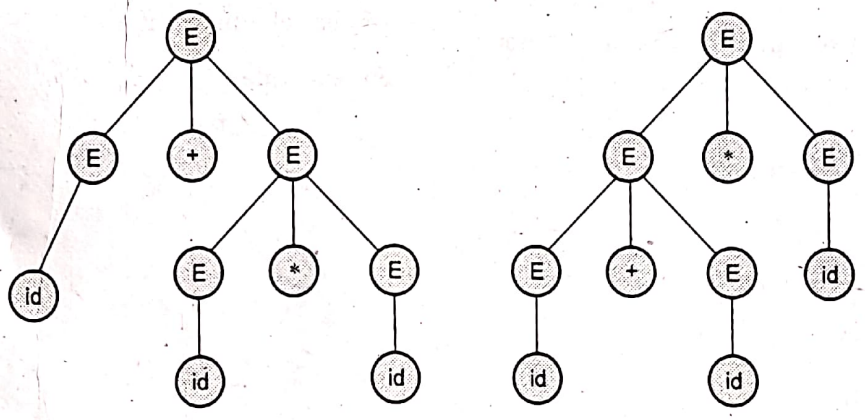
The parse tree for id + id * id will be,



Q.22 Implement the following grammar using recursive descent parser.
S-> Aa | bAc| bBa, A-> d,B -> d
☞**[JNTU : Part B, Marks 5]**

Ans. :

```
Procedure S( )
        {
            if (lookahead = 'b')
             {
               B( ) ;  // or A( ) because both gives 'd'
               if (lookahead  = 'c')
                  match ('c')
               else if (lookahead = 'a')
                  match ('a')
```



Parse tree - 1 Parse tree - 2

Fig. Q.21.1 Parse Trees

```
        else if (loohahead = '$')
        {
            declare SUCCESS ;
        }
    else
    {
        A( ) ;
            if (lookahead = 'a')
                match ('a') ;
    }
}
Procedure A( )
{
    if (lookahead = 'd')
        match ('d')
    else
        error ( );
}
Procedure B( )
{
    if (lookahead = 'd')
        match ('d')
    else
        error( );
}
```

Advantages of Recursive descent parser

1. Recursive descent parsers are simple to build.
2. Recursive descent parsers can be constructed with the help of parse tree.

Limitations of Recursive descent parser

1. Recursive descent parsers are not very efficient as compared to other parsing techniques.
2. There are chances that the program for RD parser may enter in an infinite loop for some input.
3. Recursive descent parser can not provide good error messaging
4. It is difficult to parse the string if lookahead symbol is arbitrarily long.

Q.23 Explain non-recursive predictive parsers. Draw the block diagram of it.

[JNTU : Part B, Marks 5]

Ans. : • The predictive LL(1) top-down parsing algorithm is of non-recursive type.

• In this type of parsing a **table is built.**

• For LL(1) - the **first L** means the input is scanned from left to right. The **second L** means it uses **leftmost derivation** for input string. And the **number 1** in the input symbol means it uses only one **input symbol (lookahead)** to predict the parsing process.

• The simple block diagram for LL(1) parser is as shown in Fig. Q.23.1.

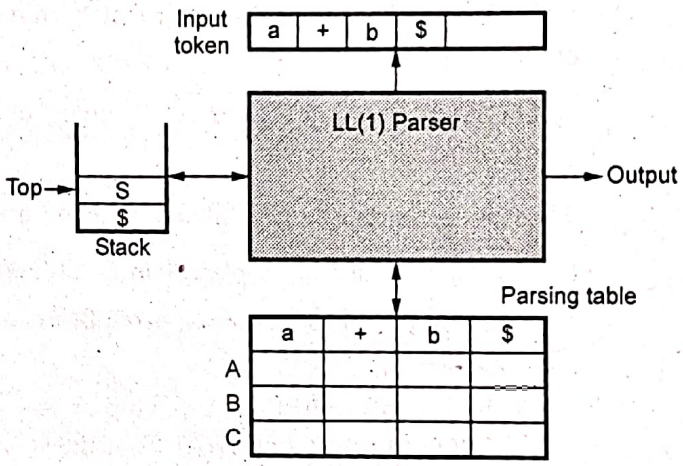


Fig. Q.23.1 Model for LL(1) parser

• The data structures used by LL(1) are **i) input buffer ii) stack iii) parsing table.**

• The LL(1) parser uses **input buffer** to store the input tokens. The **stack** is used to hold the left sentential form.The symbols in R.H.S. of rule are pushed into the stack **in reverse order** i.e. from right to left. Thus use of stack makes this algorithm non-recursive. The **table** is basically a two dimensional array. The table has row for non-terminal and column for terminals. The table can be represented as M[A,a] where A is a non-terminal and a is current input symbol.

Q.24 List out the rules for FIRST and FOLLOW.

[JNTU : Part A, Dec.-17, Marks 3]

OR List the rules for computing FOLLOW set.

[JNTU : Part A, Nov.-15, Marks 2]

Ans. : FIRST function

FIRST(α) is a set of terminal symbols that are first symbols appearing at R.H.S. in derivation of α. If $\alpha \Rightarrow \varepsilon$ then ε is also in FIRST (α).

Following are the rules used to compute the FIRST functions.

1. If the terminal symbol a the FIRST(a) = {a}.
2. If there is a rule $X \rightarrow \varepsilon$ then FIRST(X)={ε}.
3. For the rule $A \rightarrow X_1 X_2 X_3 \ldots X_k$ FIRST(A) = (FIRST(X_1) ∪ FIRST(X_2) ∪ FIRST(X_3)...∪ FIRST(X_k). Where k $X_j \leq n$ such that $1 \leq j \leq k-1$.

FOLLOW function

FOLLOW (A) is defined as the set of terminal symbols that appear immediately to the right of A. In other words

FOLLOW(A) = { a | $S \overset{*}{\Rightarrow} \alpha$ Aa β where α and β are some grammar symbols may be terminal or non-terminal.

The rules for computing FOLLOW function are as given below -

1. For the start symbol S place $ in FOLLOW(S).
2. If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) without ε is to be placed in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B \beta$ or $A \rightarrow \alpha B$ and FIRST(β)= {ε} then FOLLOW(A) = FOLLOW(B) or FOLLOW(B)=FOLLOW(A). That means everything in FOLLOW(A) is in FOLLOW(B).

Q.25 Construct FIRST and FOLLOW for the grammar.
E-> E+T | T
T -> T *F | F
F-> (E) |id

☞ **[JNTU : Part B, Nov.-17, Marks 5]**

Ans. :

FIRST(E) = {(,id}

FIRST(T) = {(,id}

FIRST(F) = {(,id}

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

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

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

Q.26 Construct predictive parsing table for grammar.
E->E+T|T
T -> T *F | F
F-> (E) |id

☞ **[JNTU : Part B, May-16, Marks 5]**

Ans. : The given grammar contains left recursion. We will eliminate left recursion first. Refer Q.15 for elimination of left recursion from given grammar. On eliminating left recursion we get -

$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) | id$

First create the table as follows.

	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

Now we will fill up the entries in the table using the above given algorithm. For that consider each rule one by one.

$E \rightarrow TE'$

$A \rightarrow \alpha$

$A = E, \alpha = TE'$

FIRST(TE') if $E' = \varepsilon$ then FIRST(T) = {(,id}

M[E, (] = E→TE'

M[E, id] = E→TE'

$E' \rightarrow +TE'$

$A \rightarrow \alpha$

$A = E', \alpha = +TE'$

FIRST(+TE') = {+ }

Hence M[E,+] = E'→ +TE'

$E' \rightarrow \varepsilon$

$A \rightarrow \alpha$

$A = E', \alpha = \varepsilon$ then

FOLLOW(E') = {),$}

M[E',)] = $E' \rightarrow \varepsilon$

M[E, $] = $E' \rightarrow \varepsilon$

$T \rightarrow FT'$

$A \rightarrow \alpha$

$A = E', \alpha = FT'$

FIRST(FT') = FIRST(F) = {(,id}

Hence M[F,(] = $T \rightarrow FT'$

And M[F,id] = $T \rightarrow FT'$

$T \rightarrow *FT'$

$A \rightarrow \alpha$

$A = T, \alpha = *FT'$

FIRST(*FT') = {*}

Hence M[T,*] = $T \rightarrow *FT'$

$T' \rightarrow \varepsilon$

$A \rightarrow \alpha$

$A = T', \alpha = \varepsilon$

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

Hence M[T',+] = $T' \rightarrow \varepsilon$

M[T',)] = $T' \rightarrow \varepsilon$

M[T',$] = $T' \rightarrow \varepsilon$

$F \rightarrow (E)$

$A \rightarrow \alpha$

$A = F, \alpha = (E)$

FIRST((E)) = { (}

Hence M[F,(]= F→(E) .

F → id

A → α

A = F, α = id

FIRST(id) = { id }

Hence M[F,id] = F → id

The complete table can be as shown below.

	id	+	*	(	)	$
E	E → TE′	Error	Error	E → TE′	Error	Error
E′	Error	E → +TE′	Error	Error	E′→ ε	E′→ ε
T	E → FT′	Error	Error	T′ → FT′	Error	Error
T′	Error	T′ → ε	T′ → *FT′	Error	T′ → ε	T′ → ε
F	F → id	Error	Error	F → (E)	Error	Error

Now the input string **id + id * id $** can be parsed using above table. At the initial configuration the stack will contain start symbol E, in the input buffer input string is placed.

Stack	Input	Action
$E	id + id * id $	

Now symbol E is at top of the stack and input pointer is at first id, hence M[E,id] is referred. This entry tells us E → TE', so we will push E′ first then T.

Stack	Input	Action
$E′ T	id + id * id $	E → TE′
$E′ T′ F	id + id * id $	E → FT′
$E′ T′ id	id + id * id $	F → id
$E′ T′	+ id * id $	
$E′	+ id * id $	T′ → ε
$E′ T+	+ id * id $	E′ → +TE′
$E′ T	id * id $	
$E′ T′ F	id * id $	T → FT′
$E′ T′ id	id * id $	F → id
$E′ T′	* id $	
$E′ T′ F*	* id $	T → *FT′
$E′ T′ F	id $	
$E′ T′ id	id $	F → id
$E′ T′	$	
$E′	$	T → ε
$	$	E′ → ε

Q.27 Construct predictive parsing table for the following grammar :
S->(L) | a
L->L,S | S
Check whether the (a,a) belong to that grammar or not.
☞[JNTU : Part B, Nov.-15, March-17, May-18, Marks 10]

Ans. : As the given grammar is left recursive because of

$$L \rightarrow L, S | S.$$

We will first eliminate left recursion. As $A \rightarrow A\ \alpha|\beta$ can be converted as

$A \rightarrow \beta\ A'$

$A' \rightarrow \alpha\ A' | \varepsilon$

We can write $L \rightarrow L, S | S$ as

$L \rightarrow SL'$

$L' \rightarrow ,SL' | \varepsilon$

Now the grammar taken for predictive parsing is -

$S \rightarrow (L) | a$

$L \rightarrow SL'$

$L' \rightarrow , SL' | \varepsilon$

Now we will compute FIRST and FOLLOW of non-terminals

$\therefore$ FIRST(S) = {(, a}

FIRST(L) = {(, a}

FIRST(L') = {', ε}

FOLLOW(S) = {',), $}

FOLLOW(L) = FOLLOW(L') = {)}

The predictive parsing table can be constructed as

	a	(	)	'	$
S	S → a	S → (L)			
L	L → SL'	L → SL'			
L'			L' → ε	L' → , SL'	

We will parse the string (a, a) using that table as shown below.

State	Input	Actions
$ S	(a, a) $	
$) L (	(a, a) $	S → (L)
$) L	a, a) $	L → SL'
$) L' S	a, a) $	S → a
$) L' a	a, a) $	
$) L'	, a) $	L' → , SL'
$) L' S,	, a) $	
$) L' S	a) $	S → a
$) L' a	a) $	
$) L'	) $	L' → ε
$)	) $	
$	$	Accept

Q.28 Show that following grammar :
S → AaAb|BbBa
A → ε
B → ε
is LL (1). **☞[JNTU : Part B, May-13, Marks 8]**

Ans. : Consider the grammar :

$S \rightarrow AaAb$

$S \rightarrow BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Now we will compute FIRST and FOLLOW functions.

FIRST(S) = {a, b} if we put

$S \rightarrow AaAb$

$S \rightarrow aAb$ When $A \rightarrow \varepsilon$

Also $S \rightarrow BbBa$

$S \rightarrow bBa$ When $B \rightarrow \varepsilon$

FIRST(A) = FIRST(B) = {ε}

FOLLOW(S) = {$}

FOLLOW(A) = FOLLOW(B) = {a, b}

The LL(1) parsing table is

	a	b	$
S	S → AaAb	S → BbBa	
A	A → ε	A → ε	
B	B → ε	B → ε	

Now consider the string "ba". For parsing -

Stack	Input	Action
$S	ba$	S → BbBa
$aBbB	ba$	B → ε
$aBb	ba$	
$aB	a$	B → ε
$a	a$	
$	$	Accept

This shows that the given grammar is LL(1)

Q.29 Consider the grammar
S → iEtSS'| a
S' → eS|ε
E → b
Whether it is LL(1) grammar. Give the explanation whether (i, t, e, b, a) are terminal symbols.

☞[JNTU : Part B, Dec.-11, Marks 16]

Ans. : Let the given grammar will be,

$S \rightarrow iCtSS'|a$

$S' \rightarrow eS|\varepsilon$

$E \rightarrow b$

Now we will compute FIRST and FOLLOW for given nonterminals.

FIRST(S) = {i, a} FOLLOW(S) = {e,$}

FIRST(S') = {e, ε} FOLLOW(S') = {e,$}

FIRST(E) = {b} FOLLOW(E) = {t}

The predictive parsing table will be

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

As we have got multiple entries in M[A, e] given grammar is **not LL(1) grammar.**

The (i, t, e, a, b) are the terminal symbols because they do not derive any production rule.

Q.30 For the following grammar.
D → TL;
L → L, id | id
T → int | float
1) Remove left recursion (if required)
2) Find first and follow for each non-terminal for resultant grammar
3) Construct LL (1) Parsing table
4) Parse the following String (Show Stack actions Clearly) and drawe parse Parse tree for the input : int id, id; **☞[JNTU : Part B, Marks 5]**

Ans. : 1) Formula for removing the left recursion is as follows :

if $A \rightarrow A\alpha | \beta$ is the rule. Then it can be converted to

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' | \varepsilon$

D → TL ;	No action as the rule is not left recursive.
L → L, id | id	L → id L' L'→, id L' L'→ ε
T → int | Float	No action as the rule rule has no left recursion.

2) The first and follow functions :

FIRST (D) = {int, float}	FOLLOW (D) = {$}
FIRST (L) = {id}	FOLLOW (L) = {;}
FIRST (L′) = {, , ε}	FOLLOW (L′) = {;}
FIRST (T) = {int, float}	FOLLOW (T) = {id}

3) Construction of Parsing table :

	id	,	int	float	;	$
D			D → TL;	D → TL;		
L	L → idL′					
L′		L′ → , idL′			L′ → ε	
T			T → int	T → float		

4) Parsing of string int id, id;

Stack	Input	Action
$D	int id,id;$	D → TL
$;LT	int id,id;$	T→ int
$;Lint	int id,id;$	
$;L	id,id;$	L→ id L′
$;L′id	id,id;$	
$;L′	, id;$	L′→ , id L′
$;L′id,	, id;$	
$;L′id	id;$	
$;L′	; $	L′→ ε
$	; $	
$	$	ACCEPT

Q.31 Verify the following grammar is LL(1) :
S → S #
S → A|B
A → a|ε
B → b|ε
Give a trace of the parser for each of the following input strings
a) aab #
b) ccd #

☞ **[JNTU : Part B, May-12, Marks 15]**

Ans. : Let

$S \rightarrow S\ \#\ |A|B$

$A \rightarrow a|\varepsilon$

$B \rightarrow b|\varepsilon$

be the given grammar. We will eliminate left recursion.

$A \rightarrow A\ \alpha|B_1|B_2, \ldots \Rightarrow A \rightarrow \beta_1\ A'|\beta_2\ A'| \ldots$

$$A \rightarrow \alpha A' \mid \varepsilon$$

$\therefore$ $S \rightarrow S\,\#\mid A \mid B$ becomes $S \Rightarrow AS' \mid BS'$

$$S' \rightarrow \#\, S' \mid \varepsilon$$

Hence we get,

$S \rightarrow AS' \mid BS'$

$S' \rightarrow \#\, S' \mid \varepsilon$

$A \rightarrow a \mid \varepsilon$

$B \rightarrow b \mid \varepsilon$

We will now compute FIRST and FOLLOW for each non terminal symbol.

FIRST (S) = {a, b, #} FOLLOW (S) = {$}

FIRST (S') = {#, ε} FOLLOW (S') = {$}

FIRST (A) = {a, ε} FOLLOW (A) = {#, $}

FIRST (B) = {b, ε} FOLLOW (B) = {#, $}

The parsing table will be

	a	b	#	$
S	$S \rightarrow A\,S'$	$S \rightarrow B\,S'$		
S'			$S' \rightarrow \#\, S'$	$S' \rightarrow \varepsilon$
A	$A \rightarrow a$			$A \rightarrow \varepsilon$
B		$B \rightarrow b$		$B \rightarrow \varepsilon$

a) Simulation of aab #

State	Input	Actions
$ S	aab # $	$S \rightarrow A\,S'$
$ S'A	aab #	$A \rightarrow a$
$ S'a	aab #	
$ S'	ab #	No action defined

Hence aab # does not belong to this grammar.

b) As c, d are not generated by any of the non terminal symbols. This string can not be derived from given grammar.

Q.32 Construct LL (1) parse table for the following grammar.

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$ ☞ **[JNTU : Part B : Dec.-12, Marks 8]**

Ans. : The FIRST and FOLLOWS can be as follows

Non-terminal	FIRST	FOLLOWS
S	{b, d}	{$}
A	{d}	{a, c}
B	{d}	{a, c}

The predictive parsing table will be,

	a	b	c	d	$
S		$S \rightarrow b\,A\,c$ $S \rightarrow b\,B\,a$		$S \Rightarrow A\,a$ $S \rightarrow B\,c$	
A				$A \rightarrow d$	
B				$B \rightarrow d$	

2.6 Bottom Up Parsing

Q.33 What are the actions performed by shift reduce parser ? ☞ **[JNTU : Part A, March-16, Marks 2]**

Ans. : Shift reduce parser attempts to construct parse tree from leaves to root. Thus it works on the same principle of **bottom up parser**. A shift reduce parser requires following data structures.

1. The **input buffer** storing the input string.
2. A **stack** for storing and accessing the L.H.S. and R.H.S. of rules.

The initial configuration of Shift reduce parser is as shown in Fig. Q.33.1

The parser performs following basic **operations.**

Fig. Q.33.1 Initial configuration

1. **Shift :** Moving of the symbols from input buffe onto the stack, this action is called shift.
2. **Reduce :** If the handle appears on the top of th stack then reduction of it by appropriate rule i done. That means R.H.S. of rule is popped o and L.H.S. is pushed in. This action is calle Reduce action.
3. **Accept :** If the stack contains start symbol onl and input buffer is empty at the same time the that action is called accept. When accept state obtained in the process of parsing then it mean a successful parsing is done.

4. **Error :** A situation in which parser cannot either shift or reduce the symbols, it cannot even perform the accept action is called as error.

Q.34 Consider the grammar
E → E – E
E → E * E
E → id
Perform Shift-Reduce parsing of the input string "id1–id2*id3". ☞ **[JNTU : Part B, Marks 5]**

Ans. :

Stack	Input buffer	Parsing action
$	id1–id2*id3$	Shift
$id1	–id2*id3$	Reduce by E→ id
$E	–id2*id3$	Shift
$E–	id2*id3$	Shift
$E–id2	*id3$	Reduce by E→ id
$E–E	*id3$	Shift
$E–E*	id3$	Shift
$E–E*id3	$	Reduce by E→ id
$E–E*E	$	Reduce by E→ E*E
$E–E	$	Reduce by E→ E–E
$E	$	Accept

Here we have followed two rules.

1. If the **incoming operator** has **more priority** than in stack operator then perform **shift.**
2. If **in stack operator** has **same or less priority** than the priority of incoming operator then perform **reduce.**

Q.35 Define handle and handle pruning. Explain the stack implementation of shift reduce parser with the help of example. ☞ **[JNTU : Part B, Marks 5]**

Ans. : Handle : Handle is a string of substring that matches the right side of the production and we can reduce such a string by non-terminal on left hand side production. Formally it can be defined as -

Defination of Handle : "Handle of right sentential form γ is a production A→β and a position of γ where the string β may be found and replaced by A to produce the previous right sentential form in rightmost derivation of γ".

Handle Pruning : Handle pruning is a process of detecting the handles and using them in reduction.

For example :

Consider the grammar

$$E \rightarrow E+E$$
$$E \rightarrow id$$

Now consider the string **id + id + id** and the rightmost derivation is

$$E \underset{rm}{\Rightarrow} E + E$$
$$E \underset{rm}{\Rightarrow} E + E + E$$
$$E \underset{rm}{\Rightarrow} E + E+ \mathbf{id}$$
$$E \underset{rm}{\Rightarrow} E + \mathbf{id} + id$$
$$E \underset{rm}{\Rightarrow} \mathbf{id} + id + id$$

The bold strings are called handles.

Right sentential form	Handle	Production
id + id + id	id	E → id
E + **id** + id	id	E → id
E + E + **id**	id	E → id
E + E + E	E + E	E → E + E
E + E	E + E	E → E + E
E		

Shift Reduce Parsing - Refer Q.33.

Q.36 What is operator precedence grammar ? Give an example. ☞ **[JNTU : Part A, March 17, Marks 2]**

Ans. : A grammar G is said to be **operator precedence** if it posses following properties -

1. No production on the right side is ε.
2. There should not be any production rule possessing two adjacent non-terminals at the right hand side.

Consider the grammar for arithmetic expressions.

E → EAE | (E) | – E | id

A→ + | – | * | / | ^

This grammar is not an operator precedent grammar as in the production rule.

$E \rightarrow EAE$

It contains two consecutive non-terminals. Hence first we will convert it into equivalent operator precedence grammar by removing A.

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \wedge E$$

$$E \rightarrow (E) \mid - E \mid id$$

In operator precedence parsing we will first define precedence relations $<\cdot$ ≐ and $\cdot>$ between pair of terminals. The meaning of these relations is

$p <\cdot q$	p gives more precedence than q.
$p \doteq q$	p has same precedence as q.
$p \cdot> q$	p takes precedence over q.

These meanings appear similar to the less than, equal to and greater than operators. Now by considering the precedence relation between the arithmetic operators we will construct the operator precedence table. The operators precedences we have considered are id,+,*,$.

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

Fig. Q.36.1 Precedence relation table

Now consider the string.

id + id * id

We will insert $ symbols at the start and end of the input string. We will also insert precedence operator by referring the precedence relation table.

$ <· id ·> + <· id ·> * <· id ·> $

We will follow following steps to parse the given string -

i) Scan the input from left to right until first $\cdot>$ is encountered.

ii) Scan backwards over = until $<\cdot$ is encountered.

iii) The handle is a string between $<\cdot$ and $\cdot>$.

The parsing can be done as follows.

$ <· id ·> + <· id ·> * <· id ·> $	Handle id is obtained between <· ·> Reduce this by E → id
E + <· id ·> * <· id ·> $	Handle id is obtained between <· ·> Reduce this by E → id
E + E * <· id ·> $	Handle id is obtained between <· ·> Reduce this by E → id

E + E * E	Remove all the non-terminals.
+ *	Insert $ at the beginning at the end. Also insert the precedence operators.
$ <· + <· * ·> $	The * operator is surrounded by <· ·> .This indicates that * becomes handle.That means we have to reduce E*E operation first.
$ <· + ·> $	Now + becomes handle. Hence we evaluate E + E.
$ $	Parsing is done.

2.7 Introduction to LR Parsing

Q.37 What does the term LR stand for ? ☞ [JNTU : Part A, Marks 2]

Ans. : • L stands for Left to Right scanning of input stream.

- The R stands for construction of rightmost derivation in reverse.
- The LR parsing is an efficient bottom-up parsing technique.

Q.38 List the properties of LR Parser. ☞ [JNTU : Part A, May-18, Marks 2]

Ans. : LR parser is widely used for following reasons,

1. LR parsers can be constructed to recognize most of the programming languages for which context free grammar can be written.
2. The class of grammar that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsers.
3. LR parser works using non backtracking shift reduce technique yet it is efficient one.
4. LR Parsers detect syntactical errors very efficiently.

Q.39 What is significance of lookahead operator in LR parsing ? ☞ [JNTU : Part A, March-17, Marks 3]

Ans. : • Lookahead is an extra token read from the current character user is reading. This helps to decide which grammar rule is applicable for deriving the string.

- It increases the efficiency of parsing.
- It avoids the problem of backtracking.

Q.40 Describe in brief about types of LR parsers. ☞ [JNTU : Part A, March-16, Marks 3]

Ans. : The LR parsers are -

1) Simple LR Parser (SLR)
2) Canonical LR Parser (CLR)
3) Lookahead LR Parser (LALR)

Q.41 Explain the terms -
(1) LR(0) Items (2) Augmented Grammar (3) Kernel Items (4) Functions for canonical set of items (5) Viable Prefix.

Ans. : 1) The LR(0) item for grammar G is production rule in which symbol • is inserted at some position in R.H.S. of the rule. For example

$S \rightarrow \bullet ABC$

$S \rightarrow A \bullet BC$

$S \rightarrow AB \bullet C$

$S \rightarrow ABC \bullet$

The production $S \rightarrow \varepsilon$ generates only one item $S \rightarrow \bullet$.

2) **Augmented grammar** : If a grammar G is having start symbol S then augmented grammar is a new grammar G′ in which S′ is a new start symbol such that $S' \rightarrow S$. The purpose of this grammar is to indicate the acceptance of input. That is when parser is about to reduce $S' \rightarrow S$ it reaches to acceptance state.

3) **Kernel items** : It is collection of items $S' \rightarrow \bullet S$ and all the items whose dots are not at the leftmost end of R.H.S. of the rule.

Non-Kernel items : The collection of all the items in which • are at the left end of R.H.S. of the rule.

4) **Functions closure and goto** : These are two important functions required to create collection of canonical set of items.

5) **Viable prefix** : It is the set of prefixes in the right sentential form of production $A \rightarrow \alpha$. This set can appear on the stack during shift/reduce action.

Q.42 Define CLOSURE(I).

☞ [JNTU : Part A, Nov.-15, Marks 3]

Ans. : For a context free grammar G, if I is the set of items then the function closure(I) can be constructed using following rules.

1. Consider I is a set of canonical items and initially every item I is added to closure(I).
2. If rule $A \rightarrow \alpha \bullet B\beta$ is a rule in closure(I) and there is another rule for B such as $B \rightarrow \gamma$ then

closure(I) : $A \rightarrow \alpha \bullet B\beta$

$B \rightarrow \bullet\gamma$

This rule has to be applied until no more new items can be added to closure(I).

The meaning of rule $A \rightarrow \alpha \bullet B\beta$ is that during derivation of the input string at some point we may require strings derivable from $B\beta$ as input. A non-terminal immediately to the right of • indicates that it has to be expanded shortly.

Q.43 Explain GOTO function in LR parsing.

Ans. : The function goto can be defines as follows.

If there is a production $A \rightarrow \alpha \bullet B\beta$ then goto($A \rightarrow \alpha \bullet B\beta$,B) = $A \rightarrow \alpha B \bullet \beta$. That means simply shifting of • one position ahead over the grammar symbol (may be terminal or non-terminal). The rule $A \rightarrow \alpha \bullet B\beta$ is in I then the same goto function can be written as goto(I,B).

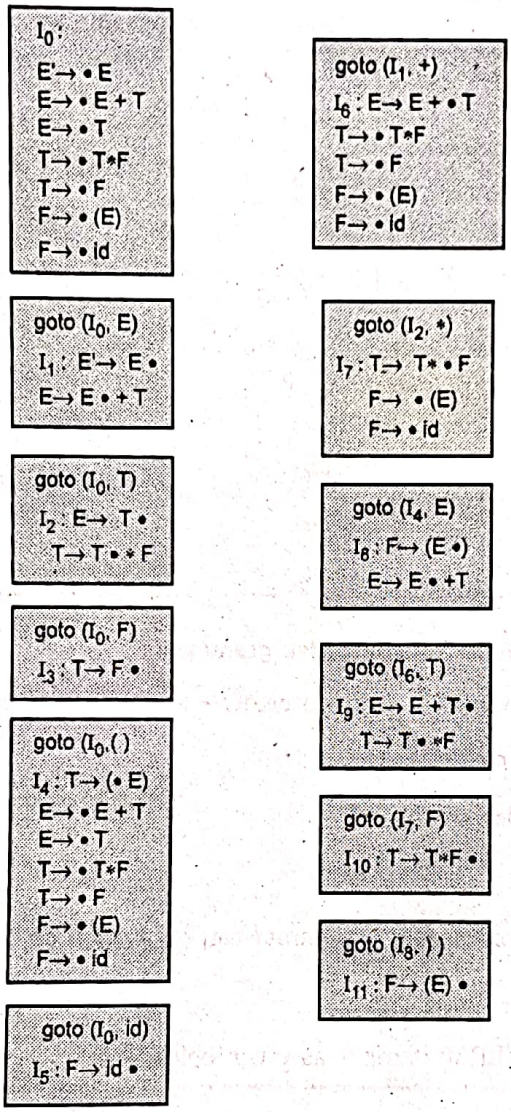
Q.44 Construct SLR parsing table for the following grammar

E-> E+T|T T->T*F|F F->(E)|id?

☞ [JNTU : Part B, March-16, Nov.-17, Marks 5, Nov.-16, Marks 10]

Ans. : We will first construct a collection of canonical set of items for the above grammar. The set of items generated by this method are also called **SLR(0) items**. As there is no lookahead symbol in this set of items, zero is put in the bracket.

State	Action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				Accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			



Consider the rules with numbering as :

1) E → E+T

2) E → T

3) T → T * F

4) T → F

5) F → (E)

6) F → id

Input string : id * id+id

We will consider two data structures while taking the parsing actions and those are - Stack and input buffer.

Stack	Input buffer	Action table	goto table	Parsing action
$0	id*id+id$	[0,id]=s5		Shift
$0id5	*id+id$	[5,*]=r6	[0,F]=3	Reduce by F → id

$0F3	*id+id$	[3,*]=r4	[0,T]=2	Reduce by T → F
$0T2	*id+id$	[2,*]=s7		Shift
$0T2*7	id+id$	[7,id]=s5		Shift
$0T2*7id5	+id$	[5,+]=r6	[7,F]=10	Reduce by F → id
$0 T2*7F10	+id$	[10,+]=r3	[0,T]=2	Reduce by T → T * F
$0T2	+id$	[2,+]=r2	[0,E]=1	Reduce by E → T
$0E1	+id$	[1,+]=s6		Shift
$0E1+6	id$	[6,id]=s5		Shift
$0E1+6id5	$	[5,$]=r6	[6,F]=3	Reduce by F → id
$0E1+6F3	$	[3,$]=r4	[6,T]=9	Reduce by T → F
$0E1+6T9	$	[9,$]=r1	[0,E]=1	E → E +T
$0E1	$	Accept		Accept

Q.45 Construct the LR(0) items for the dangling-else grammar. ☞ **[JNTU : Part A, Dec.-16, Marks 3]**

Ans. : Step 1 : The dangling else grammar is as given below -

stmt-> **if expr then** stmt

| **if expr then** stmt **else** stmt

|other

Step 2 : The simplified version of above grammar can be written as

S->iS|iSeS|o

Step 3 : The canonical set of LR(0) items is as given below -

I_0 : S'->• S S->• iS S->• iSeS S-> •o	I_1 : S'->S•
I_2 : S->i• S S->i• SeS S->• o	I_3 : S->o •
I_4 : S->iS• S->iS • eS	I_5 : S->iSe • S S-> • iS S-> • iSeS S-> • o
I_6 : S->iSeS•	

Q.46 Find the LR(0) set of items for the following grammar. Describe state diagram and construct parse table of that
S->CC
C->cC|d **☞[JNTU : Part B, March-17, Marks 10]**

Ans. : We will construct canonical set of LR(0) items

I_0 : $S' \rightarrow \bullet S$
$S \rightarrow \bullet C C$
$C \rightarrow \bullet c C$
$C \rightarrow \bullet d$

I_1 : goto (I_0, S)
$S' \rightarrow S \bullet$

I_2 : goto (I_0, C)
$S \rightarrow C \bullet C$
$C \rightarrow \bullet c C$
$C \rightarrow \bullet d$

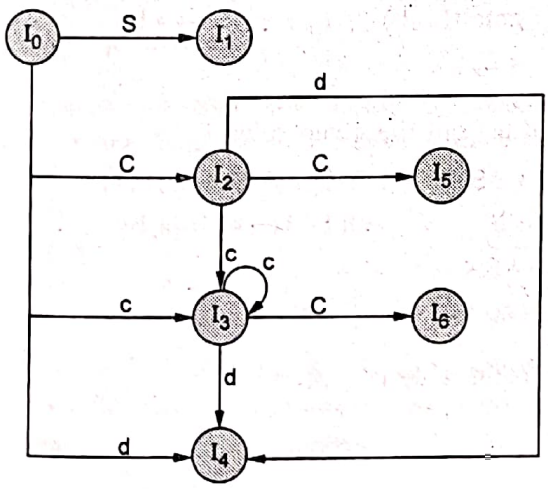
I_3 : goto (I_0, c)
$C \rightarrow c \bullet C$
$C \rightarrow \bullet c C$
$C \rightarrow \bullet d$

I_4 : goto (I_0, d)
$C \rightarrow d \bullet$

I_5 : goto (I_2, C)
$S \rightarrow C C \bullet$

I_6 : goto (I_3, C)
$C \rightarrow c, C \bullet$

The state diagram is,



The parsing table is,

State No.	Action			Goto	
	c	d	$	S	C
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		

Q.47 Construct the collection of LR(0) item sets and draw the goto graph for the following grammar S → S S |a|ε.
Indicate the conflicts (if any) in the various states of the SLR parser. **☞[JNTU : Part B, Marks 5]**

Ans. : We will number the production rules in the grammar.

1) $S \rightarrow SS$
2) $S \rightarrow a$
3) $S \rightarrow \varepsilon$

Now let us construct canonical set of items -

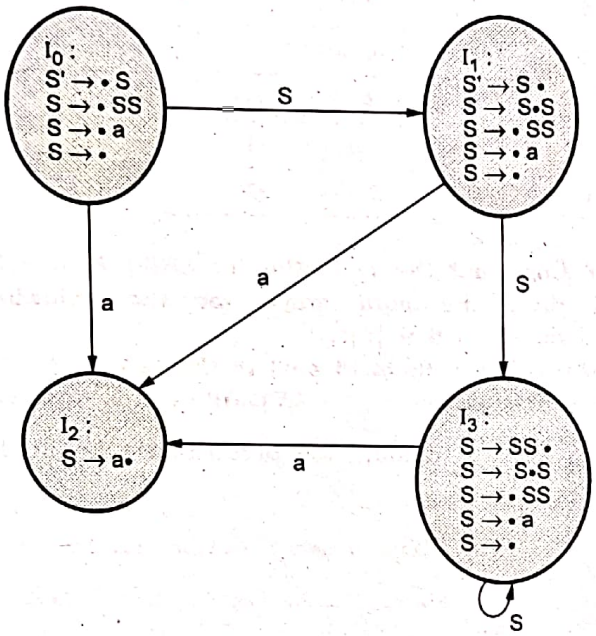
I_0 : $S' \rightarrow \bullet S$ This is augmented grammar.
$S \rightarrow \bullet SS$
$S \rightarrow \bullet a$
$S \rightarrow \bullet$

I_1 : goto (I_0, S)
$S' \rightarrow \bullet S$
$S \rightarrow S \bullet S$
$S \rightarrow \bullet SS$
$S \rightarrow \bullet a$
$S \rightarrow \bullet$

I_2 : goto (I_0, a)
$S \rightarrow a \bullet$

I_3 : goto (I_1, S)
$S \rightarrow SS \bullet$
$S \rightarrow S \bullet S$
$S \rightarrow \bullet SS$
$S \rightarrow \bullet a$
$S \rightarrow \bullet$

The goto graph will be



FOLLOW(S) = {a,$}

The construction of SLR(1) table will be.

In I_3 state there is a production rule S → • a . If we apply goto (I_3, a) then we will get I_2 state. Hence M[3, a] = S2. In I_3 there is a production S → SS • which matches with A→α• Rule S→ SS is rule number 1. And FOLLOW(S) = {a,$}

∴ M[3, a] = M[3, $] = r_1

	Action		Goto
	a	$	S
0	S2		1
1	S2	Accept	3
2	r2	r2	
3	S2 / r1	r1	

Conflict occurs. [shift-reduce conflict]

Hence conflict occurs in state I_3.

Q.48 Construct SLR parsing table for the following grammar

S → AS|b

A → SA|a ☞ **[JNTU : Part B, April-11, Marks 10]**

Ans. : We will construct a collection of canonical set of items for the given grammar.

I_0 : S′ → •S
S → •AS
S → • b
A → • SA
A → •a

I_1 : goto (I_0,S)
S′ → S•
A → S•A
A → • SA
A → • a
S → •AS
S → • b

I_2 : goto (I_0, A)
S → A•S
S → •AS
S → • b
A → • SA
A → • a

I_3 : goto (I_0, b)
S → b •

I_4 : goto (I_0,a)
A → a•

I_5 : goto (I_1,A)
A → SA•
A → A•S
S → • AS
S → • b
A → • SA
A → • a

I_6 : goto (I_1, S)
A → S•A
A → • SA
A → • a
S → • AS
S → • b

I_7 : goto (I_2, S)
S → AS •
A → S•A
A → • SA
A → • a
S → • AS
S → • b

We will list out the given rules

1. S → AS FOLLOW (S) = {a, b, $}
2. S → b FOLLOW (A) = {a, b}
3. A → SA
4. A → a

The parsing table will be

	Action			goto	
	a	b	$	S	A
0	S4	S3		1	2
1	S4	S3	Accept	6	5
2	S4	S3		7	2
3	r2	r2	r2		

4	r4	r4			
5	S4/r3	S3/r3		7	2
6	S4	S3		6	5
7	S4/r1	S3/r1	r1	6	5

Q.49 Construct the collection of non empty sets of LR(0) items for the following augmented grammar.
S → E
E → TE/T ☞ **[JNTU : Part B, Dec.-11, Marks 16]**

Ans. :

I_0 : S → • E
E → • TE
E → • T

I_1 : goto (I_0,E)
S → E •

I_2 : goto (I_0,T)
E → T • E
E → T •
E → • TE
E → • T

I_3 : goto (I_2, E)
E → TE •

Q.50 Check whether the following grammar is SLR(1) or not. Explain your answer with reasons.
S → L = R
S → R
L → * R
L → id
R → L ☞ **[JNTU : Part-B, Marks 5]**

Ans. : We will first build a canonical set of items. Hence a collection of LR(0) items is as given below.

I_0 : S′ → • S
S → • L = R
S → • R
L → • * R
L → • id
R → • L

I_1 : goto (I_0, S)
S′ → S•

I_2 : goto (I_0, L)
S → L • = R
R → L •

I_3 : goto(I_0, R)
S → R •

I_4 : goto(I_0, *)
L → * • R
R → • L
L → • * R
L → • id

I_5 : goto(I_0, id)
L → id •

I_6 : goto(I_2, =)
S → L = • R
R → • L
L → • * R
L → • id

I_7 : goto (I_4, R)
L → * R •

I_8 : goto (I_4, L)
R → L •

I_9 : goto(I_6, R)
S → L = R •

Now we will build the parsing table for the above set of items.

State	Action				Goto		
	=	*	id	$	S	L	R
0							
1							
2	s6/r5						
3							
4		s4	s5				

5							
6		s4	s5				
7							
8							
9							

As we are getting multiple entries in Action [2, =] = s6 and r5. That means shift and reduce both a shift/reduce conflict occurs on input symbol =. Hence given grammar is not SLR(1) grammar.

Q.51 Construct a DFA whose states are the canonical collection of LR(1) items for the following augmented grammar,
$S \rightarrow A$
$A \rightarrow BA \mid \varepsilon$
$B \rightarrow aB \mid b$ ☞ **[JNTU : Part B, Jan.-10, Marks 16]**

Ans. : Initially we will start with the rule

$S \rightarrow \bullet A, \$$

We will add the rules $A \rightarrow \bullet BA$ and $A \rightarrow \bullet$ to I_0.

Now let us map the rule $[A \rightarrow \alpha \bullet X \beta, a]$ with $S \rightarrow \bullet A, \$$.

Such that $A = S, \ \alpha = \varepsilon, \ X = A, \ \beta = \varepsilon, \ a = \$$.

Then second component of $A \rightarrow \bullet BA$ and $A \rightarrow \bullet$

Will be $= FIRST(\beta a)$

$= FIRST(\varepsilon, \$)$

$= FIRST(\$)$

$= (\$)$

$\therefore$ $S \rightarrow \bullet A, \$$

$A \rightarrow \bullet BA, \$$

$A \rightarrow \bullet, \$$

As there is a rule $A \rightarrow \bullet BA$ we must add the rules $B \rightarrow \bullet a\,B$ and $B \rightarrow \bullet b$ to I_0. Now to compute second component of B' S rule we will use

$A \rightarrow \bullet BA, \$$ for mapping with $[A \rightarrow \alpha \bullet X\beta, a]$

$A = A, \ \alpha = \varepsilon, \ X = B, \ \beta = A, \ a = \$$.

Then second component of $B \rightarrow \bullet a\,B$ and $B \rightarrow \bullet b$

will be $= FIRST(\beta a)$

$= FIRST(A, \$)$

$= FIRST(A)$

$= \{a, b, \varepsilon\}$

$= \{a, b\}$

$\therefore$ $B \rightarrow \bullet a\,B, a/b$

$B \rightarrow \bullet b, a/b$.

Hence I_0 will be

I_0 : $S \rightarrow \bullet A, \$$

$A \rightarrow \bullet BA, \$$

$A \rightarrow \bullet, \$$

$B \rightarrow \bullet a\,B, a/b$

$B \rightarrow \bullet b, a/b$

I_1 : goto (I_0, A)

$S \rightarrow A \bullet, \$$

I_2 : goto (I_0, B)

$A \rightarrow B \bullet A, \$$

$A \rightarrow \bullet BA, \$$

$A \rightarrow \bullet, \$$

$B \rightarrow \bullet a\,B, a/b$

$B \rightarrow \bullet b, a/b$

I_3 : goto (I_0, a)

$B \rightarrow a \bullet B, a/b$

$B \rightarrow \bullet a\,B, a/b$

$B \rightarrow \bullet b, \ a/b$

I_4 : goto (I_0, b)

$B \rightarrow b \bullet, \ a/b$

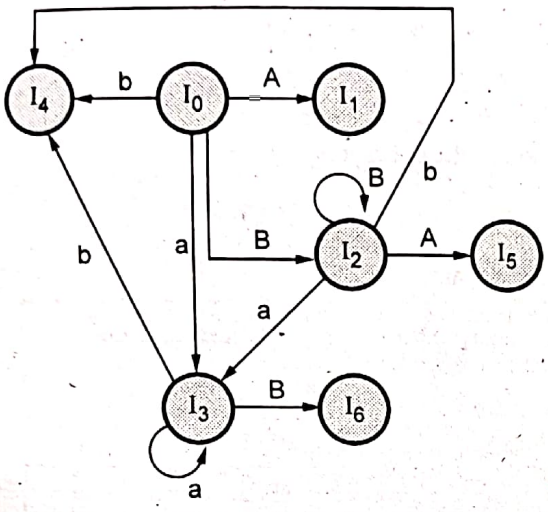
I_5 : goto (I_2, A)

$A \rightarrow BA \bullet, \$$

I_6 : goto (I_2, B)

$B \rightarrow a\,B \bullet, a/b$

The DFA can be constructed as follows -



Q.52 Construct the LR(1) parsing table for the following grammar.

1) S → CC
2) C → cC
3) C → d

☞ **[JNTU : Part B, Marks 5]**

I_0:
S′→ • S, $
S→ • CC, $
C→ • cC, c/d
C→ • d, c/d

I_1: goto (I_0, S)
S′→ S •, $

I_2: goto (I_0, C)
S→ C • C, $
C→ • cC, $
C→ • d,$

I_3: goto (I_0, c)
C→ c • C, c/d
C→ • cC, c/d
C→ • d, c/d

I_4: goto (I_0, d)
C→ d • , c/d

I_5: goto (I_2, C)
S→ CC •, $

I_6: goto (I_2, c)
C′→ c• C, $
C→ • cC, $
C→ • d, $

I_7: goto (I_2, d)
C→ d•, $

I_8: goto (I_3, C)
C→ cC•, c/d

I_9: goto (I_6, C)
C→ cC•, $

Ans. :

First we will construct the set of LR(1) items.

Now consider I_0 in which there is a rule matching with $[A \rightarrow \alpha \bullet c\beta, b]$ as

C→ • cC, c/d and if the goto is applied on a then we get the state I_3. Hence we will create entry action[0, a] = shift 3. Similarly,

In I_0

$C \rightarrow \bullet d \quad c/d$

$A \rightarrow \alpha \bullet a \beta, b$

$A = C, \alpha = \varepsilon, c = d, \beta = \varepsilon, b = c/d$

$goto(I_0, d) = I_4$

hence action[0, d] = shift 4

For state I_4

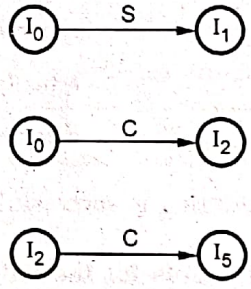
$C \rightarrow d \bullet, c/d$

$A \rightarrow \alpha \bullet, c$

$A = C, \alpha = d, c = c/d$

action[4, c] = reduce by C→ d i.e. rule 3

S′→ S •, $ in I_1



goto (I_0, S) = I_1

goto (I_0, C) = I_2

goto (I_2, C) = I_5

So we will create action[1, $] = accept.

The goto table can be filled by using the goto functions.

For instance $goto(I_0, S) = I_1$. Hence goto[0, S] = 1. Continuing in this fashion we can fill up the LR(1) parsing table as follows.

	Action			goto	
	a	d	$	S	C
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	

The remaining blank entries in the table are considered as syntactical error.

Parsing the input using LR(1) parsing table

Using above parsing table we can parse the input string "aadd" as

Stack	Input buffer	Action table	goto table	Parsing action
$0	aadd$	action[0, a]=s3		
$0a3	add$	action[3,a]=s3		Shift
$0a3a3	dd$	action[3,d]=s4		Shift
$0a3a3d4	d$	action[4,d]=r3	[3,C]=8	Reduce by C → d
$0a3a3C8	d$	action[8,d]=r2	[3,C]=8	Reduce by C → aC
$0a3C8	d$	action[8,d]=r2	[0,C]=2	Reduce by C → aC
$0C2	d$	action[2,d]=s7		Shift
$0C2d7	$	action[7,$]=r3	[2,C]=5	Reduce by C → d
$0C2C5	$	action[5,$]=r1	[0,S]=1	Reduce by S → CC
$0S1	$	accept		

Thus the given input string is successfully parsed using LR parser or canonical LR parser.

Q.53 Construct CLR parsing table for the following grammar
E-> E+T | T, T->T*F |F, F->(E) | id

☞ **[JNTU : Part B, Nov.-17, Marks 5]**

Ans. : We will construct canonical set of LR(1) items.

I_0 : $E' \rightarrow \bullet E, \$$ $E \rightarrow \bullet E + T, \$$ $E \rightarrow \bullet T, \$$ $T \rightarrow \bullet T * F, \$$ $T \rightarrow \bullet F, \$$ $\rightarrow \bullet (E), \$$ $F \rightarrow \bullet id, \$$	I_1 : goto (I_0, E) $E' \rightarrow E \bullet, \$$ $E \rightarrow E \bullet + T, \$$
I_2 : goto (I_0 , T) $E \rightarrow T \bullet, \$$ $T E' \rightarrow T \bullet * F, \$$	I_3 : goto (I_0, F) $T \rightarrow F \bullet, \$$

I_4 : goto (I_0, () F → (• E), $ E → • E + T,) E → • T,) T → • T * F,) T → • F,) F → • (E),) F → • id,)	I_5 : goto (I_0 , d) F → id •, $
I_6 : goto (I_1, +) E → E + • T, $ T → • T * F, $ T → • F, $ F → • (E), $ F → • id, $	I_7 : goto (I_2 , *) T → T * • F, $ F → • (E), $ F → • id, $
I_8 : goto (I_4, E) F → (E •), $ E → E • +T ,)	I_9 : goto (I_6, T) E → E + T • , $ T → T • * F, $
I_{10} : goto (I_7, F) T → T * F •, $	I_{11} : goto (I_8 ,)) E → (E) •, $

Let us number out the original grammer rules as these rule numbers are used for shift and reudce operations in parsing table.

i) E → E + T

2) E → T

3) T → T * F

4) T → F

5) F → (E)

6) F → id

The parsing table can be given below -

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				ACCEPT			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4					
7	s5			s4				9	3

8		s6			s11				10
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Q.54 Construct LALR parser table for the following grammar for CLR parser.
$S \rightarrow L = R$, $S \rightarrow R, L \rightarrow *R$, $R \rightarrow L$. **[JNTU : Part B, May-18, Marks 10]**

Ans. : To derive the given grammar using LALR method we have to build the canonical collection of items using LR(1) items.

The LR(1) items are

I_0 : $S' \rightarrow \bullet S, \$$
$S \rightarrow \bullet L = R \$$
$S \rightarrow \bullet R, \$$
$L \rightarrow \bullet * R, = |\$$
$L \rightarrow \bullet id, =|\$$
$R \rightarrow \bullet L, \$$

I_1 : goto (I_0, S)
$S' \rightarrow S \bullet, \$$

I_2 : goto (I_0, L)
$S \rightarrow L \bullet = R, \$$
$R \rightarrow L \bullet, \$$

I_3 : goto (I_0, R)
$S \rightarrow R \bullet, \$$

I_4 : goto (I_0, *)
$L \rightarrow * \bullet R, = |\$$
$R \rightarrow \bullet L, = |\$$
$L \rightarrow \bullet * R, = |\$$
$L \rightarrow \bullet id, = |\$$

I_5 : goto (I_0, id)
$L \rightarrow id \bullet, = |\$$

I_6 : goto (I_2, =)
$S \rightarrow L = \bullet R \$$
$R \rightarrow \bullet L, \$$
$L \rightarrow \bullet * R, \$$
$L \rightarrow \bullet id, \$$

I_7 : goto (I_4, R)
$L \rightarrow * R \bullet = |\$$

I_8 : goto (I_4, L)
$R \rightarrow L \bullet, = |\$$

I_9 : goto (I_6, R)
$S \rightarrow L = R \bullet, \$$

I_{10} : goto (I_6, L)
$R \rightarrow L \bullet, \$$

I_{11} : goto (I_6 *)
$L \rightarrow * \bullet R, \$$
$R \rightarrow \bullet L, \$$
$L \rightarrow \bullet * R, \$$
$L \rightarrow \bullet id, \$$

I_{12} : goto (I_6, id)
$L \rightarrow id \bullet, \$$

I_{13} : goto (I_{11}, R)
$L \rightarrow * R \bullet, \$$

From above set of items we have got

i) I_4 and I_{11} give same production but lookheads are different. Hence merge them to form I_{411}

ii) $I_5 = I_{12}$ Hence I_{512}

iii) $I_7 = I_{13}$ Hence I_{713}

iv) $I_8 = I_{10}$ Hence I_{810}

Therefore the set of items for LALR are

I_0 : $S' \rightarrow \bullet S, \$$

$S \rightarrow \bullet L = R, \$$

$S \rightarrow \bullet R, \$$

$L \rightarrow \bullet * R, =|\$$

$L \rightarrow \bullet id, =|\$$

$R \rightarrow \bullet L, \$$

I_{713} : $L \rightarrow * R \bullet, =|\$$

I_{810} : $R \rightarrow L \bullet, =|\$$

I_9 : $S \rightarrow L = R \bullet, \$$

I_1 : $S' \rightarrow S \bullet, \$$

I_2 : $S \rightarrow L \bullet, = R, \$$

$R \rightarrow L \bullet, \$$

I_3 : $S \rightarrow R \bullet, \$$

I_{411} : $L \rightarrow * \bullet R, =|\$$

$R \rightarrow L \bullet, =|\$$

$L \rightarrow \bullet * R, =|\$$

$L \rightarrow \bullet Id, =|\$$

I_{512} $L \rightarrow id \bullet =|\$$

I_6 : $S \rightarrow L = \bullet R, \$$

$R \rightarrow \bullet L, \$$

$L \rightarrow \bullet * R, \$$

$L \rightarrow \bullet id, \$$

From above set of items we have got

The parsing table can be constructed as follows -

	Action				goto		
State	id	*	=	$	S	L	R
0	S5	S4			1	2	3
1				acc			
2			S6	r5			
3				r2			
4	S5	S4				8	7
5			r4	r4			
6	S12	S11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	S12	S11				10	13
12				r4			
13				r3			

The LALR parsing table is

	Action				goto		
State	id	*	=	$	S	L	R
0	S512	S411			1	2	3
1				acc			
2			S6	r5			
3				r2			
411	S512	S411				810	713
512			r4	r4			
6	S512	S411				810	9
713			r3	r3			
810			r5	r5			
9				r1			

Q.55 Give CLR parsing table for the grammar S->L=R |R L->*R |id R->L

☞ [JNTU : Part B, March-16, Marks 5]

Ans. : Refer Q.54.

Q.56 Write an algorithm for computing LR(k) items sets. ☞ [JNTU : Part B, Nov.-16, Marks 5]

Ans. :

1. For the grammar G initially add $S' \rightarrow \bullet S$ in the set of item C.

2. For each set of items I_i in C and for each grammar symbol X (may be terminal or non-terminal) add closure (I_i, X). This process should be repeated by applying goto((I_i, X) for each X in I_i such that goto((I_i, X) is not empty and not in C. The set of items has to constructed until no more set of items can be added to C.
3. The closure function can be computed as follows.

   For each item $A \rightarrow \alpha \bullet X \beta$ a and rule $X \rightarrow \gamma$ and $b \in FIRST(\beta\ a)$

   such that $X \rightarrow \bullet\ \gamma$ and b is not in I then add $X \rightarrow \bullet\ \gamma$, b to I.
4. Similarly the goto function can be computed as : For each item
   $[A \rightarrow \alpha \bullet X\beta, a]$ is in I and rule$[A \rightarrow \alpha X \bullet \beta, a]$ is not in goto items then add
   $[A \rightarrow \alpha X \bullet \beta, a]$ to goto items.

Q.57 Write a procedure to construct LALR parsing table. ☞[JNTU : Part B, March-17, Marks 5]

Ans. : The algorithm for construction of LALR parsing table is as given below.

Step 1 : Construct the LR(1) set of items.

Step 2 : Merge the two states I_i and I_j if the first component (i.e. the production rules with dots) are matching and create a new state replacing one of the older state such as
$I_{ij} = I_i \cup I_j$

Step 3 : The parsing actions are based on each item I_i. The actions are as given below.

a) If $[A \rightarrow \alpha \bullet a\ \beta, b]$ is in I_i and goto(I_i, a) = I_j then create an entry in the action table action[I, a] = shift j.

b) If there is a production $[A \rightarrow \alpha \bullet, a]$ in I_i then in the action table action $[I_i, a]$ = reduce by $A \rightarrow \alpha$. Here A should not be S'.

c) If there is a production $S' \rightarrow S \bullet, \$$ in I_i then action[i, $] = accept.

Step 4 : The goto part of the LR table can be filled as : The goto transitions for state i is considered for non-terminals only. If goto(I_i, A) = I_j then goto[I_i, A] = j.

Step 5 : If the parsing action conflict then the algorithm fails to produce LALR parser and grammar is not LALR(1). All the entries not defined by rule 3 and 4 are considered to be "error".

Q.58 Give LALR parsing table for the following grammar
E-> E+T | T T->T*F |F F->(E) | id
☞[JNTU : Part B, Nov.-15, Marks 10]

Ans.: Refer Q.53.

Q.59 Show that the following grammar.
S → Aa| bAc| Bc| bBa
A → d
B → d
is LR (1) but not LALR(1). ☞[JNTU : Part B]

Ans. : We will number out the production rules in given grammar.

1) $S \rightarrow Aa$
2) $S \rightarrow bAc$
3) $S \rightarrow Bc$
4) $S \rightarrow bBa$
5) $A \rightarrow d$
6) $B \rightarrow d$

Now we will first construct canonical set of LR(1) items.

$S' \rightarrow \bullet S, \$$

Initially for the augmented grammar the second component is $. As this rule $[S' \rightarrow \bullet S, \$]$ is matching with $[A \rightarrow \alpha \bullet X\ \beta, a]$ where α is ε, X is S. B is ε and a is $. We will apply closure on S and add all the production rules deriving S. The second component in these production rules will be $. This is because for $[S' \rightarrow \bullet S, \$]$ the β = ε and a = $ ∴ FIRST (β a) = 8, FIRST (ε $) = FIRST ($) = $. Hence we will get

$S' \rightarrow \bullet S, \$$

$S \rightarrow \bullet Aa, \$$

$S \rightarrow \bullet bAc, \$$

$S \rightarrow \bullet Bc, \$$

$S \rightarrow \bullet bBa, \$$

Now we have got one rule [S → •Aa, $] which is matching with [A → a•Xβ, a] and [X → γ]. Hence we will add the closure on A. Hence [A → •d] will be added in this list. Now the second component of A → •d will be decided. As [S → •Aa, $] is matching with [A → A•X β, a] having X as A, β as a and a as $.

Then FIRST (β a) = FIRST (a$) = FIRST (a) = a. Hence rule [A → •d, a] will be added.

Now,

S' → • S, $

S → • Aa, $

S → • bAc, $

S → • Bc, $

S ⇒ • bBa, $

A → • d, a

Now notice the rule S → •Bc, $. It suggest to apply closure on B (As after dot B comes immediately) This rule is matching with [A → α•X β, a] and [X → γ]. Hence we will add rule deriving B. Hence B → •d will be added in the above list. Now

S → •Bc, $ can be mapped with [A → α•X β, a].

Where α = ε

X = B

β = c

a = $

Hence FIRST (β a) = FIRST (c $) = FIRST (c) = c.

Hence rule [B → •d, c] will be added.

Hence finally I_0 will be -

I_0 : S' → • S

S → • Aa, $

S → • bAc, $

S → • Bc, $

S → • bBa, $

A → • d, a

B → • d, c

Continue with applying goto on each symbol.

I_1 : goto (I_0, S)

S' → S•, $

There is no chance of applying closure or goto in this state. Hence it will have only one rule.

I_2 : goto (I_0, A)

S' → A • a, $

Applied goto on A. But as after dot a terminal symbol comes we cannot apply any rule further. The second component is carried as it is.

I_3 : goto (I_0, b)

S → b • Ac, $

S → b•Ba, $

A → •d, c

B → •d, a

After dot A comes hence rule for A → •d will be added. As [S → b•Ac, $] is matching with [A → α•X β, a and X → γ], FIRST (β a) = FIRST (c$) = FIRST (c) = c. The second component of A ⇒ •d is c.

I_4 : goto (I_0, B)

S' → B•c, $

The goto on B is applied and second component is carried as it is.

I_5 : goto (I_0, d)

A ⇒ d•, a

B → d•, c

The goto on d in state I_0 is applied with corresponding second components as it is.

Continuing in this fashion we will get further states.

I_6 : goto (I_2, a)

S → Aa•, $

I_9 : goto (I_3, d)

A → d•, c

B → d•, a

I_7 : goto (I_3, A)
S → bA • c, $

I_{10} : goto (I_4, c)
S → Bc • , $

I_8 : goto (I_3, B)
S → bB • a, $

I_{11} : goto (I_7, c)
S → bAc • , $

I_{12} : goto (I_8, a)
S → bBa • , $

Now using the above set of LR(1) items we will construct LR(1) parsing table as follows.

State	Action					Goto		
	a	b	c	d	$	S	A	B
0		s3		s5		1	2	4
1					ACCEPT			
2	s6							
3				s9			7	8
4			s10					
5	r5		r6					
6					r1			
7			s11					
8	s12							
9	r6		r5					
10					r3			
11					r2			
12					r4			

We can parse the string "bda" using above constructed LR (1) parsing table as :

Stack	Input buffer	Action
$0	bda$	Shift 3
$0b3	da$	Shift 9
$0b3d9	a$	Reduce by B → d
$0b3B8	a$	Shift 12
$0b3B8a12	$	Reduce by S → bBa
$0S1	$	Accept

Now we will construct a set of LALR(1) items. In this construction we will simply merge the states deriving same production rules which differ in their second components only. In above set of LR(1) items state I_5 and I_9 are such states i.e.

I_5 : goto (I_0, d)
A → d • , a
B → d • , c

I_9 : goto (I_3, d)
A → d • , c
B → d • , a

We will form only one state by merging state 5 and 9 as

I_{59} : goto (I_0, d)
A → d • , a / c
B → d • , a / c

Hence the LALR(1) set of items are given as :

I_0 : S' → • S, $
S → • Aa, $
S → • bAc, $
S → • Bc, $
A → • d, a
B → • d, c

I_1 : goto (I_0, S)
S' → S • , $

I_2 : goto (I_0, A)
S → A • a, $

I_3 : goto (I_0, b)
S → b • Ac, $

I_6 : goto (I_2, a)
S → Aa • , $

I_7 : goto (I_3, A)
S → bA • c, $

I_8 : goto (I_3, B)
S → bB • a, $

I_{10} : goto (I_4, c)
S → Bc • , $

I_{11} : goto (I_7, c)
S → bAc • , $

I_{12} : goto (I_8, a)
S → bBa • , $

S → b•Ba, $

A → •d, c

B → •d, a

I_4 : goto (I_0, B)

S → B•c, $

I_{59} : goto (I_0, d)

A → d•, a / c

B → d•, a / c

The LALR parsing table will be -

State	Action					Goto		
	a	b	c	d	$	S	A	B
0		s3		s5		1	2	4
1					ACCEPT			
2	s6							
3				s9			7	8
4			s10					
59	r5 r6		r5 r6					
6					r1			
7			s11					
8	s12							
10					r3			
11					r2			
12					r4			

The parsing table shows multiple entries in Action [59, a] and Action [59, c]. This is called **reduce/reduce conflict**. Because of this conflict we cannot parse input.

Thus it is shown that given grammar is LR(1) but not LALR(1)

Q.60 Design LALR (1) parser for following grammer

S → a A d | b B d | a B c | b A c|

A → e

B → e

[JNTU : Part B, June-14, Marks 8]

Ans. : We will first write an augmented grammar and number it out.

1. S' → S
2. S → a A d
3. S → b B d
4. S → a B c
5. S → b A c
6. A → e
7. B → e

Now, let us generate LR (1) Canonical set of items.

I_0 : S' → • S, $

S → • a A d, $

S → • b B d, $

S → • a B c, $

S → • b A c, $

I_1 : goto (I_0, S)

S' → S •, $

→ I_6 : goto (I_2, e)

A → e •, d

B → e •, c

I_2: goto (I_0, a)

S → a •Ad, $

S → a • Bc, $

A → • e, d

B → • e, c

I_3: goto (I_0, b)

S → b • Bd, $

S → b • A c, $

A → • e, c

B → • e, d

I_4: goto (I_2, A)

S → aA • d, $

I_7: goto (I_3, B)

S → bB • d, $

I_8: goto (I_3, A)

S → bA • c, $

→ I_9 : goto (I_3, e)

A → e •, c

B → e •, d

I_{10}: goto (I_4, d)

S → aAd •, $

I_{11}: goto (I_5, c)

S → aBc •, $

I_{12} : goto (I_7, d)

I_5 : goto (I_2, B) $S \rightarrow bBd \bullet, \$$

$S \rightarrow aB \bullet c, \$$ I_{13} : goto (I_8, c)

$S \rightarrow bAc \bullet, \$$

From above construction we can combine I_6 and I_9 to from I_{69} state as,

I_{69}: $A \rightarrow e \bullet, c/d$ and $B \rightarrow \bullet e, c/d$

The parsing table can be constructed as follows :

	Action						goto		
	a	b	c	d	e	$	S	A	B
0	S2	S3					1		
1						Accept			
2					S6			4	5
3					S9			8	7
4				S10					
5			S11						
69			r6/r7	r6/r7					
7				S12					
8			S13						
10						r2			
11						r4			
12						r3			
13						r5			

As we get Table [69] as reduce/reduce conflict given grammar is not LALR.

Q.61 Compare and contrast between SLR, LALR and LR parsers ☞ **[JNTU : Part A, May-18, Marks 10]**

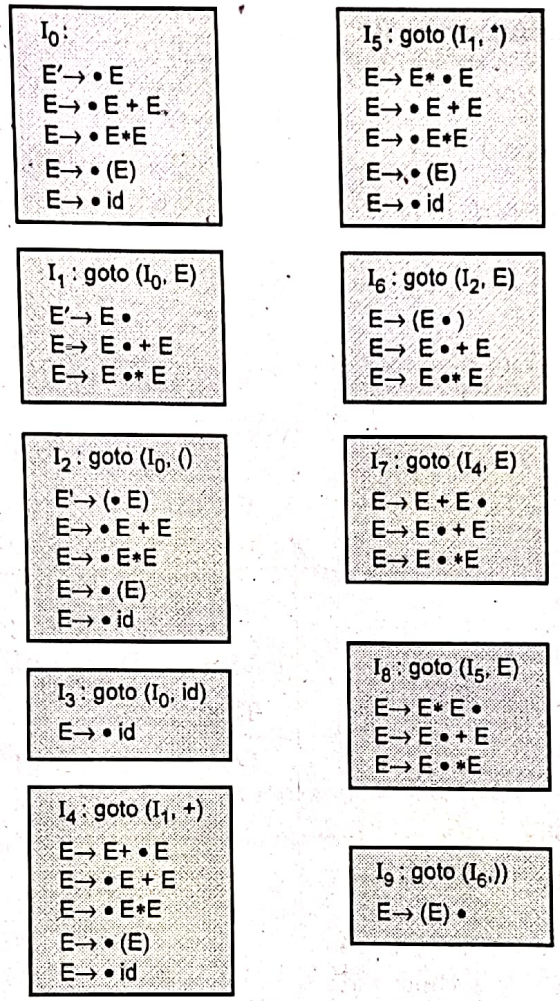
Ans. :

Sr. No.	SLR parser	LALR parser	Canonical LR parser
1.	SLR parser is smallest in size.	The LALR and SLR have the same size.	LR parser or canonical LR parser is largest in size.
2.	It is an easiest method based on FOLLOW function.	This method is applicable to wider class than SLR.	This method is most powerful than SLR and LALR.
3.	This method exposes less syntactic features than that of LR parsers.	Most of the syntactic features of a language are expressed in LALR.	This method exposes less syntactic features than that of LR parsers.
4.	Error detection is not immediate in SLR.	Error detection is not immediate in LALR.	Immediate error detection is done by LR parser.
5.	It requires less time and space complexity.	The time and space complexity is more in LALR but efficient methods exist for constructing LALR parsers directly.	The time and space complexity is more for canonical LR parser.

2.8 Using Ambiguous Grammar

Q.62 Find the SLR parsing table for the given grammar.
E->E+E | E*E | (E) |id.
And parse the sentence (a+b)*c
☞ **JNTU : Part B, Nov.-15, Marks 10]**

Ans. :



Here FOLLOW(E) = {+, *,), $}.

We have computed FOLLOW (E) as it will be required while processing.

Now using the same rules of building SLR(0) parsing table we will generate the parsing table for above set of items.

State	Action						Goto
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			Accept	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		(s4)	(s5)		s9		
7		(s4 or r1)	(s5 or r1)		r1	r1	
8		s4 or r2	s5 or r2		r2	r2	
9			r3		r3	r3	

Simulation of (a + b)*c

Stack	Input	Actions with conflit resolution
$0	(a + b) * C$	Shift
$0(2	a + b) * C$	Shift
$0(2a3	+ b) * C$	Reduce by E → id
$0(2E6	+ b) * C$	Shift
$0(2E6+4	b) * C$	Shift
$0(2E6 +4b3	) * C$	Reduce by E → id
$0(2E6+4E7	) * C$	Reduce by E → E + E
$0(2E6	) * C$	Shift
$0(2E6)9	* C$	Reduce by E → (E)
$0E1	*C$	Shift
$0E1*5	C$	Shift
$0E1*5C3	$	Reduce by E → id
$0E1*5E8	$	Reduce by E → E * E
$0E1	$	Accept

2.9 Parser Generators

Q.63 Give the specification of the YACC Parser generator ☞ **[JNTU : Part A, Dec.-16, Marks 2]**

Ans. : The YACC specification file consists of three parts declaration section, translation rule section and supporting C functions.

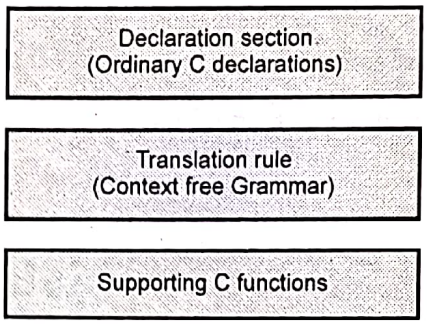


Fig. Q.63.1 Parts of YACC specification

The specification file with these sections can be written as

```
%{
/* declaration section */
%}
%%
 /* Translation rule section */
%%
/* Required C functions*/
```

1. **Declaration part :** In this section ordinary C declarations can be put. Not only this we can also declare grammar tokens in this section. The declaration of tokens should be within %{ and %}.
2. **The translation rule section :** It consists of all the production rules of context free grammar with corresponding actions. For instance

```
rule 1      action 1
rule 2      action 2
.
.
.
rule n      action n
```

If there are more than one alternatives to a single rule then those alternatives should be separated by | character. The actions are typical C statements. If CFG is

LHS → alternative 1 | alternative 2 | ..... | alternative n

Then

```
LHS :  alternative 1    { action 1}
     | alternative 2    { action 2}
       .
       .
       .
       alternative n    {action n}
       ;
```

3. **C functions section :** This section consists of one main function in which the routine yyparse() will be called. And it also consists of required C functions.

Q.64 Discuss in brief about YACC.

☞ **[JNTU : Part B, Nov.-17, March-17, Marks 5]**

OR Write short notes on - YACC.

☞ **[JNTU : Part A, May-18, Marks 3]**

Ans. : • YACC is one such automatic tool for generating the parser program.

- YACC stands for **Yet Another Compiler Compiler** which is basically the utility available from UNIX.
- Basically YACC is **LALR parser** generator.
- The YACC can report conflicts or ambiguities (if at all) in the form of error messages.

- The LEX tool is for lexical analyzer. LEX and YACC work together to analyse the program syntactically.
- The typical YACC translator can be represented as shown in Fig. Q.64.1.

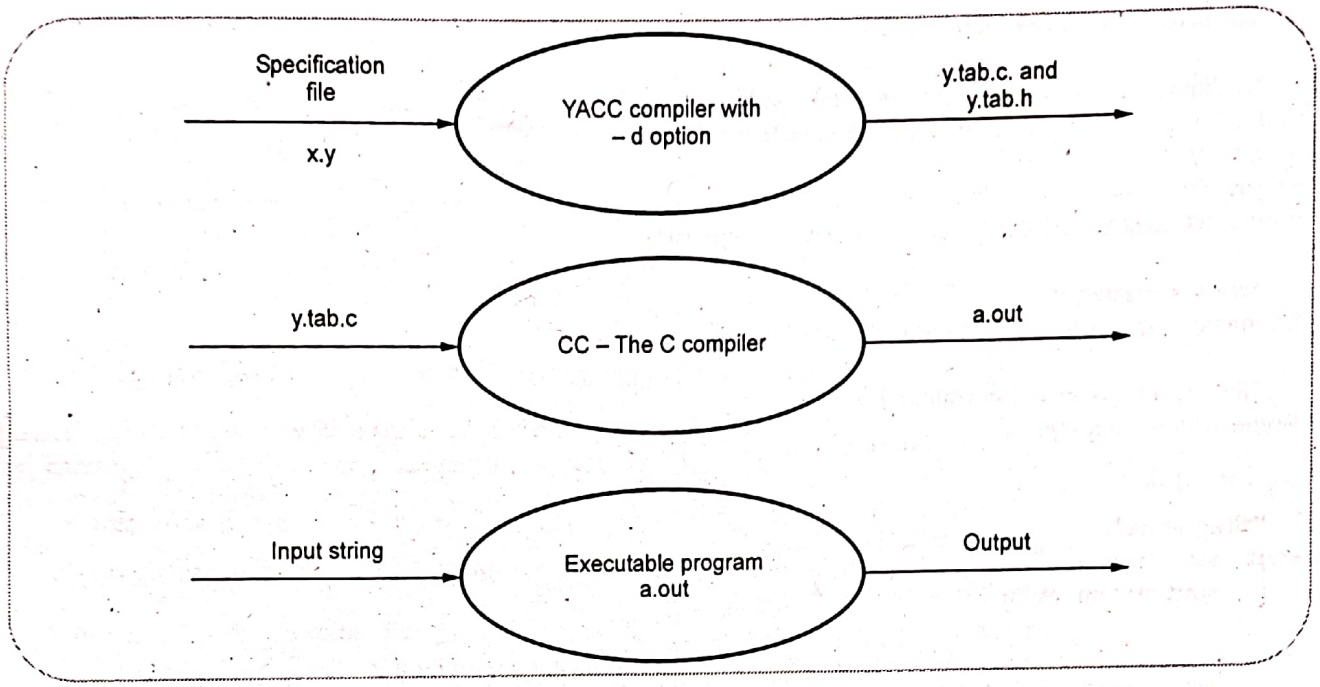


Fig. Q.64.1 YACC : Parser generator model

- First we write a YACC specification file; let us name it as **x.y**. This file is given to the YACC compiler by UNIX command

  yacc x.y

- Then it will generate a parser program using your YACC specification file. This parser program has a standard name as **y.tab.c**. This is basically parser program in C generated automatically. You can also give the command with **– d** option

  yacc – d x.y

By – d option two files will get generated one is **y.tab.c** and other is **y.tab.h**. The header file y.tab.h will store all the tokens and so you need not have to create y.tab.h explicitly.

The generated **y.tab.c** program will then be compiled by C compiler and generates the executable **a.out** file. Then you can test your YACC program with the help of some valid and invalid strings.

Q.65 Write a YACC program that will take regular expression as input and produce its parse tree as output.

☞ [JNTU : Part B, March-16, Nov.-16, Marks 5]

Ans. :

```
/*Program Name :calci.y */
%{
  double memvar;
%}

  /*To define possible symbol types*/
%union
{
  double dval;
}
```

```
  /*Tokens used which are returned by lexer*/
%token <dval> NUMBER
%token <dval> MEM
%token LOG SINE nLOG COS TAN

  /*Defining the precedence and associativity*/
%left '-' '+'              /*Lowest precedence*/
%left '*' '/'
%right '^'
%left LOG SINE nLOG COS TAN /*Highest precedence*/

  /*No associativity*/
%nonassoc UMINUS       /*Unary Minus*/

  /*Sets the type for non-terminal*/
%type <dval> expression

%%
  /*Start state*/
start:  statement '\n'
    |   start statement '\n'
    ;

        /*For storing the answer(memory)*/
statement: MEM '=' expression     {memvar = $3;}
    |   expression   {printf("Answer = %g\n",$1);}
    ;   /*For printing the answer*/

        /*For binary arithmetic operators*/
expression:  expression '+' expression {$$ = $1 + $3;}
    |   expression '-' expression {$$ = $1 - $3;}
    |   expression '*' expression {$$ = $1 * $3;}
    |   expression '/' expression
        {   /*To handle divide by zero case*/
            if($3 == 0)
             yyerror("divide by zero");
            else
             $$ = $1 / $3;
        }
    |   expression '^' expression    {$$ = pow($1,$3);}
    ;

        /*For unary operators*/
expression:  '-' expression   %prec UMINUS  {$$ = -$2;}
        /*%prec UMINUS signifies that unary minus should have
          the highest precedence*/
    |   '(' expression ')'        {$$ = $2;}
    |   LOG  expression       {$$ = log($2)/log(10);}
    |   nLOG   expression       {$$ = log($2);}
        /*Trignometric functions*/
    |   SINE  expression   {$$ = sin($2 * 3.141592654 / 180);}
    |   COS   expression   {$$ = cos($2 * 3.141592654 / 180);}
    |   TAN   expression   {$$ = tan($2 * 3.141592654 / 180);}
    |   NUMBER         {$$ = $1;}
```