R20 CD Unit-2 - compiler design notes

COMPUTER SCIENCE AND ENGINEERING (Andhra University)

UNIT II

Syntax Analysis: Introduction, context-free grammars (CFG), derivation, top-down parsing,
recursive and non-recursive top-down parsers, bottom-up parsing, Operator precedence parser,
Introduction to LR parsing: simple LR parser, more powerful LR parsers, using ambiguous
grammars, parser hierarchy, and automatic parser generator YACC tool.

Syntax Analysis: Introduction, context-free grammars (CFG), derivation, top-down parsing, recursive and non-recursive top-down parsers, bottom-up parsing, Operator precedence parser,

Syntax Analysis: Introduction

A parsing or syntax analysis is a process which takes the input string w and produce either a parse tree (Syntactic structure) or generates the syntactic errors.

Basic Issues in Parsing:
There are two important issues in Parsing:

i. Specification of syntax
ii. Representation of input after parsing

A very important issue in parsing is specification of syntax in programming language. Specification of syntax means how to write any programming statement. There are certain characteristics of specification of syntax:

1. Should be precise and Unambiguous
2. Specification should be in detail,
3. Specification should be complete

This specification is called Context-free grammar.

Derivation and Parse Trees

Derivation from S means generation of string w from S. For constructing derivation two things are important.

i) Choice of non-terminal from several others.
ii) Choice of rule from production rules for corresponding non-terminal.

Definition of derivation tree:

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) ii) Every vertex can be derived from (V U T U ϵ)
iii) If there exists a vertex A with children R1, R2,….,Rn then there should be production A → R_1 R_2 … Rn
iv) The leaf nodes are from set T and interior nodes are from set V.

Instead of choosing the arbitrary non-terminal one can choose.

i) Either leftmost non-terminal in a sentential form then it is called leftmost derivation.
ii) rightmost non-terminal in a sentential form, then it is called rightmost derivation.

Example

Consider the following grammar

S→(L)|a, L→L,S|S

Construct leftmost derivations and parse trees for the following sentences

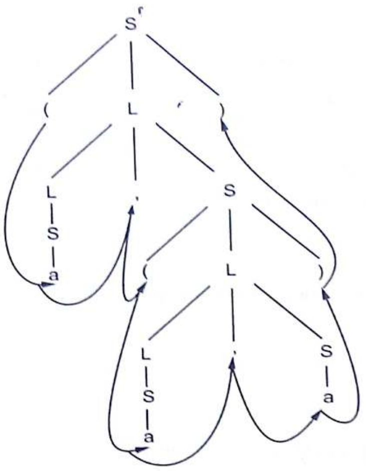
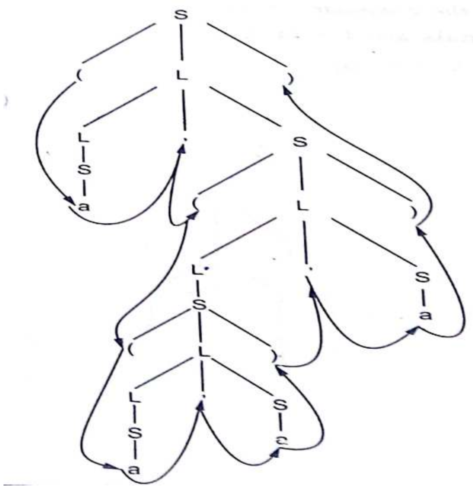
(i) (a,(a,a))   (ii) (a,((a,a),(a,a)))

The Terminals are T = {a,(,)}

The non-terminals are V= {L,S}

The Start Symbol is S

The Parse Tree for

(a,(a,a))	**(a,((a,a),(a,a)))**
	

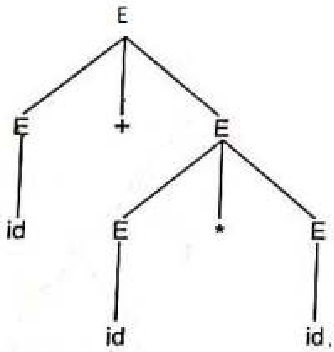
Leftmost derivation:

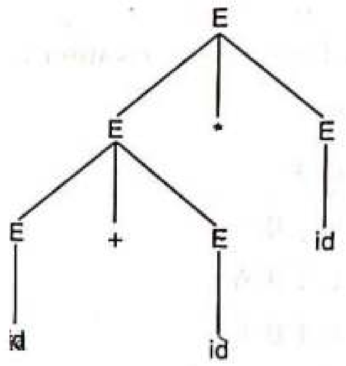
(a,(a,a))	*(a,((a,a),(a,a)))*
S	*S*
(L)	*(L)*
(L,S)	*(L,S)*
(L,(L))	*(S,S)*
(S,(L))	*(a,S)*
(a,(L))	*(a,(L))*
(a,(L,S))	*(a,(L,S))*
(a,(S,S))	*(a,(S,S))*
(a,(a,a))	*(a,((L)),S))*
	(a,((L,S),S))
	(a,((S,S),S))
	(a,((a,a),S))
	(a,((a,a),(L)))
	(a,((a,a),(L,S)))
	(a,((a,a),(S,S)))
	(a,((a,a),(a,a)))

Ambiguous Grammar

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

For example: E→E+E|E*E|(E) |id then id+id*id

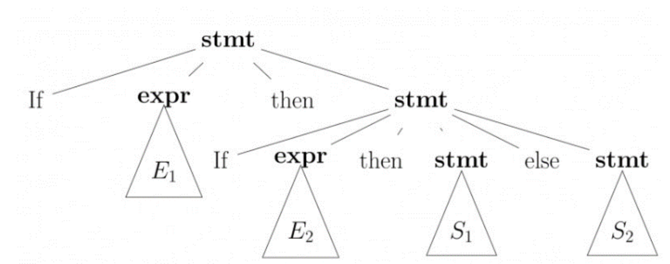


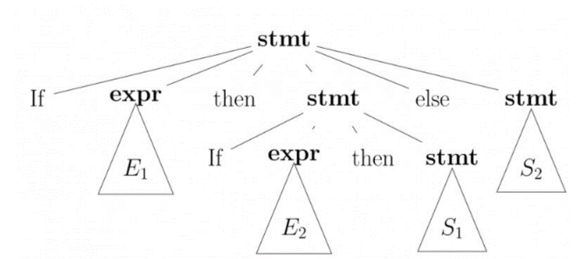


Consider another example,
stmt → if expr then stmt | if expr then stmt else stmt | other

This grammar is ambiguous since the string if E1 then if E2 then S1 else S2 has the following Two parse trees for leftmost derivation

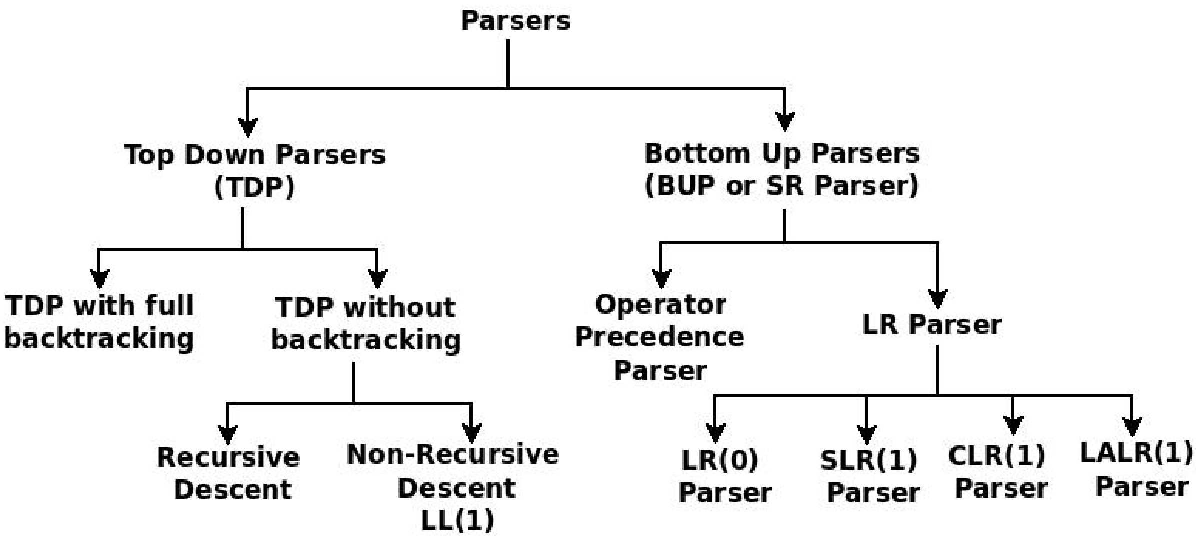
If E1 then if E2 then S1 else S2





Ambiguous Grammar Vs Unambiguous Grammar

Ambiguous Grammar	**Unambiguous Grammar**
A grammar is said to be ambiguous if for at least one string generated by it, it produces more than one. Parse tree Derivation tree Syntax tee Left-most derivation Right-most derivation	A grammar is said to be unambiguous if for all the string generated by it, it produces exactly one Parse tree Derivation tree Syntax tee Left-most derivation Right-most derivation
For ambiguous grammar, leftmost derivation and rightmost derivation represents different parse trees	For unambiguous grammar, leftmost derivation and rightmost derivation represents same parse trees
Ambiguous grammar contains a smaller number of non-terminals	Unambiguous grammar contains a greater number of non-terminals
For ambiguous grammar, length of parse tree is less	For unambiguous grammar, length of parse tree is large
Ambiguous grammar is faster than unambiguous grammar in the derivation of a tree	Unambiguous grammar is slower than unambiguous grammar in the derivation of a tree



Problems with Top-Down Parsing

1. **Backtracking**
2. **Left recursion**
3. **Left factoring**

Backtracking

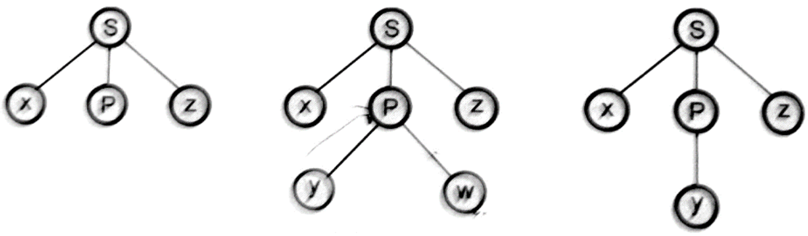
Backtracking is a technique in which for expansion of non-terminal symbol we choose one alternative and if some mismatch occurs then we try another alternative if any.

For Example:

$$S \rightarrow xPz$$

$$P \rightarrow yw|y$$

Then



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.

Left recursion

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow A\alpha$ for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production $A \rightarrow A\alpha \mid \beta$ it can be replaced with a sequence of two productions

$$A \rightarrow \beta A'$$

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

Without changing the set of strings derivable from A.

Example: Consider the following grammar for arithmetic expressions:

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

First eliminate the left recursion for E as

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

Then eliminate for T as

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

Thus, the obtained grammar after eliminating left recursion is

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

Algorithm to eliminate left recursion:

```
1. Arrange the non-terminals in some order A1, A2 . . . An.
2. for i := 1 to n do begin
    for j := 1 to i-1 do begin
         replace each production of the form Ai → Aj γ
         by the productions Ai → δ1 γ | δ2γ | . . . | δk γ.
  where Aj → δ1 | δ2 | . . . | δk are all the current Aj-productions;
  end
  eliminate the immediate left recursion among the Ai- productions
  end
```

Left factoring

If the grammar is left factored then it becomes suitable for the use. Basically, left factoring is used when it is not clear that which of the two alternatives is used to expand the non-terminal. By left factoring we may be able to re-write the production in which the decision can be deferred until enough of the input is seen to make the right choice.

In general if

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

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

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

For example: Consider the following grammar.

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

The left factored grammar becomes,

$S \rightarrow iEtSS' \mid a$ $\quad S' \rightarrow eS \mid \epsilon$ $\quad E \rightarrow b$

For example: Consider the following grammar.

$A \rightarrow aAB \mid aA \mid a$

B→bB|b

If the rule is A→ $\alpha\beta_1|\alpha\beta_2|\ldots$ Is a production then the grammar needs to be left-factored.

Consider, A→aAB|Aa|a
Where $\alpha \rightarrow a$, $\beta_1 \rightarrow AB$, $\alpha \rightarrow a$, $\beta_2 \rightarrow A$, $\alpha \rightarrow a$, $\beta_3 \rightarrow \epsilon$
We have to convert it to

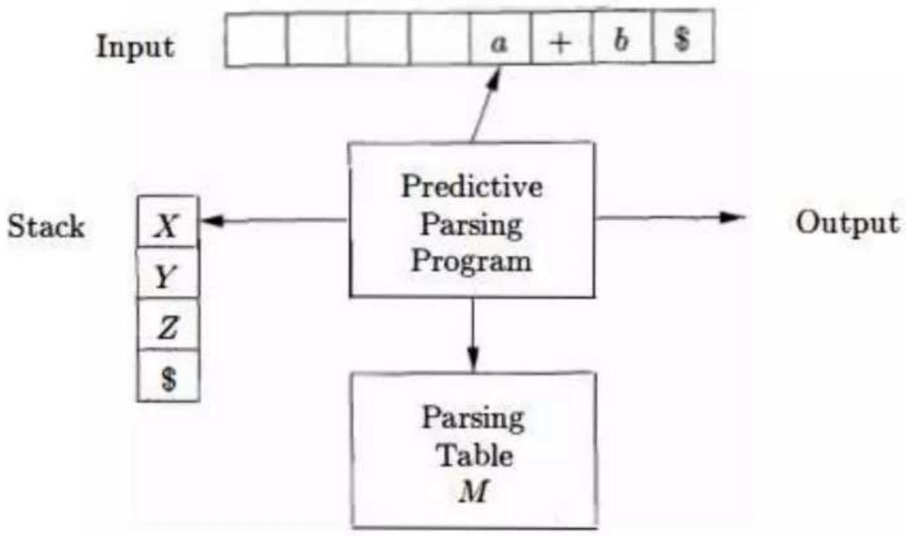
$A \rightarrow \alpha A'$	➔	$A \rightarrow aA'$
$A' \rightarrow \beta_1 \mid \beta_2$	➔	$A' \rightarrow AB \mid A \mid \epsilon$

Similarly,
B → bB|b
Where $\alpha \rightarrow b$, $\beta_1 \rightarrow B$, $\alpha \rightarrow b$, $\beta_2 \rightarrow \epsilon$
We have to convert it to

$A \rightarrow \alpha A'$	➔	$B \rightarrow bB'$
$A' \rightarrow \beta_1 \mid \beta_2$	➔	$B' \rightarrow B \mid \epsilon$

Non-recursive Predictive Parsing:

It is possible to build a non-recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a non-terminal. The non-recursive parser in Fig looks up the production to be applied in a parsing table.



A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of S. The parsing table is a two-dimensional array M[A,a],where A is a non-terminal, and a is a terminal or the symbol $

The program considers X, the symbol on top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1. If X = a =$,the parser halts and announces successful completion of parsing.
2. If X =a ≠$, the parser pops X off the stack and advances the input pointer to the next input symbol.

4. If X is a nonterminal, the program consults entry M[X,a] of the parsing table M.

This entry will be either an X-production of the grammar or an error entry. If, for example, M[X,a] = {X→UVW}, the parser replaces X on top of the stack by WVU (with U on top). If M[X, a] = error, the parser calls an error recovery routine.

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G These functions are FIRST and FOLLOW.

Rules for FIRST ():

1. If X is terminal, then FIRST(X) is {X}.

2. If X → ε is a production, then add ε to FIRST(X).

3. If X is non-terminal and X → aα is a production then add a to FIRST(X).

4. If X is non-terminal and X → Y 1 Y2… Yk is a production, then place a in FIRST(X) if

for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1),… ,FIRST(Yi-1); that is,

Y1,… .Yi-1 => ε. If ε is in FIRST(Yj) for all j=1,2,..,k, then add ε to FIRST(X).

Rules for FOLLOW():

1. If S is a start symbol, then FOLLOW(S) contains $.

2. If there is a production A → αBβ, then everything in FIRST(β) except ε is placed in

follow(B).

3. If there is a production A → αB, or a production A → αBβ where FIRST(β) contains ε, then

everything in FOLLOW(A) is in FOLLOW(B)

Algorithm for construction of predictive parsing table:

Input : Grammar G

Output : Parsing table M

Method :

1. For each production A → α of the grammar, do steps 2 and 3.

2. For each terminal a in FIRST(α), add A → α to M[A, a].

3. If ε is in FIRST(α), add A → α to M[A, b] for each terminal b in FOLLOW(A). If ε is

in FIRST(α) and $ is in FOLLOW(A) , add A → α to M[A, $].

4. Make each undefined entry of M be error.

Algorithm: Non-recursive predictive parsing.

Input: A string w and a parsing table M for grammar G.

Output: If w is in L(G), a leftmost derivation of w; otherwise, an error .

Method: Initially, the parser is in a configuration in which it has $$ on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive

parsing table M to produce a parse for the input.

set ip to point to the first symbol of w$:

repeat

let X be the top stack symbol and a the symbol pointed to by ip;

if X is a terminal or $ then

if X = a then

pop X from the stack and advance ip

else error()

else /* X is a nomerminal */

if M[X,a] =X→Y1Y2…….Yk, then

begin

pop X from the stack:

push Yk , Yk - 1Y1 , onto the stack, with Y1 on top;

output the production X→Y1Y2 …..Yk

end

else error()

until X ≠$ /* stack is empty*/

Example:

Consider the following grammar:

E → E+T | T

T → T*F | F

F → (E) | id

After eliminating left recursion, the grammar is

E → TE'

E' → +TE' | ε

T → FT '

T'→ *FT ' | ε

F → (E) | id

FIRST( ) :

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

FIRST(F) = { (, id }

FOLLOW( ):

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

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

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

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

Predictive parsing table for the given grammar is shown in Fig.

M[X,a]	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 →**id**			F → (E)		

With input id+id*id the predictive parser makes the sequence of moves shown in Fig.

STACK	INPUT	OUTPUT
$E	**id+id*id$**	E →TE'
$E'T	**id+id*id$**	T →FT'
$E'T'F	**id+id*id$**	F →**id**
$E'T'**id**	**id+id*id$**	pop
$E'T'	**+id*id$**	T' →ε
$E'	**+id*id$**	E' →+TE'
$E'T+	**+id*id$**	pop
$E'T	**id*id$**	T →FT'
$E'T'F	**id*id$**	F →**id**
$E'T'**id**	**id*id$**	Pop
$E'T'	***id$**	T' →*FT'
$E'T'F*	***id$**	Pop
$E'T'F	**id$**	F →**id**
$E'T'**id**	**id$**	Pop
$E'T'	**$**	T' →ε
$E'	**$**	E' →ε
$	**$**	Accept

LL(1) Grammars:

For some grammars the parsing table may have some entries that are multiply-defined. For example, if G is left recursive or ambiguous, then the table will have at least one multiply defined entry. A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

Example: Consider this following grammar:
S→ iEtS | iEtSeS | a
E → b
After eliminating left factoring, we have
S→ iEtSS' | a S'→ eS | ε
E → b

To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.

FIRST(S) ={ i, a } FIRST(S') = {e, ε } FIRST(E) = { b}
FOLLOW(S) = { $,e } FOLLOW(S') = { $,e } FOLLOW(E) = {t}

Parsing Table for the grammar:

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

Since there are more than one production for an entry in the table, the grammar is not LL(1) grammar.

LR PARSERS:

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(k) parsing. The "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the "k" for the number of input symbols of lookahead that are used in making parsing decisions. When (k) is omitted, it is assumed to be 1.

LL vs. LR

LL	***LR***
Does a left-most derivation	Does a rightmost derivation in reverse
Starts with the root non-terminal on the stack	Ends with the root non-terminal on the stack
Ends when the stack is empty	Starts with an empty stack
Uses the stack for designating what is still to be expected	Uses the stack for designating what is already seen
Builds the parse tree top-down	Builds the parse tree bottom-up
Continuously pops a nonterminal off the stack and pushes the corresponding right hand side	Tries to recognize a right-hand side on the stack, pops it and pushes the corresponding non-terminals
Expands the non-terminals	Reduces the non-terminals
Reads the terminals when it pops one off the stack	Reads the terminals while it pushes them on the stack
Pre-order traversal of the parse tree	Post-order traversal of the parse tree

Types of LR parsing method:

1. SLR- Simple LR

Easiest to implement, least powerful.

2. CLR- Canonical LR

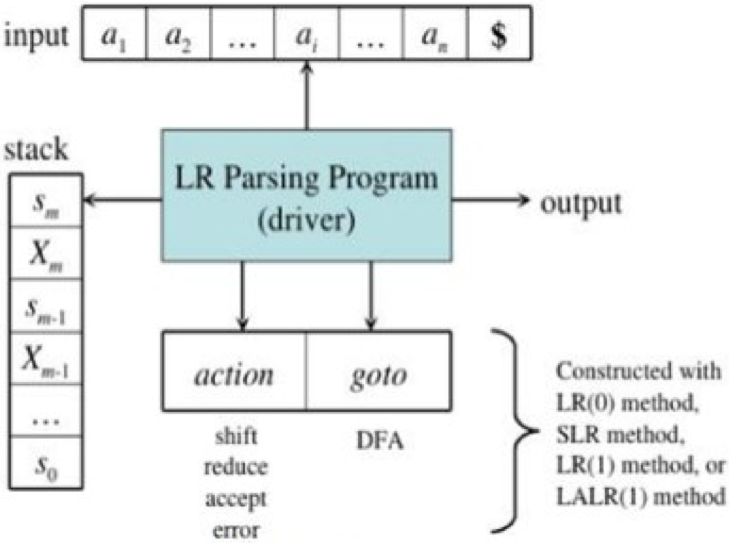
Most powerful, most expensive.

3. LALR- Look -Ahead LR

Intermediate in size and cost between the other two methods

The LR Parsing Algorithm:

The schematic form of an LR parser is shown in Fig 2.25. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (action and goto). The driver program is the same for all LR parser. The parsing table alone changes from one parser to another. The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form s0X1s1X2s2…… Xmsm , where sm is on top. Each Xi is a grammar symbol and each si is a symbol called a state.



Model of an LR Parser

The parsing table consists of two parts: action and goto functions.

Action: The parsing program determines sm, the state currently on top of stack, and ai, the current input symbol. It then consults action[sm,ai] in the action table which can have one of four values:

1. shift s, where s is a state,
2. reduce by a grammar production A → β,
3. accept, and
4. error.

Goto: The function goto takes a state and grammar symbol as arguments and produces a state.

Consider the following grammar

S→TL; T→int|float L→L,id|id

Parse the input string int id,id; using shift-reduce parser

Stack	***Input Buffer***	***Parsing Action***
$	int id,id;$	Shift
$int	id,id;$	Reduce by T→ int
$T	id,id;$	Shift
$Tid	,id;$	Reduce by L→id
$TL	,id;$	Shift
$TL,	id$	Shift
$TL,id	;$	Reduce by L→L,id
$TL	;$	Shift
$TL;	$	Reduce by S→TL;
$S	$	Accepted

Operator Precedence Parsing:

Operator grammars have the property that no production right side is ε (empty) or has two adjacent non-terminals. This property enables the implementation of efficient operator precedence parsers.
Example: The following grammar for expressions:

$$E \rightarrow E\ A\ E \mid (E) \mid -E \mid id$$
$$A \rightarrow + \mid - \mid * \mid / \mid \wedge$$

This is not an operator grammar, because the right side EAE has two consecutives non-terminals. However, if we substitute for A each of its alternate, we obtain the following operator grammar:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid E \wedge E \mid - E \mid id$$

In operator-precedence parsing, we define three disjoint precedence relations between pair of terminals. This parser relies on the following three precedence relations.

Relation	Meaning
a <· *b*	*a* yields precedence to *b*
a =· *b*	*a* has the same precedence as *b*
a ·> *b*	*a* takes precedence over *b*

These precedence relations guide the selection of handles. These operator precedence relations allow delimiting the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

Defining Precedence Relations:

The precedence relations are defined using the following rules:
Rule-01:
- If precedence of b is higher than precedence of a, then we define a < b
- If precedence of b is same as precedence of a, then we define a = b
- If precedence of b is lower than precedence of a, then we define a > b

Rule-02:
- An identifier is always given the higher precedence than any other symbol.
- $ symbol is always given the lowest precedence.

Rule-03:
- If two operators have the same precedence, then we go by checking their associativity.

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

Example: The input string: id1 + id2 * id3
After inserting precedence relations, the string becomes:

$ <· id1 ·> + <· id2 ·> * <· id3 ·> $

Having precedence relations allows identifying handles as follows:
1. Scan the string from left end until the leftmost ·> is encountered.
2. Then scan backwards over any =’s until a <· is encountered.
3. Everything between the two relations <· and ·> forms the handle.

Input string	Precedence relations inserted	Action
id+id*id	$<.id.>+<.id.>*<.id.>$	
E+id*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
E	$$	Accepted

Implementation of Operator-Precedence Parser:

• An operator-precedence parser is a simple shift-reduce parser that is capable of parsing a subset of LR(1) grammars.

• More precisely, the operator-precedence parser can parse all LR(1) grammars where two consecutive non-terminals and epsilon never appear in the right-hand side of any rule.

Example 1: Consider the unambiguous grammar,

E→E + T E→T T→T * F T→F F→(E) F→id

Constructing Precedence Relation Table

	+	*	id	(	)	$
+	>	<	<	<	>	>
*****	>	>	<	<	>	>
id	>	>	**e**	**e**	>	>
(	<	<	<	<	=	**e**
)	>	>	**e**	**e**	>	>
$	<	<	<	<	**e**	**Accept**

Precedence Relation Table

Parse the given input string (id+id)*id$

STACK	REL.	INPUT	ACTION
$	$ < (	(id+id)*id$	Shift (
$(	(< id	id+id)*id$	Shift id
$(id	id > +	+id)*id$	Pop id
$(	(< +	+id)*id$	Shift +
$(+	+ < id	id)*id$	Shift id
$(+id	id >)	)*id$	Pop id
$(+	+ >)	)*id$	Pop +
$(	(=)	)*id$	Shift)
$() $(	) > *	*id $	Pop) Pop (
$	$ < *	*id $	Shift *
$*	* < id	id$	Shift id
$*id	id > $	$	Pop id
$*	* > $	$	Pop *
$		$	Accept

Parse the input string (id+id)*id$

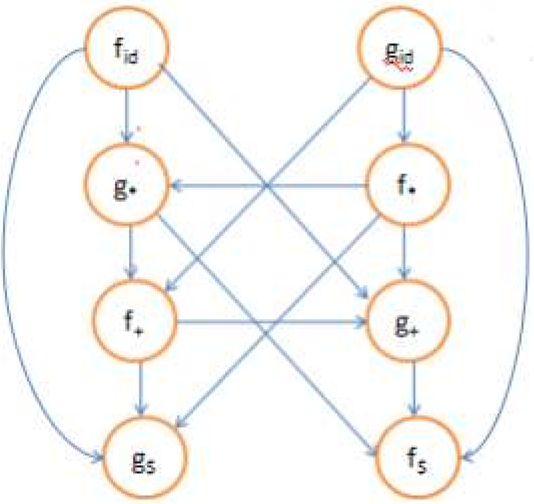
Precedence Functions:

Compilers using operator-precedence parsers need not store the table of precedence relations. In most cases, the table can be encoded by two precedence functions f and g that map terminal symbols to integers. We attempt to select f and g so that, for symbols a and b.

1. f (a) < g(b) whenever a<·b.
2. f (a) = g(b) whenever a = b. and
3. f(a) > g(b) whenever a ·> b.

Algorithm for Constructing Precedence Functions:

1. Create functions fa for each grammar terminal a and for the end of string symbol.
2. Partition the symbols in groups so that fa and gb are in the same group if a = b (there can be symbols in the same group even if they are not connected by this relation).
3. Create a directed graph whose nodes are in the groups, next for each symbols a and b do: place an edge from the group of gb to the group of fa if a <· b, otherwise if a ·> b places an edge from the group of fa to that of gb.
4. If the constructed graph has a cycle, then no precedence functions exist. When there are no cycles collect the length of the longest paths from the groups of fa and gb respectively.



Precedence Graph

There are no cycles,so precedence function exist. As f$ and g$ have no out edges,f($)=g($)=0.The longest path from g+ has length 1,so g(+)=1.There is a path from gid to f* to g* to f+ to g+ to f$,so g(id)=5.The resulting precedence functions are:

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Example 2:

Consider the following grammar-

S → (L) | a   L → L , S | S

Construct the operator precedence parser and parse the string (a , (a , a)).

The terminal symbols in the grammar are { (,) , a , , }
We construct the operator precedence table as-

	a	(	)	,	$
a		>	>	>	>
(	<	>	>	>	>
)	<	>	>	>	>
,	<	<	>	>	>
$	<	<	<	<	

Parsing Given String-
Given string to be parsed is (a , (a , a)).
We follow the following steps to parse the given string-
Step-01:
We insert $ symbol at both ends of the string as-

$ (a , (a , a)) $

We insert precedence operators between the string symbols as-

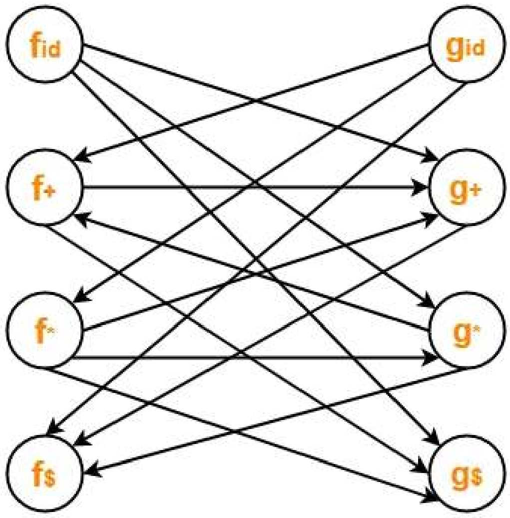
< (< a > , < (< a > , < a >) >) > $

Step-02: We scan and parse the string as-
$ < (< a > , < (< a > , < a >) >) > $
$ < (S , < (< a > , < a >) >) > $
$ < (S , < (S , < a >) >) > $
$ < (S , < (S , S) >) > $
$ < (S , < (L , S) >) > $
$ < (S , < (L) >) > $
$ < (S , S) > $
$ < (L , S) > $
$ < (L) > $
$ < S > $
$ $
Accepted
Example: Consider the following grammar- E → E + E | E x E | id
- Construct Operator Precedence Parser.
- Find the Operator Precedence Functions.

Solution-
The terminal symbols in the grammar are { + , x , id , $ }
We construct the operator precedence table as-

		g →			
		id	+	x	$
f ↓	id		>	>	>
	+	<	>	<	>
	x	<	>	>	>
	$	<	<	<	

The graph representing the precedence functions is-



Here, the longest paths are-

- $f_{id} \rightarrow g_x \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$
- $g_{id} \rightarrow f_x \rightarrow g_x \rightarrow f_+ \rightarrow g_+ \rightarrow f_\$$

The resulting precedence functions are-

	+	x	id	$
f	2	4	4	0
g	1	3	5	0

CONSTRUCTING SLR PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:
1. Find LR(0) items.
2. Completing the closure.
3. Compute goto(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items:

A → •XYZ   A → X•YZ   A → XY•Z   A → XYZ•

Closure operation:

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to closure(I).
2. If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX•β] such that [A→ α•Xβ] is in

1.Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G`
2. Construct the canonical collection of set of items C for G‟
3. Construct the parsing action function action and goto using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G‟

Output : The SLR parsing table functions action and goto for G'

Method :

1. Construct C ={I0, I1, …. In}, the collection of sets of LR(0) items for G'.
2. State i is constructed from Ii. The parsing functions for state i are determined as follows:

(a) If [A→α•aβ] is in Ii and goto(Ii,a) = Ij, then set action[i,a] to "shift j". Here a must be terminal.

(b) If[A→α•] is in Ii , then set action[i,a] to "reduce A→α" for all a in FOLLOW(A).

(c) If [S‟→S•] is in Ii, then set action[i,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3.The goto transitions for state i are constructed for all non-terminals A using the rule: If goto(Ii,A)= Ij, then goto[i,A] = j.

4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→•S]

SLR Parsing algorithm:

Input: An input string w and an LR parsing table with functions action and goto for grammar G.

Output: If w is in L(G), a bottom-up-parse for w; otherwise, an error indication.

Method: Initially, the parser has s0 on its stack, where s0 is the initial state, and w$ in the input buffer. The parser then executes the following program :

set ip to point to the first input symbol of w$;
repeat forever begin
let s be the state on top of the stack and a the symbol pointed to by ip;
if action[s, a] =shift s‟ then begin
push a then s‟ on top of the stack;
advance ip to the next input symbol
end
else if action[s, a]=reduce A→β then begin
pop 2* |β |symbols off the stack;
let s‟ be the state now on top of the stack;
push A then goto[s‟, A] on top of the stack;
output the production A→ β

```
 end
 else if action[s, a]=accept then
return
 else error( )
end
```

Example: Implement SLR Parser for the given grammar:

1.E→E + T

2.E→T

3.T→T * F

4.T→F

5.F→(E)

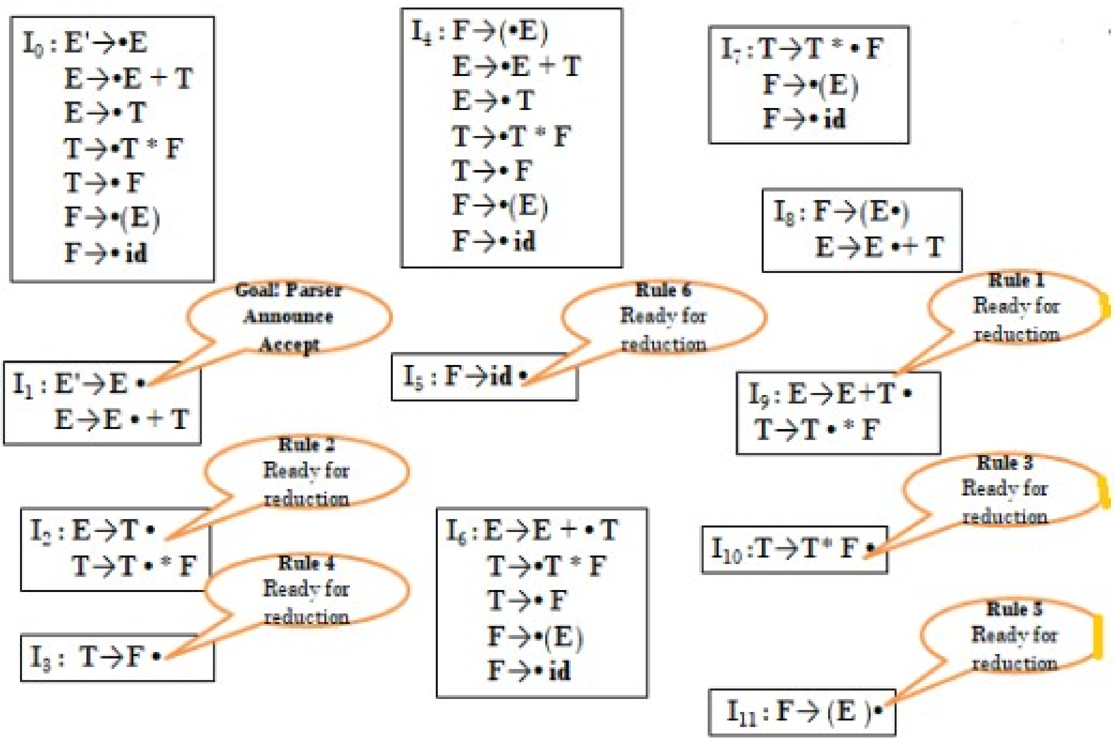
6.F→id

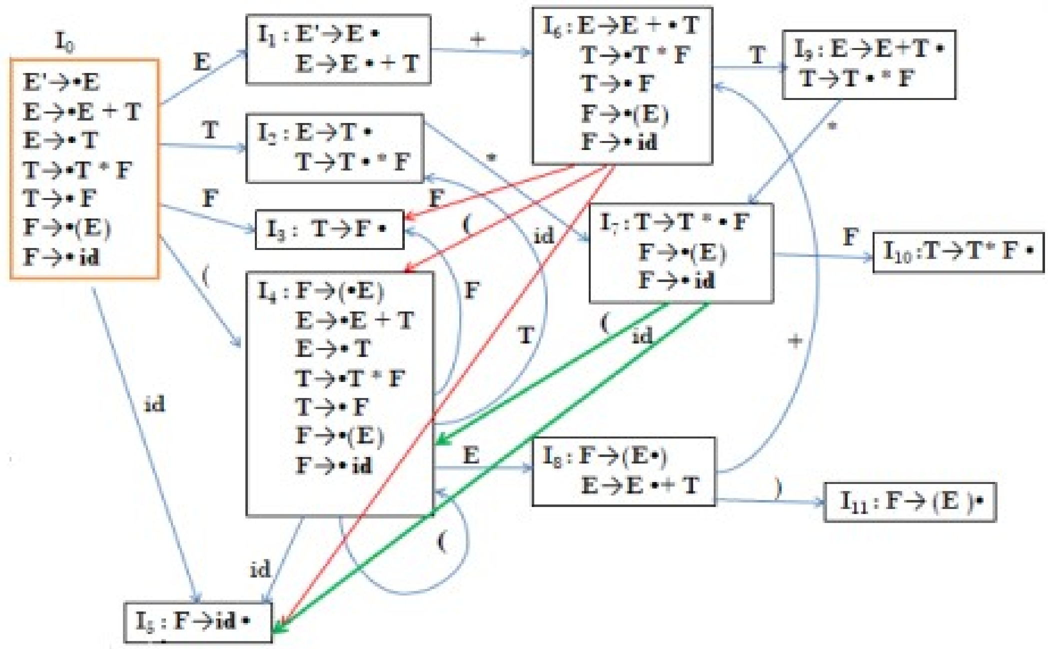
Step 1 : Convert given grammar into augmented grammar.

Augmented grammar:

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

Step 2 : Find LR (0) items.





Step 3 : Construction of Parsing table.

1. Computation of FOLLOW is required to fill the reduction action in the ACTION part of the table.

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

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
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			

1. si means shift and stack state i.
2. rj means reduce by production numbered j.
3. acc means accept.
4. Blank means error.

Step 4: Parse the given input. The Fig shows the parsing the string id*id+id using stack implementation

Stack	Input	Action
0	id*id+id$	s5 Shift 5
0 id 5	*id+id$	r6 Reduce by F→id
0 F 3	*id+id$	r4 Reduce by T→F
0 T 2	*id+id$	s7 Shift 7
0 T 2 * 7	id+id$	s5 Shift 5
0 T 2 * 7 id 5	+id$	r6 Reduce by F→id
0 T 2 * 7 F 10	+id$	r3 Reduce by T→T*F
0 T 2	+id$	r2 Reduce by E→T
0 E 1	+id$	s6 Shift 6
0 E 1+ 6	id$	s5 Shift 5
0 E 1+ 6 id 5	$	r6 Reduce by F→id
0 E 1+ 6 F 3	$	r4 Reduce by T→F
0 E 1+ 6 T 9	$	r1 Reduce by E→E+T
0 E 1	$	Accept

Construct LR(0) parsing table for the following grammar

S→cA|ccB   A→cA|a   B→ccB|b

Solution:

Now we will construct set of items

S→cA   S→ccB   A→cA   A→a   B→ccB   B→b

I_0:

S'→.S
S→.cA
S→.ccB
A→.cA
A→.a
B→.ccB
B→.b

I_1: goto(I_0,S)

S'→.S

I_2: goto(I_0,c)

S→c.A
S→c.cB
A→c.A
B→c.cB
A→.cA
A→.a

I_3: goto(I_0,a)

A→a.

I_4: goto(I_0,b)

B→b.

I_5: goto(I_2,A)

S→cA.
A→cA.

I_6: goto(I_2,c)

S→cc.B
B→cc.B
A→c.A
A→.cA
A→.a
B→.ccB
B→.b

I_7: goto(I_6,B)

S→ccB.
B→ccB.

I_8: goto(I_6,A)

A→cA.

I_9: goto(I_6,c)

A→c.A
B→c.cB
A→.cA
A→.a

I_{10}: goto(I_9,c)

B→cc.B
A→c.A
A→.cA
A→.a
B→.ccB
B→.b

I_{11}: goto(I_{10},B)

B→ccB.

FOLLOW(S) ={$} FOLLOW(A)={$} FOLLOW(B)={$}

The SLR parsing table can be constructed as follows:

States	***Action***				***Goto***		
	a	b	c	$	S	A	B
0	S3	S4	S2		1		
1				Accept			
2	S3		S6			5	
3				r4			
4				r6			
5				r1/r3			
6	S3	S4	S9			8	7
7				r2/r5			
8				r3			
9	S3		S10			8	
10	S3	S4	S9			8	11

11				r5			

Construct the **LR(1)** parsing table for the following grammar

S→CC   C→aC   C→d

Solution:

We will initially add S'→.S,$ as the first rule in I_0. Now watch

S'→.S,$ with

[A→α.Xβ,a] hence S'→.S,$

A=S',α=ε,X=S,β=ε,a=$

If there is a production X→γ,b then add X**→.γ,b**

S→.CC,   b∈FIRST(βa)

b∈FIRST(ε $) as ε$=$

b∈FIRST($)

b={$)

S→.CC,$ will be added in Io.

Now S→.CC,$ is in Io we will match it with A→α.Xβ,a

A=S',α=ε,X=C,β=C,a=$

If there is a production X→γ,b then add X→γ,b

C→.aC,   b∈FIRST(β a)

C→.d   b∈FIRST(C $)

b∈FIRST(C) as FIRST(C)={a,d}

b={a,d}

C→.aC, a or d will be added in I_0.

Similarly, C→.d,a/d will be added in I_0.

Hence

I_0:

S'→.S,$

S→.CC,$

C→.aC,a/d

C→.d,a/d

Now apply goto on **I_0**.

S'→.S,$

S→.CC,$

C→.aC,a/d

C→.d,a/d

Hence

I_1: goto(I_0,S)

S'→S.,$

Now apply goto on C in I_0.

S→C.C,$ add in I_2. Now as after dot C comes we will add the rules of C.

X=C,β=ε,a=$

X→.γ,b where   b∈FIRST(β a)

C→.aC   b∈FIRST(∈ $)

C→.d   b∈FIRST($)

Hence C$\rightarrow$.aC,\$ and C$\rightarrow$.d,\$ will be added to I_2.

I_2: goto(I_0,C)

S$\rightarrow$C.C,\$
C$\rightarrow$.aC,\$
C$\rightarrow$.d,\$

Now we will apply goto on a of I_0 for rule C$\rightarrow$.aC,a/d that becomes C$\rightarrow$.aC,a/d will be added in I_3.

C$\rightarrow$a.C,a/d
As A=C,α=a,X=C,β=ε,a=a/d
Hence X$\rightarrow$.γ,b
C$\rightarrow$.aC b∈FIRST(βa)
b∈FIRST(ε a) or FIRST(ε b)
b=a/d

I_3: goto(I_0,a)

C$\rightarrow$a.C,a/d
C$\rightarrow$.aC,a/d
C$\rightarrow$.d,a/d

I_4: goto(I_0,d)

C$\rightarrow$d.,a/d

I_5: goto(I_2,C)

S$\rightarrow$CC.,\$

Apply goto on a form I_2 on the rule C$\rightarrow$.aC,\$ we will get the state I6.

I_6: goto(I_2,a)

C$\rightarrow$a.C,\$
C$\rightarrow$.aC,\$
C$\rightarrow$.d,\$

You can note one thing that I_3 and I_6 are different because the second component in I_3 and I_6 is different.

I_7: goto(I_2,d)

C$\rightarrow$d.,\$

I_8: goto(I_3,C)

C$\rightarrow$aC.,a/d

I_9: goto(I_6,C)

C$\rightarrow$aC.,\$

You can note one thing that I_4,I_7 and I_8,I_9 are different because the second component in I_4,I_7 and I_8,I_9 is different.

For remaining states I_7,I_8 and I_9 we cannot apply goto. Hence the process of construction of set of LR(1) items is completed. Thus the set of LR(1) items consists of I_0 and I_9 states.

States	***Action***			***Goto***	
	a	d	\$	S	C

0	S3	S 4		1	2
1			Accep t		
2	S 6	S7			5
3	S3	S 4			8
4	r3	r3			
5			r1		
6	S 6	S7			9
7			r3		
8	r2	r2			
9			r2		

Construction of LALR parsing Table:

The LALR parsing table is constructed as follows.

1. Construct the collection of sets of LR(1) items.
2. Merge the two states Ii and Ij if the first component is matching and replace the two states with the merged state i.e. Iij=IiU Ij.
3. Build the LALR parse table similar to LR(1) parse table.
4. Parse the input string using LALR parse table similar to LR(1) parsing.

As per the algorithm, in the given LR(1) items we have the following set of items with the same core

- I3 and I6
- I4 and I7
- I8 and I9

Now, after merging the above-mentioned sets we get,

I36:

C → c.C, c/d/$
C → .cC, c/d/$
C → d., c/d/$

I47:

C → d., c/d/$

I89:

C → cC., c/d/$

Now the LALR(1) items are

I0

S' → .S, $
S → .CC, $
C → .cC, c/d
C → .d, c/d

I1:

S → S., $

I2:

S → C.C, $

C → .cC, $
C → .d, $

I36:
C → c.C, c/d/$
C → .cC, c/d/$
C → d., c/d/$

I47:
C → d., c/d/$

I5:
S → CC., $

I89:
C → cC., c/d/$

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		

Example:- Show that the following grammar is LR(1) but not LALR(1).
S→Aa | bAc |Bc|bBa
A→d
B → d

Solution:- Convert the above grammar as augmented grammar.
S'→S
S → Aa | bAc |Bc|bBa
A →d
B → d

The initial set of items is
I0:
S' → ·S, $
S →· Aa, $
S → · bAC, $
S →· B c, $
S → ·bBa ,$
A →·d
B → ·d

Computing the remaining canonical items, we have

I1: goto(I0,S)
S'→S.,$

I2: goto(I0,A)
S → A·a, $

I3: goto(I0,b)
S→b.Ac,$
S→b.Ba,$

A→.d,c
B→.d,a

I4: goto(I0,B)
S→B.c,$

I5: goto(I0,d)
A→d.,a
B→d.,c

I6: goto(I2,a)
S→Aa.,$

I7: goto(I3,A)
S→bA.c,$

I8: goto(I3,B)
S→bB.a,$

I9: goto(I3,d)
A→d.,c
B→d.,a

I10: goto(I4,c)
S→Bc.,$

I11: goto(I7,c)
S→bAc.,$

I12: goto(I8,a)
S→bBa.,$

The remaining sets of items yield no GOTO'S, so we are done.

The LR(1) parsing table for the above grammar is

States	***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				

As the first component of states I5 and I9 are same we merge the two states to get I59.

I59: GOTO(I0/ I3,d)

A →d· , a/c

B →d· , c/a

States	***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		r6/r5					
6				r1				
7			S11					
8	S12							
10				r3				
11				r2				
12				r4				

The LALR 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 the given grammar is LR(1) but not LALR.

Error Recovery in LR Parsing: -

An LR parser will detect an error when it consults the parsing action table and finds an error entry. Errors are never detected by consulting the goto table. An LR parser will announce an error as soon as there is no valid continuation for the portion of the input thus far scanned. A canonical LR parser will not make even a single reduction before announcing an error. SLR and LALR parsers may make several
reductions before announcing an error, but they will never shift an erroneous input symbol onto the stack.

Panic-mode error recovery: -

In LR parsing, we can implement panic-mode error recovery as follows. We scan down the stack until a state s with a goto on a particular nonterminal A is found. Zero or more input symbols are then discarded until a symbol a is found that can legitimately follow A. The parser then stacks the state GOTO(S,a) and
resumes normal parsing. This method of recovery attempts to eliminate the phrase containing the syntactic error. If the parser determines that a string derivable from A contains an error. Part of that string has already been processed, and the result of this processing is a sequence of states on top of the stack. The remainder of the string is still in the input, and the parser attempts to skip the remainder of this string by looking a terminal that follows A. By removing states from the stack, skipping over the input, and pushing GOTO(s, A) on the stack, the parser pretends that it has found an instance of A and resumes normal parsing.

Phrase-level error recovery: -

Phrase-level recovery is implemented by examining each error entry in the LR parsing table and an appropriate recovery procedure can then be constructed; In designing specific error-handling routines for an LR parser, we can fill in each blank entry in the action field with a pointer to an error routine that will take the appropriate action selected by the compiler designer. The actions may include insertion
or deletion of symbols from the stack or the input or both, or alteration and transposition of input symbols. The modifications should be such that the LR parser will not get into an infinite loop. A safe

strategy will assure that at least one input symbol will be removed or shifted eventually, or that the stack will eventually shrink if the end of the input has been reached.