UNIT-III
Bottom-up Parsing

Objective:
To understand Bottom-up Parsing.

Syllabus:
Shift-Reduce parsing, operator precedence parsing, LR parsers: Construction of SLR, CLR (1), LALR Parsers.

Learning Outcomes:
Students will be able to

- understand shift reduce parsing.
- implement operator precedence parser.
- construct bottom-up LR parsers like SLR, CLR and LALR.

Bottom-up parsing:

- Bottom-up parsing attempts to construct a parse tree for the given input string in the bottom-up manner.
- Parse tree is constructed for an input string beginning at the leaves(the bottom) and working up towards the root(the top)
- A general type of bottom-up parser is a shift-reduce parser.

Shift-Reduce Parsing:

- A general style of bottom-up syntax analysis is Shift-Reduce parsing.
- Shift-Reduce parsing attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root.
- For example, consider the grammar
  s->aABe
  A->Abc|b
  B->d
  The sentence **abbcde** can be reduced to S by the following steps

abbcde	(A->b)
aAbcde	(A->Abc)
aAde	(B->d)
aABe	(S->aABe)
S	

Handle:

- A handle of a string is a substring that matches the right side of a production and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a right most derivation.

Handle Pruning:

- A rightmost derivation in reverse can be obtained by handle pruning. That is we start with a string of terminals w that we wish to parse.
- If w is a sentence of the grammar at hand, then w=γ_n , where γ_n is the nth right-sentential form of some rightmost derivation.

$$\Rightarrow S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \quad \gamma_n = w$$

Stack implementation of Shift-Reduce Parsing:

- A way to implement a shift-reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string w to be parsed.
- $ is used to mark the bottom of the stack and also right end of the input.
- Initially, the stack is empty, and the string w is on the input.
- The parser operates by shifting zero or more input symbols onto the stack until a handle β is on top of the stack
- The parser then reduces β to the left side of the appropriate production. Pop the handle and reduce the handle with its non-terminal and push it on to the stack.
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.
- After entering this configuration, the parser halts and announces successful completion of parsing.

Actions in shift-reduce parser:

1. Shift: The next input symbol is shifted onto the top of the stack
2. Reduce: The parser knows the right end of the handle is at the top of the stack and replaces the handle with a non-terminal.
3. Accept: The parser announces successful completion of parsing
4. Error: The parser discovers that a syntax error has occurred and calls an error recovery routine.

Example:

S->aABe

A->Abc|b

B->d

Implement a shift-reduce parser on the input “abbcde”

Stack	Input	Action
$	abbcde$	shift
$a	bbcde$	shift
$ab	bcde$	Reduce by A->b
$aA	bcde$	shift
$aAb	cde$	shift
$aAbc	de$	Reduce by A->Abc
$aA	de$	shift
$aAd	e$	Reduce by B->d
$aAB	e$	shift
$aABe	$	Reduce by S->aABe
$S	$	Accept

The parser announces successful completion of parsing.

Conflicts in Shift-Reduce Parsing:

- **shift/reduce conflict:** The parser cannot decide whether to shift or to reduce.
- **reduce/reduce conflict:** The parser cannot decide which of several reductions to make.

Operator Precedence Parser:

Operator precedence parser can be constructed for a grammar called operator grammar.

Operator Grammars have the property that no production on right side is ε or has two adjacent non-terminals.

Example:

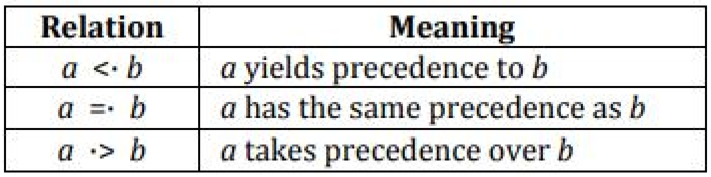
E→EAE|(E)|-E| id

A→+|-|*|/

The grammar can be rewritten as

E→E+E|E-E|E*E|E/E|(E)|-E| id

Operator precedence relations:

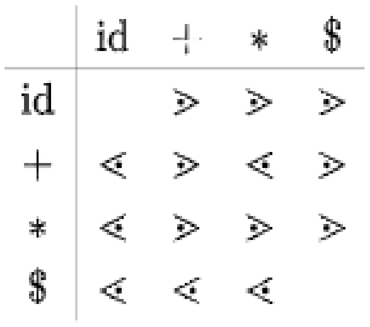


Relation	Meaning
a <· *b*	*a* yields precedence to *b*
a =· *b*	*a* has the same precedence as *b*
a ·> *b*	*a* takes precedence over *b*

Operator-Precedence Relations from Associativity and Precedence:
The rules are designed to select the "proper" handles to reflect a given set of associativity and precedence rules for binary operators.

1. If operator θ1 has higher precedence than operator θ2, make θ1 .> θ2 and θ2 <. θ1.
2. If θ1 and θ2 are operators of equal precedence, then make
   θ1 .> θ2 and θ2 .> θ1, if the operators are left associative
   θ1 <. θ2 and θ2 <. θ1, if the operators are right associative.
3. Make the following for all operators θ
   θ <. id and id .> θ,
   θ <. (and (<. θ,
) .> θ and θ .>),
   θ .> $ and $.> θ,
   Also make
   (=), $ <. (, $ <. id,
   (<. (, id .> $,) .> $,
   (<. id, id .>),) .>)

Example:
Operator precedence relations for the grammar E→E+E|E*E| id



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

Operator-Precedence parsing algorithm:

Input: An input string w and a table of precedence relations.

Output: If well formed with a placeholder nonterminal E labeling all interior nodes; otherwise, an error indication.

Method: Initially, the stack contains $ and the input buffer the string w$.

```
(1) set ip to point to the first symbol of w$;
(2) repeat forever
(3)       if $ is on top of the stack and ip points to $ then
(4)             return
          else begin
(5)             let a be the topmost terminal symbol on the stack
                and let b be the symbol pointed to by ip;

(6)             if a <. b or a = b then begin
(7)                   push b onto the stack;
(8)                   advance ip to the next input symbol;
                end;
(9)             else if a .> b then       /* reduce */
(10)                  repeat
(11)                        pop the stack
(12)                    until the top stack terminal is replaced by
<.
                        to the terminal most recently popped
(13)            else error( )
             end;
```

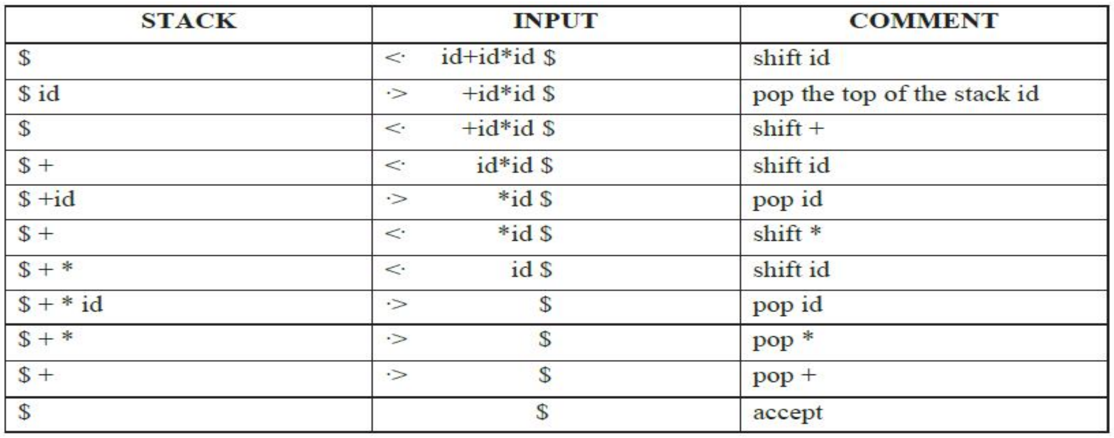
Stack implementation of operator precedence parser:

Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm.

Example:

Consider the grammar E→E+E| E*E| id.

Input string is id+id*id



STACK	INPUT	COMMENT
$	<· id+id*id $	shift id
$ id	·> +id*id $	pop the top of the stack id
$	<· +id*id $	shift +
$ +	<· id*id $	shift id
$ +id	·> *id $	pop id
$ +	<· *id $	shift *
$ + *	<· id $	shift id
$ + * id	·> $	pop id
$ + *	·> $	pop *
$ +	·> $	pop +
$	$	accept

LR Parsers:

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

Advantages of LR parsing:

- LR parsers can be constructed to recognize virtually all programming-language constructs for which CFG can be written.
- It is an efficient non backtracking shift-reduce parsing method.
- The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.
- An LR parser can detect a syntactic error as soon possible.

Drawbacks:

- It is too much of work to construct on LR parsers by hand for a typical programming language grammar. A specialized tool called a LR parser generator is needed. Example: YACC

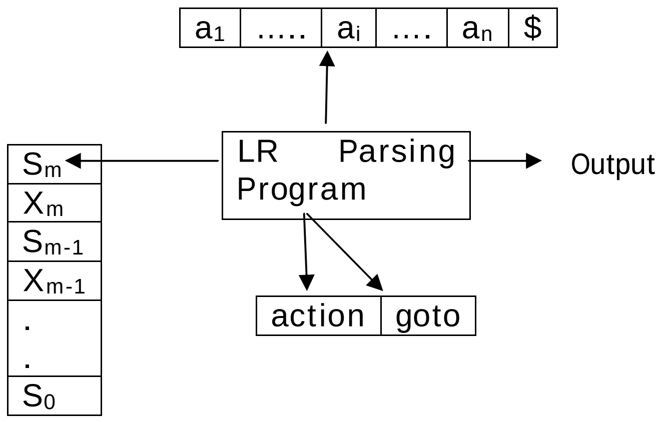
Types of LR parsing methods:

1. Simple LR(SLR) is the easiest to implement, but the least powerful of the three. It may fail to produce a parsing table for certain grammars on which the other methods succeed.
2. Canonical LR (CLR) is the most powerful, and the most expensive

3. Lookahead LR (LALR) is intermediate in power and cost between the other two. The LALR will work on most programming language grammars and with some effort, can be implemented efficiently.

Model of an LR parser:

- The schematic form of an LR parser 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 parsers, only the parsing table changes form 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 $S_0X_1S_1X_2...X_mS_m$, where S_m is on the top.
- Each X_i is a grammar symbol and each S_i is a symbol called a state.
- The parsing table consists of two parts, a parsing action function action and a goto function goto.



- The program deriving the LR parser behaves as follows:
  - ➢ It determines S_m , the state currently on the top of the stack, and a_i, the current input symbol. It then consults action[S_m ,a_i] the parsing action table entry for state S_m and input a_i , which has one of four values:
    1. Shift S, where S is a state
    2. Reduce by a grammar production $A \rightarrow \beta$
    3. Accept and
    4. Error

- The function goto takes a state and grammar symbol as arguments and produces a state.
- Viable Prefix: The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes.

LR Parsing Algorithm

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

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

Method: Initially, the parser has S_0 on its stack, where S_0 is the initial state, and w$ in the input buffer.

The parser then executes the program until an accept or error action is encountered.

Algorithm:

```
Set ip to point to the first 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 S¹ then begin
            push a then S¹ 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 S¹ be the state now on top of the stack;
            push A then goto[S¹ ,A] on top of the stack;
            output the production A->β
        end
        else if action[S,a]=accept then
            return
        else
            error()
end
```

- A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input

  $(S_0\ X_1\ S_1\ X_2\ \ldots\ X_m\ S_m\ ,\ a_i\ ,\ a_{i+1},\ldots\ a_n\ \$)$

This configuration represents the right sentential form

$X_1 X_2 \ldots.. X_m a_i a_{i+1} \ldots. a_n$

- The next move of the parser is determined by reading a_i , the current input symbol, and S_m , the state on top of the stack, and then consulting the parsing action table entry action[S_m ,a_i]
- The configurations resulting after each of the four types of moves are as follows:
  1. if action[S_m ,a_i]=shift s, the parser executes a shift move, entering the configuration

     $(S_0\ X_1\ S_1\ X_2\ \ldots..\ X_m\ S_m\ ,\ a_i\ ,\ a_{i+1},\ldots.\ a_n\ \$)$
  2. if action[S_m ,a_i]=reduce A->β, the parser executes a reduce move, entering the configuration

     $(S_0\ X_1\ S_1\ X_2\ \ldots..\ X_{m-r}\ S_{m-r}\ A_s,\ a_i\ ,\ a_{i+1},\ldots.\ a_n\ \$)$

     where s=goto[S_{m-r} ,A] and r the length of β, the right side of production. Here the parser first popped 2r symbols off the stack (r state symbols and r grammar symbols)
  3. if action[S_m ,a_i]=accept, parsing is completed
  4. if action[S_m ,a_i]=error, the parser has discovered an error and calls an error recovery routine.
- All LR parsers behave in this fashion; the only difference between one LR parser and another is the information in the parsing action and goto fields of the parsing table.

Constructing SLR parsing table

- Simple LR or SLR is the weakest of all the parsers in terms of the number of grammars for which it succeeds
- It is easy to implement
- A grammar for which an SLR parser can be constructed is said to be an SLR grammar
- **LR(0) item:** An LR(0) item of a grammar G is a production of G with a dot at some position of the right side

  Thus production A->XYZ yields the four items

  A->.XYZ

  A->X.YZ

  A->XY.Z

  A->XYZ.
- The production A->� generates only one item, A->.

- An item can be represented by a pair of integers, the first giving the number of productions and the second the position of the dot

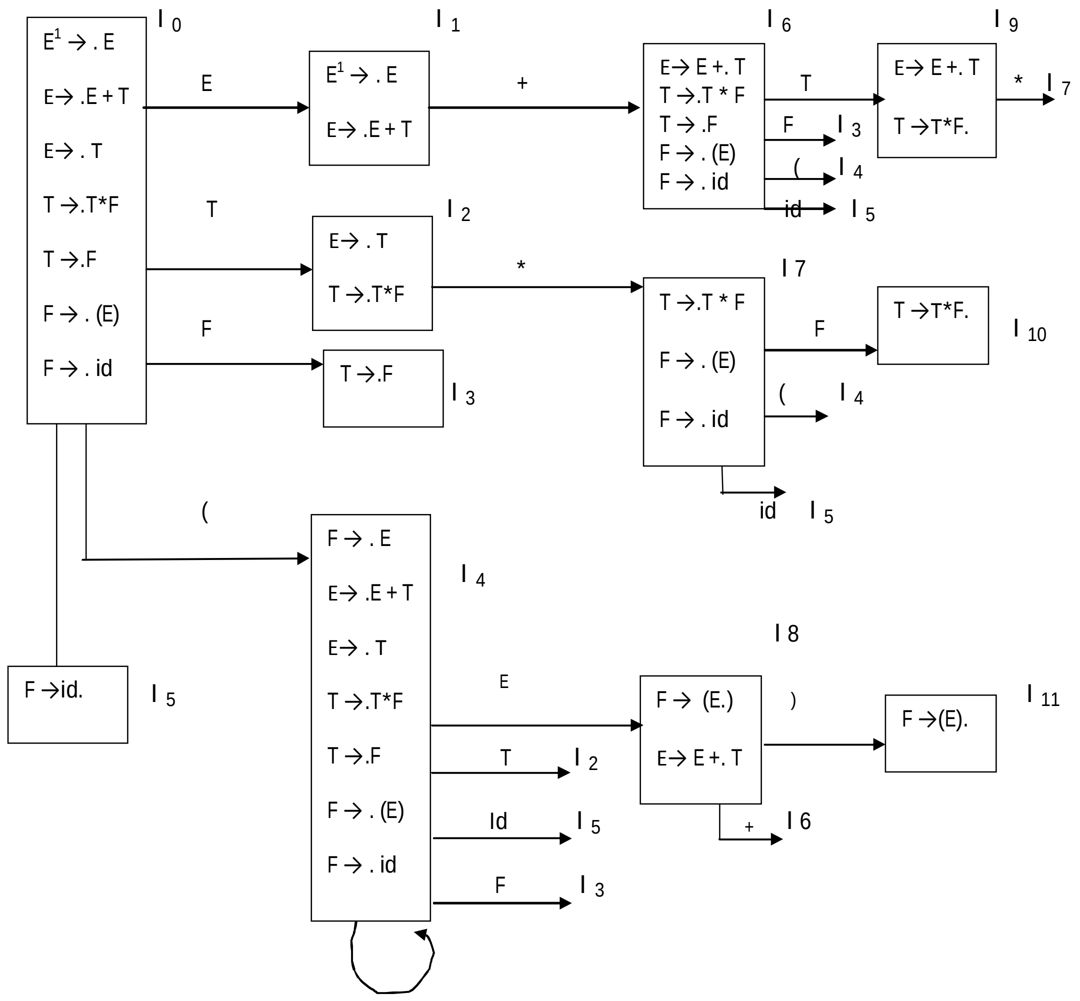
Example: Construct SLR parsing table for the grammar

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

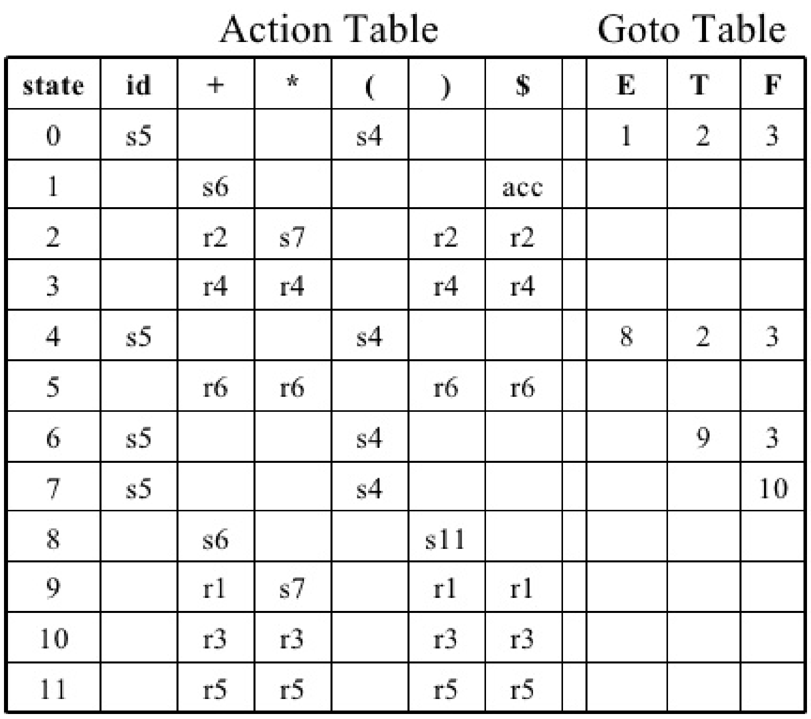
Step 1: Consider augmented grammar

E^1 ->E
E->E+T|T
T->T*F|F
F->(E)|id

Step 2: Construct Canonical LR(0) items



Step 3: Construct SLR parsing table for the grammar



	Action Table						Goto Table		
state	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			

Step 4: Moves of SLR parser on the input id * id + id

stack	input	action
0	id*id+id$	shift
0id5	*id+id$	Reduce by f->id
0F3	*id+id$	Reduce by T->F
0T2	*id+id$	Shift
0T2*7	id+id$	Shift
0T2*7id5	+id$	Reduce by F->id
0T2*7F10	+id$	Reduce by T->T*F
0T2	+id$	Reduce by E->T
0E1	+id$	Shift
0E1+6	id$	Shift
0E1+6id5	$	Reduce by F->id
0E1+6F3	$	Reduce by T->F
0E1+6T9	$	Reduce by E->E+T
0E1	$	accept

Constructing CLR Parsing Table:

Input: An augmented grammar G'.
Output: The canonical LR parsing table functions action and goto for G'

1. Construct C={I0,I1..........,In}, the collection of sets of LR(1) items for G'.
2. State I of the parser is constructed from Ii. The parsing actions for state I are determined as follows :
   a) If [A → α. a β, b] is in Ii, and goto(Ii, a) = Ij, then set action[i,a] to "shift j." Here, a is required to be a terminal.
   b) If [A → α., a] is in Ii, A ≠ S', then set action[i,a] to "reduce A → α."
   c) If [S'→S.,$] is in Ii, then set action[i ,$] to "accept."
   d) If a conflict results from above rules, the grammar is said not to be LR(1), and the algorithm is said to be fail.
3. The goto transition for state i are determined as follows: 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 containing item [S'→.S, $].

Example: Construct CLR parsing table for the grammar

S->CC

C->cC/d

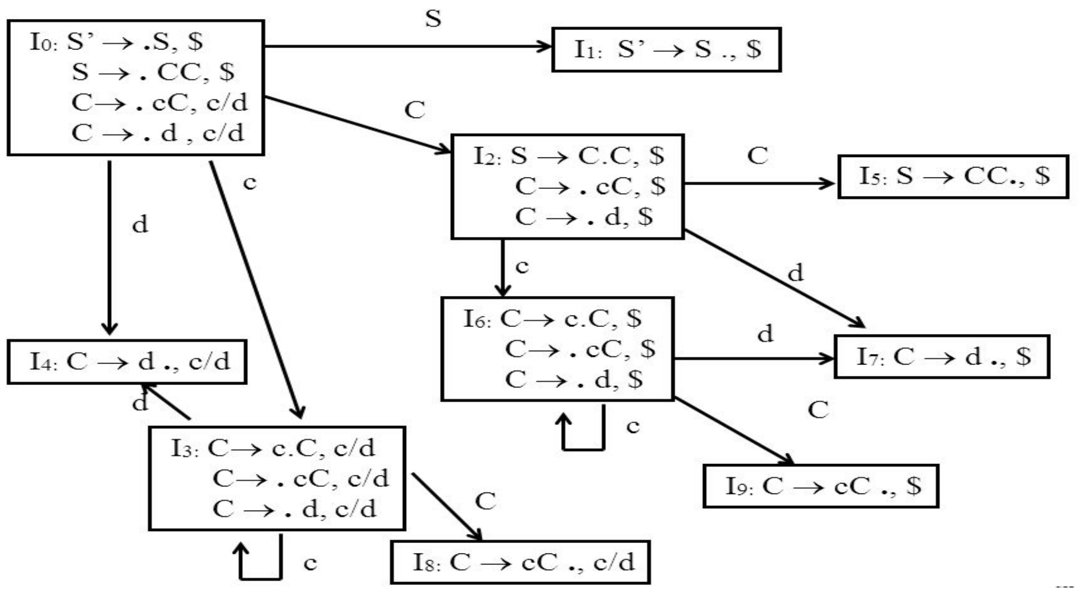
Step 1: The augmented grammar is

S'->S

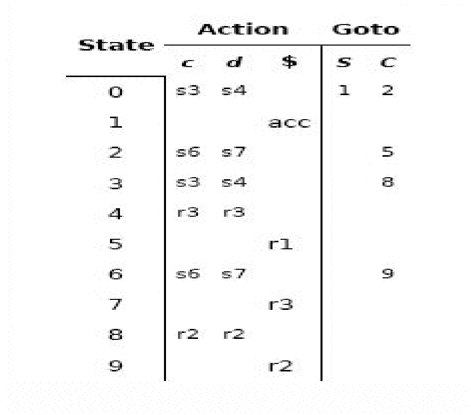
S->CC

C->cC/d

Step 2: Construct sets of LR(1) items for the given grammar



Step 3:- Construct canonical parsing table.



State	Action			Goto	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Step 4:- Moves of CLR parser for the input cdd

stack	input	action
0	cdd$	shift
0c3	dd$	shift
0c3d4	d$	Reduce by c->d
0c3c8	d$	Reduce by c->cc
0c2	d$	Shift
0c2d7	d$	Reduce by c->d
0C2C5	$	Reduce by S →Cc
0S1	$	Accept

Constructing LALR Parsing Table:

CORE: A core is a set of LR (0) (SLR) items for the grammar, and an LR (1) (Canonical LR) grammar may produce more than two sets of items with the same core.

The core does not contain any look ahead information.

Example: Let S1 and S2 are two states in a Canonical LR grammar.

S1 – {C ->c.C, c/d; C -> .cC, c/d; C -> .d, c/d}

S2 – {C ->c.C, $; C -> .cC, $; C -> .d, $}

These two states have the same core consisting of only the production rules without any look ahead information.

Algorithm:

Input: An augmented grammar G'.

Output: The LALR parsing table actions and goto for G'.

Method:

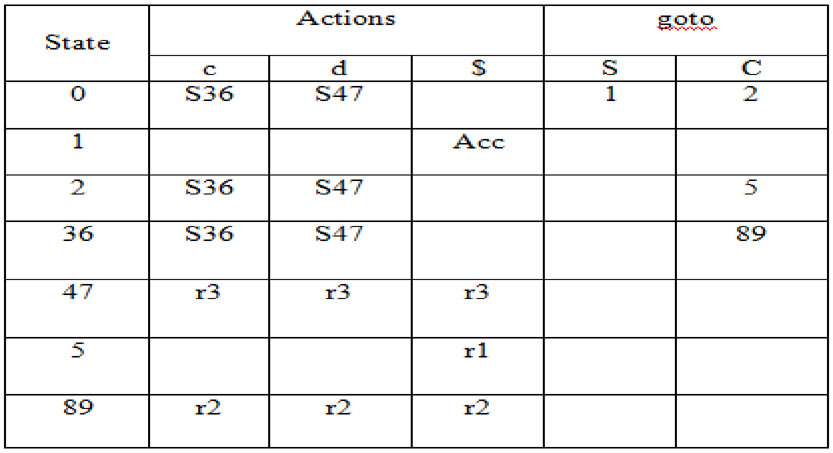
1. Construct C= {I0, I1, I2,… , In}, the collection of sets of LR(1) items.
2. For each core present in among these sets, find all sets having the core, and replace these sets by their union.
3. Parsing action table is constructed as for Canonical LR.
4. The goto table is constructed by taking the union of all sets of items having the same core. If J is the union of one or more sets of LR (1) items, that is, J=I1 U I2 U … U Ik, then the cores of goto(I1,X), goto(I2,X),…, goto(Ik, X) are the same as all of them have same core. Let K be the union of all sets of items having same core as goto(I1, X). Then goto(J,X)=K.

Example: Construct LALR parser for the grammar

$S \rightarrow CC$

$C \rightarrow cC / d$

1. Construct the set of LR (1) items.
   Consider the grammar whose set of LR(1) items were shown in CLR parser.
2. Merge the sets with common core together as one set, if no conflict (shift-shift or shift-reduce) arises.
   I47: $C \rightarrow d.,\ c/d/\$$
   I36: $C \rightarrow c.C,\ \ c/d/\$$
   $C \rightarrow .c.C,\ \ c/d/\$$
   $C \rightarrow .d,\ \ c/d/\$$
   I89: $C \rightarrow cC.,\ c/d/\$$
3. The parsing table is constructed from the collection of merged sets of items using the same algorithm for LR (1) parsing.



State	Actions			goto	
	c	d	$	S	C
0	S36	S47		1	2
1			Acc		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

4. Moves of LALR parser for the input cdd

State	Input	Action
0	cdd$	Shift
0c3	dd$	Shift
0c3d4	d$	reduce by C→d
0c3C6	d$	reduce by C→cC
0C2	d$	Shift
0C2d4	$	reduce by C→d
0C2C5	$	reduce by S→Cc
0S1	$	accept

UNIT-III
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. Which of the following derivations does a bottom-up parser use while parsing an input string? The input is assumed to be scanned in left to right order. [ ]
   a) Leftmost derivation
   b) Leftmost derivation traced out in reverse
   c) Rightmost derivation
   d) Rightmost derivation traced out in reverse
2. Which of the following describes a handle (LR-parsing) appropriately? [ ]
   a. It is the position in a sentential form where the next shift or reduce operation will occur
   b. It is non-terminal whose production will be used for reduction in the next step
   c. It is a production that may be used for reduction in a future step along with a position in the sentential form where the next shift or reduce operation will occur
   d. It is the production p that will be used for reduction in the next step along with a position in the sentential form where the right hand side of the production may be found
3. Shift reduce parsing belongs to a class of [ ]
   a) bottom up parsing b. top down parsing
   c. recursive parsing d. predictive parsing
4. Which one of the following is TRUE at any valid state in shift – reduce parsing?
   a. Viable prefixes appear only at the bottom of the stack and not inside
   b. Viable prefixes appear only at the top of the stack and not inside
   c. The stack contains only a set of viable prefixes [ ]
   d. The stack never contains viable prefixes
5. LALR grammars are subset of LR(1). [True/False]
6. Which of the following is the most powerful parsing method? [ ]
   a) SLR b. LALR c. CLR d. LL(1)
7. What is the precedence between '('and ')'? [ ]
   a. <· b. = c. ·> d. blank
8. Operator precedence parser is ________ type of parser.
9. Operator precedence parser can parse ambiguous grammars. [True/False]
10. YACC builds up ____________ parsing table [ ]
    a. SLR b. LALR c. CLR d. LL(1)
11. For the grammar S$\rightarrow$ SS+ | SS* | a , identify the handle for the first reduction step for the string "aa*a+" [ ]

a. a b. SS+ c. SS* d. S+S

12. Consider the grammar S→ +SS | *SS | a. [ ]
    For a sentence +*aaa, the handles in the right-sentential form of the reduction are
    a. a, *SS, +SS, a and a b. a, +SS, *SS, a and a
    c. a, +*Saa, +*SSa, a and a d. a, a, *SS, a and +SS
13. Let the number of states in SLR(1), LR(1) and LALR(1) parsers for the grammar be n1, n2 and n3 respectively. The following relationship holds good [ ]
    a. n1 < n2 < n3 b. n1 = n3 < n2 c. n1 = n2 = n3 d. n1 ≥ n3 ≥ n2
14. Consider the following grammar.
    S → dA | aB
    A → bA | c
    B → bB | c
    Consider the following LR(0) items corresponding to the grammar above.
    (i) S → d.A (ii) A→ .bA (iii) A→ .c
    Given the items above, which of them will appear in the same set in the canonical sets-of-items for the grammar? [ ]
    a. (i) and (ii) b. (ii) and (iii) c. (i) and (iii) d. (i), (ii) and (iii)
15. What is the maximum number of reduce moves that can be taken by a bottom-up parser for a grammar with no epsilon- and unit-production (i.e., of type A → ε and A → a) to parse a string with n tokens? [ ]
    a. n/2 b. n-1 c. 2n-1 d. 2n
16. The grammar S→aSa | bS | c is? [ ]
    a. LL(1) but not LR(1) b. LR(1) but not LL(1)
    c. Both LL(1) and LR(1) d. Neither LL(1) nor LR(1)
17. Consider SLR(1) and LALR(1) tables for CFG. Which of the following is false?
    a. Goto of both tables may be different [ ]
    b. Shift entries are identical in both tables
    c. Reduce entries in tables may be different
    d. Error entries may be different

SECTION-B

Descriptive Questions

1. What is Bottom-up parsing?
2. Define Handle with an example.
3. Explain Shift-Reduce parser. What are its drawbacks?
4. Why LR parsing is good and attractive?
5. What is an operator grammar? Explain with an example.
6. Define LR grammar. Explain the model of LR parser.
7. Write the steps involved for construction of SLR parser.

8. Construct Shift-Reduce parser for the following grammar

   $S \rightarrow aABb$
   $A \rightarrow c \mid aB$
   $B \rightarrow d \mid bA$

9. Identify whether the given grammar is LL (1) or SLR (1).

   $S \rightarrow AaAb \mid BbBa$
   $A \rightarrow \varepsilon$
   $B \rightarrow \varepsilon$

10. Apply SLR parsing technique for the following grammar

    $S \rightarrow SA \mid A$
    $A \rightarrow a$

11. Construct SLR parser for the following grammar

    $S \rightarrow SS+ \mid SS* \mid a$

12. Construct CLR parser for the following grammar

    $S \rightarrow SS+ \mid SS* \mid a$

13. Construct LALR parser for the following grammar

    $S \rightarrow SS+ \mid SS* \mid a$

14. Construct LALR parser for the following grammar

    $S \rightarrow SA \mid A$
    $A \rightarrow a$

15. Construct Operator precedence parsing table for the following grammar

    $P \rightarrow SbP \mid S \mid SbS$
    $S \rightarrow WbS \mid W$
    $W \rightarrow L*W \mid L$
    $L \rightarrow id$

SECTION-C

Gate Questions

1. Among simple LR (SLR), canonical LR, and look – ahead LR (LALR), which of the following pairs identify the method that is very easy to implement and the method that is the most powerful, in that order? [ ] **(GATE CS 2015)**

   a. SLR, LALR b. Canonical LR, LALR
   c. SLR, canonical LR d. LALR, canonical LR

2. An LALR(1) parser for a grammar G can have shift-reduce (S-R) conflicts if and only if [ ] **(GATE CS 2015)**

   a. the SLR(1) parser for G has S-R conflicts

b. the LR(1) parser for G has S-R conflicts
c. the LR(0) parser for G has S-R conflicts
d. the LALR(1) parser for G has reduce-reduce conflicts

3. Consider the following grammar G [ ] **(GATE CS 2015)**

$S \rightarrow F \mid H$

$F \rightarrow p \mid c$

$H \rightarrow d \mid c$

Where S, F, and H are non – terminal symbols, p, d, and c are terminal symbols. Which of the following statements (s) is/are correct?

S1: LL(1) can parse all strings that are generated using grammar G

S2: LR(1) can parse all strings that are generated using grammar G

a. only S1 b. only S2
c. Both S1 and S2 d. Neither S1 nor S2

4. A canonical set of items is given below [ ] **(GATE CS 2014)**

$S \rightarrow L. > R$

$Q \rightarrow R.$

On input symbol < the set has

a. a shift-reduce conflict and a reduce-reduce conflict.
b. a shift-reduce conflict but not a reduce-reduce conflict.
c. a reduce-reduce conflict but not a shift-reduce conflict.
d. neither a shift-reduce nor a reduce-reduce conflict.

5. An LALR(1) parser for a grammar G can have shift-reduce (S-R) conflicts if and only if

a. the SLR(1) parser for G has S-R conflicts [ ] **(GATE CS 2008)**
b. the LR(1) parser for G has S-R conflicts
c. the LR(0) parser for G has S-R conflicts
d. the LALR(1) parser for G has reduce-reduce conflicts

6. Which of the following grammar rules violate the requirements of an operator grammar? P, Q, R are non terminals, and r,s,t are terminals [ ] **(GATE CS 2004)**

a. $P \rightarrow QR$ b. $P \rightarrow QsR$
c. $P \rightarrow \varepsilon$ d. $P \rightarrow QtRr$

7. Which of the following statements is false? [ ] **(GATE CS 2001)**

a. An unambiguous grammar has same leftmost and rightmost derivation
b. An LL(1) parser is a top-down parser
c. LALR is more powerful than SLR
d. An ambiguous grammar can never be LR(k) for any k.