CC EXPERIMENT NO. 05

Shubham Golwal | 2020300015 | TE COMPS

Aim:

Implement LR(0) parser for Given Grammar.

(1) $E \rightarrow E$ **sub** E **sup** E
(2) $E \rightarrow E$ **sub** E
(3) $E \rightarrow E$ **sup** E
(4) $E \rightarrow \{ E \}$
(5) $E \rightarrow c$

Theory:

The LR parser is an efficient bottom up syntax analysis technique that can be used to large class of context-free grammar.This technique is also called LR(0) parsing.

- L stands for left to right scanning
- R stands for rightmost derivation in reverse • 0 stands for no. of input symbols of lookahead.

Augmented grammar:

If P is a grammar with starting symbol S,then G' (augmented grammar for G) is a grammar with a new starting symbol S' and productions S-> .S' . The purpose of this new starting production is to indicate the parser when it should stop parsing. The ' . ' before S indicates the left side of ' . ' has been read by a compiler and the right side of ' . ' is yet to be read by a compiler.

Steps for constructing the LR parsing table :

1. Writing augmented grammar
2. LR(0) collection of items to be found
3. Defining 2 functions: goto(list of terminals) and action(list of non-terminals) in the parsing table.

$I_0 \rightarrow$ $E' \rightarrow .E$
$E \rightarrow .E \text{ sub } E \text{ sup } E$
$E \rightarrow .E \text{ sub } E$
$E \rightarrow .E \text{ sup } E$
$E \rightarrow .\{E\}$
$E \rightarrow .c$

$I_1 :$ $E' \rightarrow E.$
$E \rightarrow E. \text{ sub } E \text{ sup } E$
$E \rightarrow E. \text{ sub } E$
$E \rightarrow E. \text{ sup } E$

$I_2 :$ $E \rightarrow \{.E\}$
$E \rightarrow .E \text{ sub } E \text{ sup } E$
$E \rightarrow .E \text{ sub } E$
$E \rightarrow .E \text{ sup } E$
$E \rightarrow .\{E\}$
$E \rightarrow .c$

$I_3 :$ $E \rightarrow c.$

$I_4 :$ $E \rightarrow E \text{ sub}. E \text{ sup } E$
$E \rightarrow E \text{ sub}. E$
$E \rightarrow .E \text{ sub } E \text{ sup } E$
$E \rightarrow .E \text{ sub } E$
$E \rightarrow .E \text{ sup } E$
$E \rightarrow .\{E\}$
$E \rightarrow .c$

I_8 : E → E sup . E
E → . E sub E sup E
E → . E sub E
E → . E sup E
E → . { E }
E → . c

I_9 : E → E . sub E sup E
E → E . sub E
E → E . sup E
E → { E . }

I_4 : E → E . sub E sup E
E → E sub E . sup E
E → E . sup E
E → E sub E .
E → E . sup E

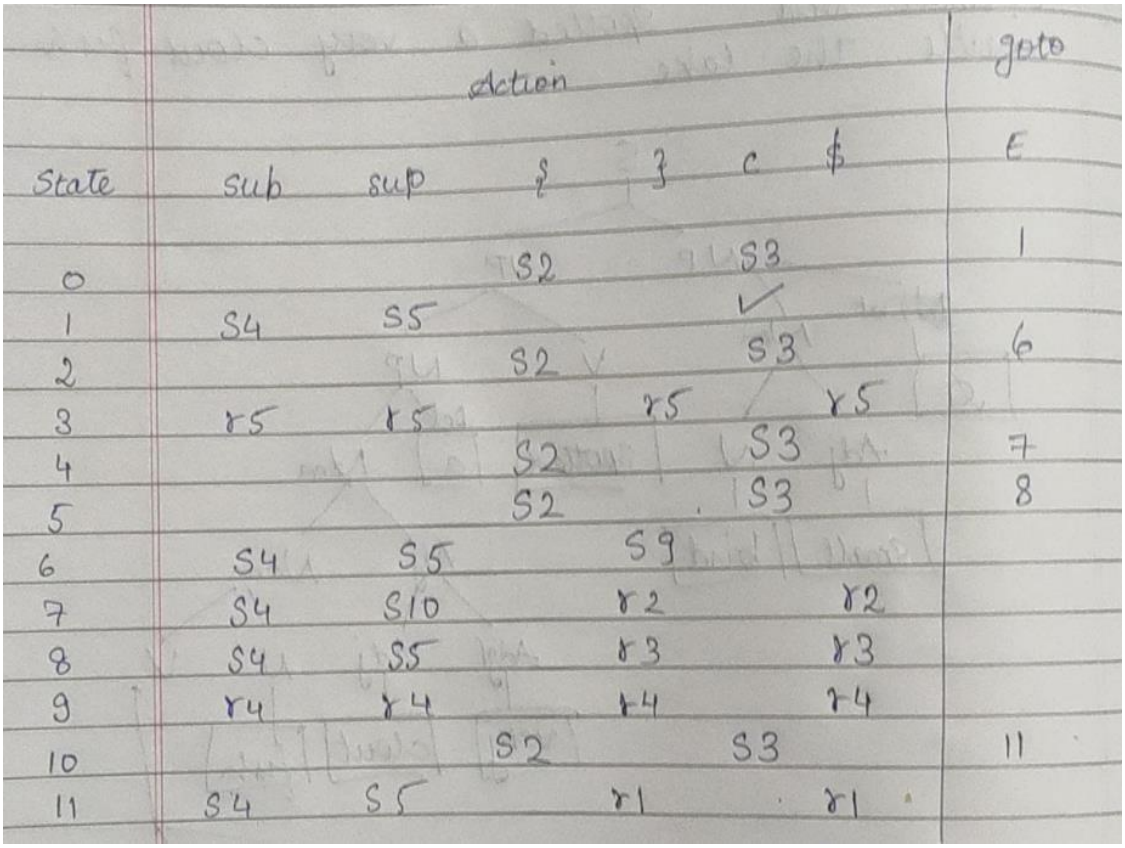
I_5 : E → E . sub E sup E
E → E . sub E
E → E . sup E
E → E sup E .

I_9 : { E } .

I_{10} : E → E sub E sup . E
E → E sup . E
E → . E sub E sup E , . E sub E , . E sup E , . { E } .

I_{11} : E → E . sub E sup E
E → E sub E sup E .
E → E . sub E , E sub E . , E . sup E

Parsing Table:



State	Action						Goto
	sub	sup	{	}	c	$	E
0			S2		S3		1
1	S4	S5				✓	
2			S2		S3		6
3	r5	r5		r5		r5	
4			S2		S3		7
5			S2		S3		8
6	S4	S5		S9			
7	S4	S10		r2		r2	
8	S4	S5		r3		r3	
9	r4	r4		r4		r4	
10			S2		S3		11
11	S4	S5		r1		r1	

Implementation Code:

```
import java.util.*;
import java.util.stream.*;

public class Parser {
    private final List<List<String>> production = Arrays.asList(
            Arrays.asList("E", "sub", "E", "sup", "E"),
            Arrays.asList("E", "sub", "E"),
            Arrays.asList("E", "sup", "E"),
            Arrays.asList("E", "{", "E", "}"),
            Arrays.asList("E", "c")
    );

    private final Integer[][] table = new Integer[][]{
            {2, null, null, null, 3, null, 1},
            {null, null, 5, 4, null, -1, null},
            {2, null, null, null, 3, null, 6},
            {null, 5, 5, 5, null, 5, -1},
            {2, null, null, null, 3, null, 7},
```

```
            {2, null, null, null, 3, null, 8},
            {null, 9, 5, 4, null, null, -1},
            {null, 2, 4, 10, null, 2, -1},
            {null, 3, 5, 4, null, 3, -1},
            {null, 4, 4, 4, null, 4, -1},
            {2, null, null, null, 3, null, 11},
            {null, 1, 5, 4, null, 1, -1}
    };

    private final Map<String, Integer> terminals = new HashMap<String, Integer>()
{{
        put("{", 0);
        put("}", 1);
        put("sup", 2);
        put("sub", 3);
        put("c", 4);
        put("$", 5);
    }};

    private final Map<String, Integer> nonTerminals = new HashMap<String,
Integer>() {{
        put("S", 0);
        put("E", 1);
    }};

    public boolean parse(String input) {
        List<String> tokens = Arrays.stream(input.split("
")).collect(Collectors.toList());
        tokens.add("$");

        Deque<Integer> stack = new ArrayDeque<>();
        stack.push(0);
        int ptr = 0;
        boolean error = false;

        while (true) {
            System.out.print("Stack: ");
            System.out.println(stack);

            int sd = stack.peek();
            int bd = terminals.get(tokens.get(ptr));
            Integer ptd = table[sd][bd];

            // Error Occurence
            if (ptd == null) {
```

```
                error = true;
                break;
            }

            // String accepted
            if (ptd == -1) {
                break;
            }

            // REDUCE Case
            if (ptd < 0) {
                List<String> prod = production.get(-ptd - 1);
                int l = prod.get(1).equals("") ? 1 : prod.size();

                if (stack.size() < 2 * l) {
                    error = true;
                    break;
                }

                for (int i = 0; i < 2 * l; i++) {
                    stack.pop();
                }

                sd = stack.peek();
                bd = nonTerminals.get(prod.get(0));

                if (table[sd][bd] == null) {
                    error = true;
                    break;
                }

                stack.push(bd);
                stack.push(table[sd][bd]);
            }

            // SHIFT Case

            else {
                stack.push(ptd);
                ptr++;
            }
        }

        if (error) {
            System.out.println("Error: Invalid input");
```

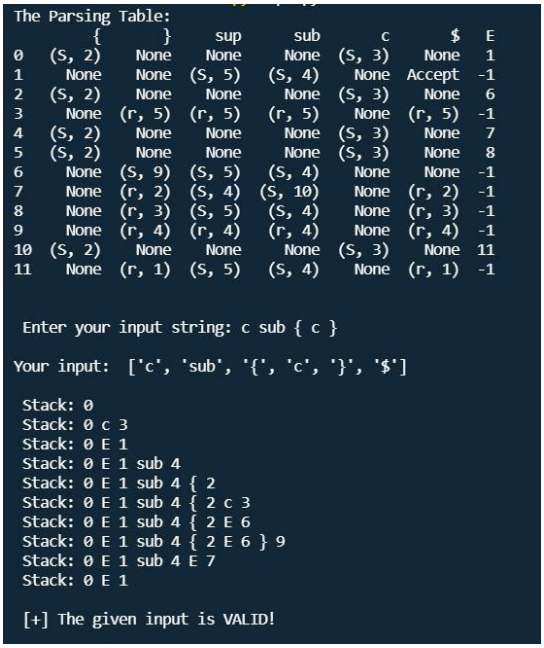
```
            return false;
        } else {
            System.out.println("Success: Valid input");
            return true;
        }
    }

    public static void main(String[] args) {
        Parser parser = new Parser();

        // Test input strings
        String input1 = "c";
        String input2 = "{ c }";
        String input3 = "{ sub c }";
        String input4 = "{ c sup c }";
        String input5 = "{ sub c sup c }";
        String input6 = "{ sub c } sup c";
        String input7 = "{ sub { c sup c } }";
        String input8 = "{ sub { c sub c } sup { c sub c } }";

        parser.parse(input1);
        parser.parse(input2);
        parser.parse(input3);
        parser.parse(input4);
        parser.parse(input5);
        parser.parse(input6);
        parser.parse(input7);
        parser.parse(input8);
    }
}
```

Result:



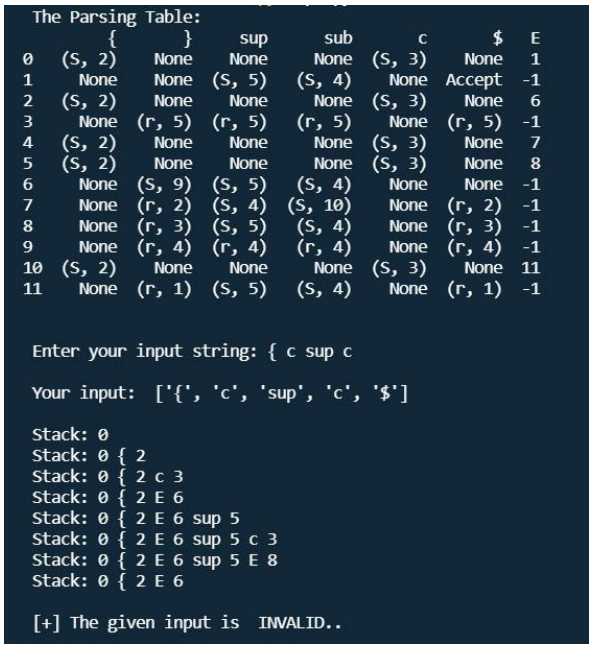
```
The Parsing Table:
        {       }     sup      sub       c        $   E
0   (S, 2)    None    None     None  (S, 3)    None   1
1     None    None  (S, 5)   (S, 4)    None  Accept  -1
2   (S, 2)    None    None     None  (S, 3)    None   6
3     None  (r, 5)  (r, 5)   (r, 5)    None  (r, 5)  -1
4   (S, 2)    None    None     None  (S, 3)    None   7
5   (S, 2)    None    None     None  (S, 3)    None   8
6     None  (S, 9)  (S, 5)   (S, 4)    None    None  -1
7     None  (r, 2)  (S, 4)  (S, 10)    None  (r, 2)  -1
8     None  (r, 3)  (S, 5)   (S, 4)    None  (r, 3)  -1
9     None  (r, 4)  (r, 4)   (r, 4)    None  (r, 4)  -1
10  (S, 2)    None    None     None  (S, 3)    None  11
11    None  (r, 1)  (S, 5)   (S, 4)    None  (r, 1)  -1


 Enter your input string: c sub { c }

Your input:  ['c', 'sub', '{', 'c', '}', '$']

 Stack: 0
 Stack: 0 c 3
 Stack: 0 E 1
 Stack: 0 E 1 sub 4
 Stack: 0 E 1 sub 4 { 2
 Stack: 0 E 1 sub 4 { 2 c 3
 Stack: 0 E 1 sub 4 { 2 E 6
 Stack: 0 E 1 sub 4 { 2 E 6 } 9
 Stack: 0 E 1 sub 4 E 7
 Stack: 0 E 1

 [+] The given input is VALID!
```



```
The Parsing Table:
        {       }     sup      sub       c        $   E
0   (S, 2)    None    None     None  (S, 3)    None   1
1     None    None  (S, 5)   (S, 4)    None  Accept  -1
2   (S, 2)    None    None     None  (S, 3)    None   6
3     None  (r, 5)  (r, 5)   (r, 5)    None  (r, 5)  -1
4   (S, 2)    None    None     None  (S, 3)    None   7
5   (S, 2)    None    None     None  (S, 3)    None   8
6     None  (S, 9)  (S, 5)   (S, 4)    None    None  -1
7     None  (r, 2)  (S, 4)  (S, 10)    None  (r, 2)  -1
8     None  (r, 3)  (S, 5)   (S, 4)    None  (r, 3)  -1
9     None  (r, 4)  (r, 4)   (r, 4)    None  (r, 4)  -1
10  (S, 2)    None    None     None  (S, 3)    None  11
11    None  (r, 1)  (S, 5)   (S, 4)    None  (r, 1)  -1


 Enter your input string: { c sup c

 Your input:  ['{', 'c', 'sup', 'c', '$']

 Stack: 0
 Stack: 0 { 2
 Stack: 0 { 2 c 3
 Stack: 0 { 2 E 6
 Stack: 0 { 2 E 6 sup 5
 Stack: 0 { 2 E 6 sup 5 c 3
 Stack: 0 { 2 E 6 sup 5 E 8
 Stack: 0 { 2 E 6

 [+] The given input is  INVALID..
```

Conclusion:

From the above experiment, I was able to implement code and programmatically execute and verify the working of LR(0) parser by manually finding the parsing table for a given grammar.