Name: Adwait Hegde
Roll No: 2019130019
TE Comp (Batch-A)

EXPERIMENT 4

Aim:

Implement LR(0) parser for Given Grammar.

(1) $E \rightarrow E \text{ sub } E \text{ sup } E$
(2) $E \rightarrow E \text{ sub } E$
(3) $E \rightarrow E \text{ sup } E$
(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.

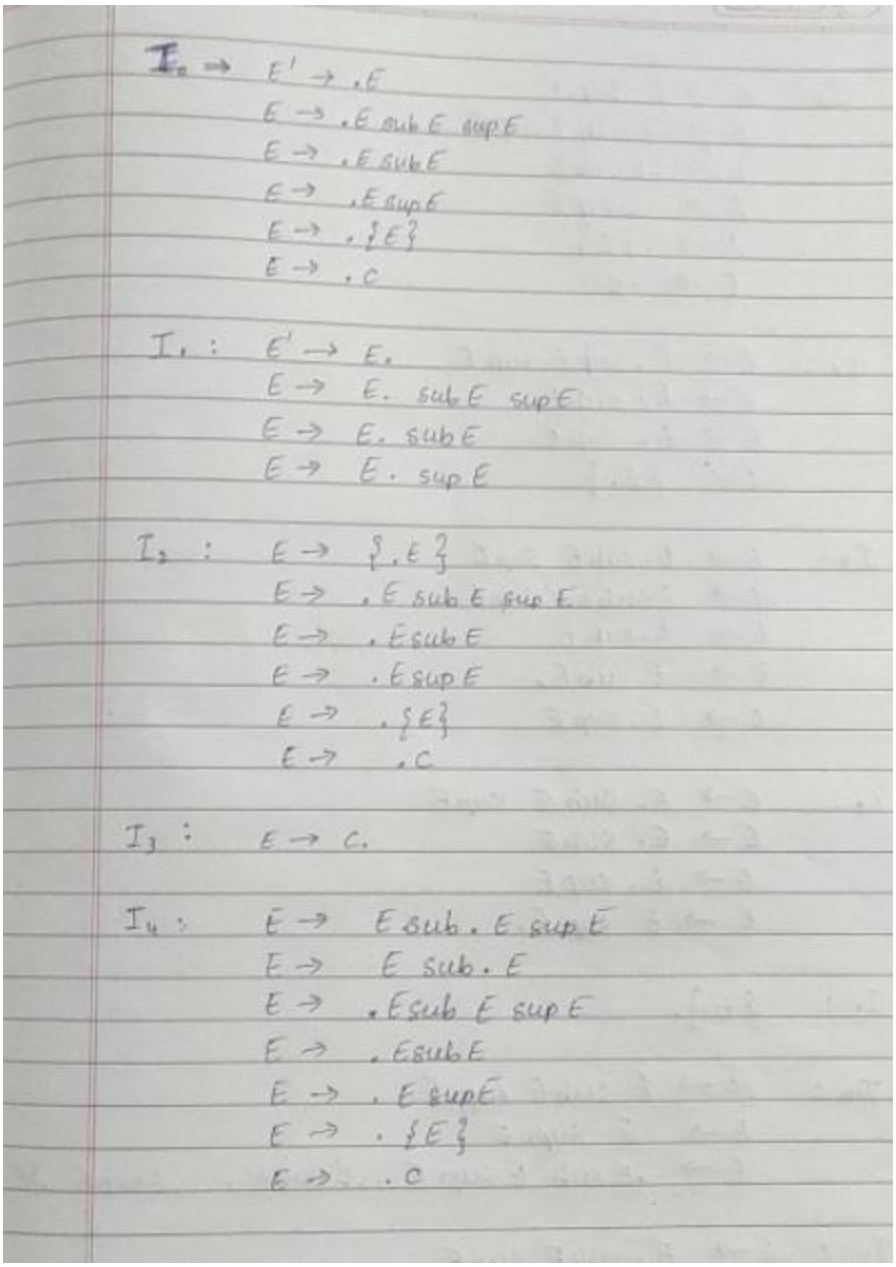
- L stands for left to right scanning
- R stands for rightmost derivation in reverse
- 0 stands for no. of input symbols of lookahead.

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.



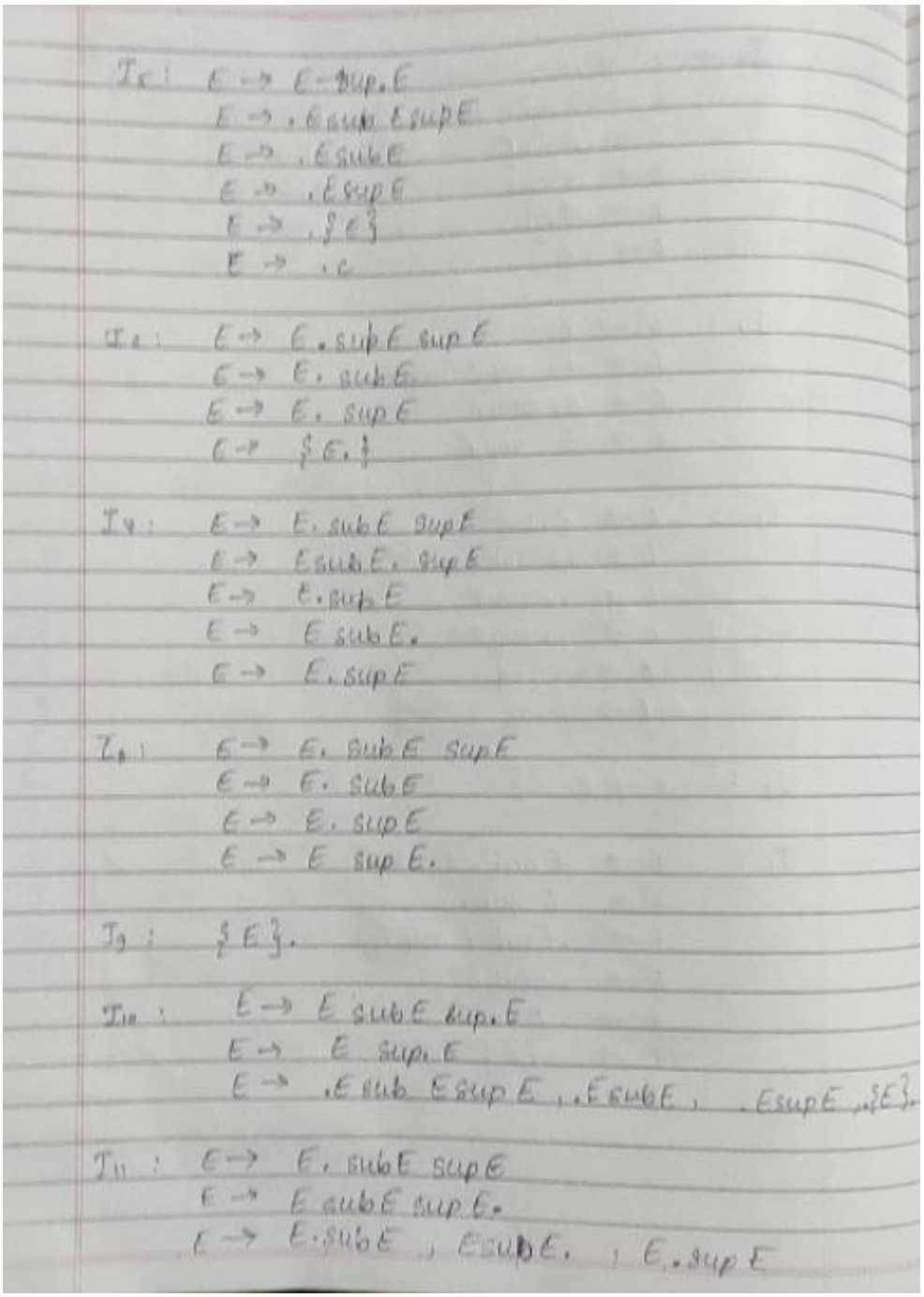
$I_0 \rightarrow E' \rightarrow .E$
$E \rightarrow .E \text{ sub } E \text{ sup } E$
$E \rightarrow .E \text{ sub } E$
$E \rightarrow .E \text{ sup } E$
$E \rightarrow .\{E\}$
$E \rightarrow .c$

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

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

$I_3 : E \rightarrow c.$

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



I_8 : $E \rightarrow E \text{ sup } . E$
$E \rightarrow . E \text{ sub } E \text{ sup } E$
$E \rightarrow . E \text{ sub } E$
$E \rightarrow . E \text{ sup } E$
$E \rightarrow . \{ E \}$
$E \rightarrow . c$

I_9 : $E \rightarrow E . \text{sub } E \text{ sup } E$
$E \rightarrow E . \text{sub } E$
$E \rightarrow E . \text{sup } E$
$E \rightarrow \{ E . \}$

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

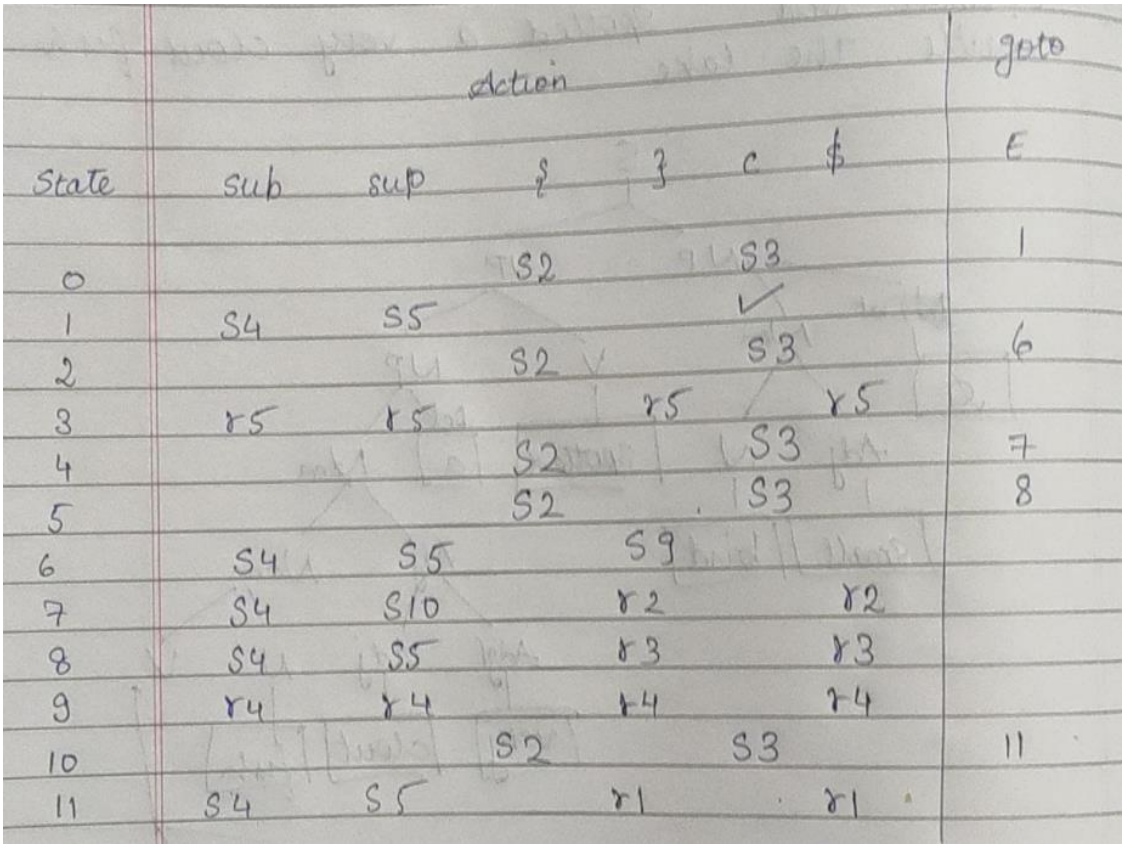
I_5 : $E \rightarrow E . \text{sub } E \text{ sup } E$
$E \rightarrow E . \text{sub } E$
$E \rightarrow E . \text{sup } E$
$E \rightarrow E \text{ sup } E .$

I_6 : $\{ E \} .$

I_{10} : $E \rightarrow E \text{ sub } E \text{ sup } . E$
$E \rightarrow E \text{ sup } . E$
$E \rightarrow . E \text{ sub } E \text{ sup } E \; , . E \text{ sub } E \; , . E \text{ sup } E \; , . \{ E \} .$

I_{11} : $E \rightarrow E . \text{sub } E \text{ sup } E$
$E \rightarrow E \text{ sub } E \text{ sup } E .$
$E \rightarrow E . \text{sub } E \; , E \text{ sub } E . \; , E . \text{sup } E$

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 pandas as pd

production = [
    ('E', ['E','sub','E','sup','E']),
    ('E', ['E','sub','E']),
    ('E', ['E','sup','E']),
    ('E', ['{','E','}']),
    ('E', ['c']),
]

df = [
    [ ('S',2), None, None, None, ('S',3), None, 1 ],
    [ None, None, ('S',5), ('S',4), None, 'Accept', -1 ],
    [ ('S',2), None, None, None, ('S',3), None, 6 ],
    [ None, ('r',5), ('r',5), ('r',5), None, ('r',5), -1 ],
    [ ('S',2), None, None, None, ('S',3), None, 7 ],
    [ ('S',2), None, None, None, ('S',3), None, 8 ],
    [ None, ('S',9), ('S',5), ('S',4), None, None, -1 ],
```

```
    [ None, ('r',2), ('S',4), ('S',10), None, ('r',2), -1 ],
    [ None, ('r',3), ('S',5), ('S',4), None, ('r',3), -1 ],
    [ None, ('r',4), ('r',4), ('r',4), None, ('r',4), -1 ],
    [ ('S',2), None, None, None, ('S',3), None, 11 ],
    [ None, ('r',1), ('S',5), ('S',4), None, ('r',1), -1 ],
]
df = pd.DataFrame(data=df, columns=['{', '}', 'sup', 'sub', 'c', '$', 'E'])
print(" The Parsing Table:")
print(df)

# Taking input string from the user:
inp = input("\n\n Enter your input string: ").split(' ')
inp.append('$')
print()
print(" Your input: ", inp)
print()

stack = [0]
ptr = 0

error = False
while True:
    print(" Stack: ", end='')
    print(*stack, sep=' ')

    sd = stack[-1]          # stack data
    bd = inp[ptr]           # buffer data
    ptd = df.loc[sd][bd]   # parsing table data

    # Error Occurence
    if ptd == None:
        error = True
        break

    # String accepted
    elif ptd=='Accept':
        break

    # REDUCE Case
    elif ptd[0]=='r':
        prod = production[ptd[1]-1]
        l = len(prod[1])
        if len(stack)<2*l:
            error = True
            break
```

```
        for _ in range(2*l):
            stack.pop()
        sd = stack[-1]
        bd = prod[0]
        if df.loc[sd][bd] ==-1:
            error = True
            break
        stack.append(bd)
        stack.append(df.loc[sd][bd])

    # SHIFT Case
    elif ptd[0]=='S':
        stack.append(bd)
        stack.append(ptd[1])
        ptr+=1


if not error:
    print("\n [+] The given input is VALID!\n")
else:
    print("\n [+] The given input is  INVALID..\n")
```

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.