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

EXPERIMENT 2

Aim:

To implement LL(1) parser in python.

Theory:

LL(1) Parsing:

Here the 1st **L** represents that the scanning of the Input will be done from Left to Right manner and the second **L** shows that in this parsing technique we are going to use Left most Derivation Tree. And finally, the **1** represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Algorithm to construct LL(1) Parsing Table:

Step 1: First check for left recursion in the grammar, if there is left recursion in the grammar remove that and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

First(): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.
Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha)
Find First(α) and for each terminal in First(α), make entry A –> α in the table.

If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A –> α in the table.
If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> α in the table for the $.

To construct the parsing table, we have two functions:
In the table, rows will contain the Non-Terminals and the column will contain the Terminal Symbols. All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of the First set.

Implementation

Grammar:

$S \rightarrow F$

$S \rightarrow (S\ T)$

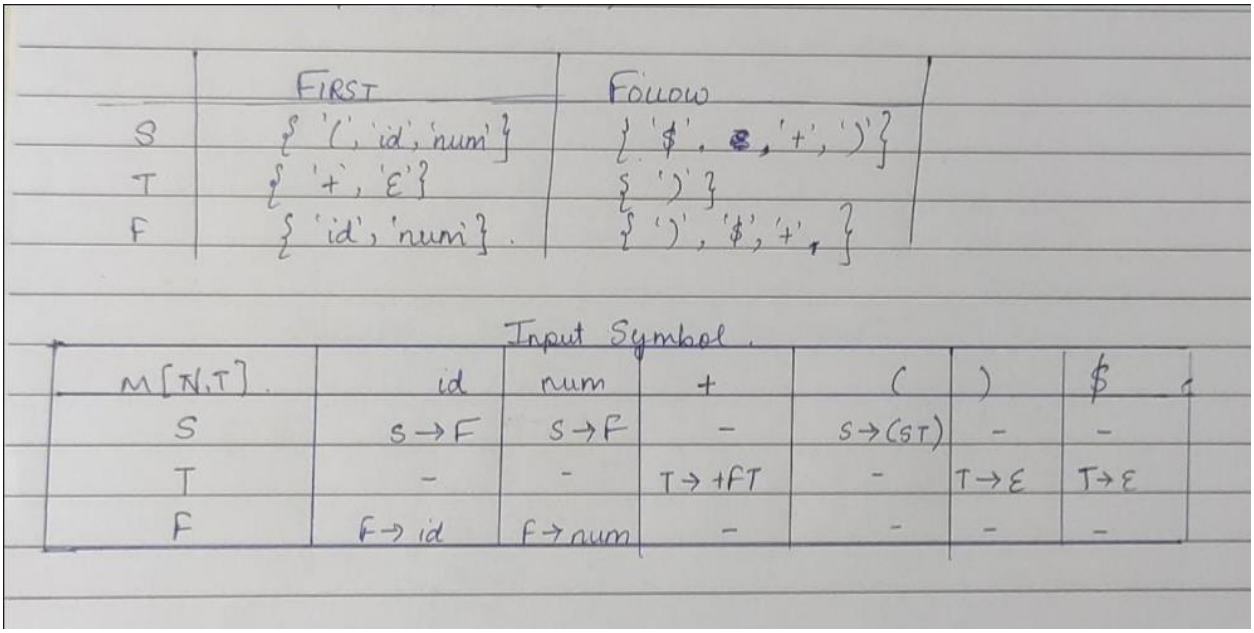
$T \rightarrow \varepsilon$

$T \rightarrow +\ F\ T$

$F \rightarrow id$

$F \rightarrow num$

Calculation:



	FIRST	FOLLOW
S	{ '(', id, num }	{ $,), + }
T	{ +, ε }	{) }
F	{ id, num }	{), $, + }

Input Symbol

M[N,T]	id	num	+	(	)	$
S	S→F	S→F	-	S→(ST)	-	-
T	-	-	T→+FT	-	T→ε	T→ε
F	F→id	F→num	-	-	-	-

Code:

```
import pandas as pd

# Input Symbol calculated for the productions
#  S -> F | (ST)
#  T -> +FT | e
#  F -> i | n

df = [
    [ ['S', ['F']], None, ['F', ['i']] ],
    [ ['S', ['F']], None, ['F', ['n']] ],
    [ None, ['T', ['+','F','T']], None ],
    [ ['S', ['(','S','T',')']], None, None ],
    [ None, ['T', []], None ],
    [ None, ['T', []], None ],
```

```
]
df = pd.DataFrame(data=df, columns=['S', 'T', 'F'],
index=['i','n','+','(',')','$'])
df = df.T
print("The Input Symbol:")
print(df)

# Taking input string from the user:
inp = input("\n\n Enter your input string: ")
inp = [i for i in inp+'$']
inp = inp[::-1]
print()
print("Your input: ", inp)
print()

terminals = ['i','+','n','(',')','$']
stack = ['$', 'S']


while len(stack)>0 and len(inp)>0:
    print("STACK:",stack)
    print("INPUT:",inp)
    print()
    top = stack.pop()
    if top in terminals:
        if top != inp.pop():
            break
        else:
            continue
    next = df.loc[top][inp[-1]]
    if next:
        if next[0]==top:
            stack += next[1][::-1]
        else:
            break
    else:
        break

if inp==stack==[] :
    print(" [+] The given input is VALID!")
else:
    print(" [+] The given input is  INVALID..")
```

Output:

```
PS D:\SPIT\SEM 6\CC Lab> py exp2_1.py
The Input Symbol:
          i         n              +                    (         )         $
S  [S, [F]]  [S, [F]]           None  [S, [(, S, T, )]]      None      None
T      None      None  [T, [+, F, T]]              None  [T, []]  [T, []]
F  [F, [i]]  [F, [n]]           None              None      None      None


 Enter your input string: (i(n)(n)i)

Your input:  ['$', ')', 'i', ')', 'n', '(', ')', 'n', '(', 'i', '(']

STACK: ['$', 'S']
INPUT: ['$', ')', 'i', ')', 'n', '(', ')', 'n', '(', 'i', '(']

STACK: ['$', ')', 'T', 'S', '(']
INPUT: ['$', ')', 'i', ')', 'n', '(', ')', 'n', '(', 'i', '(']

STACK: ['$', ')', 'T', 'S']
INPUT: ['$', ')', 'i', ')', 'n', '(', ')', 'n', '(', 'i']

STACK: ['$', ')', 'T', 'F']
INPUT: ['$', ')', 'i', ')', 'n', '(', ')', 'n', '(', 'i']

STACK: ['$', ')', 'T', 'i']
INPUT: ['$', ')', 'i', ')', 'n', '(', ')', 'n', '(', 'i']

STACK: ['$', ')', 'T']
INPUT: ['$', ')', 'i', ')', 'n', '(', ')', 'n', '(']

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

```
The Input Symbol:
          i        n              +                (        )       $
S  [S, [F]]  [S, [F]]           None  [S, [(, S, T, )]]     None     None
T      None      None  [T, [+, F, T]]              None  [T, []]  [T, []]
F  [F, [i]]  [F, [n]]           None              None     None     None


 Enter your input string: ((n)+i)

Your input:  ['$', ')', 'i', '+', ')', 'n', '(', '(']

STACK: ['$', 'S']
INPUT: ['$', ')', 'i', '+', ')', 'n', '(', '(']

STACK: ['$', ')', 'T', 'S', '(']
INPUT: ['$', ')', 'i', '+', ')', 'n', '(', '(']

STACK: ['$', ')', 'T', 'S']
INPUT: ['$', ')', 'i', '+', ')', 'n', '(']

STACK: ['$', ')', 'T', ')', 'T', 'S', '(']
INPUT: ['$', ')', 'i', '+', ')', 'n', '(']

STACK: ['$', ')', 'T', ')', 'T', 'S']
INPUT: ['$', ')', 'i', '+', ')', 'n']

STACK: ['$', ')', 'T', ')', 'T', 'F']
INPUT: ['$', ')', 'i', '+', ')', 'n']

STACK: ['$', ')', 'T', ')', 'T', 'n']
INPUT: ['$', ')', 'i', '+', ')', 'n']

STACK: ['$', ')', 'T', ')', 'T']
INPUT: ['$', ')', 'i', '+', ')']

STACK: ['$', ')', 'T', ')']
INPUT: ['$', ')', 'i', '+', ')']

STACK: ['$', ')', 'T']
INPUT: ['$', ')', 'i', '+']

STACK: ['$', ')', 'T', 'F', '+']
INPUT: ['$', ')', 'i', '+']

STACK: ['$', ')', 'T', 'F']
INPUT: ['$', ')', 'i']

STACK: ['$', ')', 'T', 'i']
INPUT: ['$', ')', 'i']

STACK: ['$', ')', 'T']
INPUT: ['$', ')']

STACK: ['$', ')']
INPUT: ['$', ')']

STACK: ['$']
INPUT: ['$']

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

Conclusion:

From the above experiment, I was able to implement code and programatically execute and verify the working of an LL(1) parser by hardcoded Input symbol calculated from the given grammar.

Ref.:

https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/