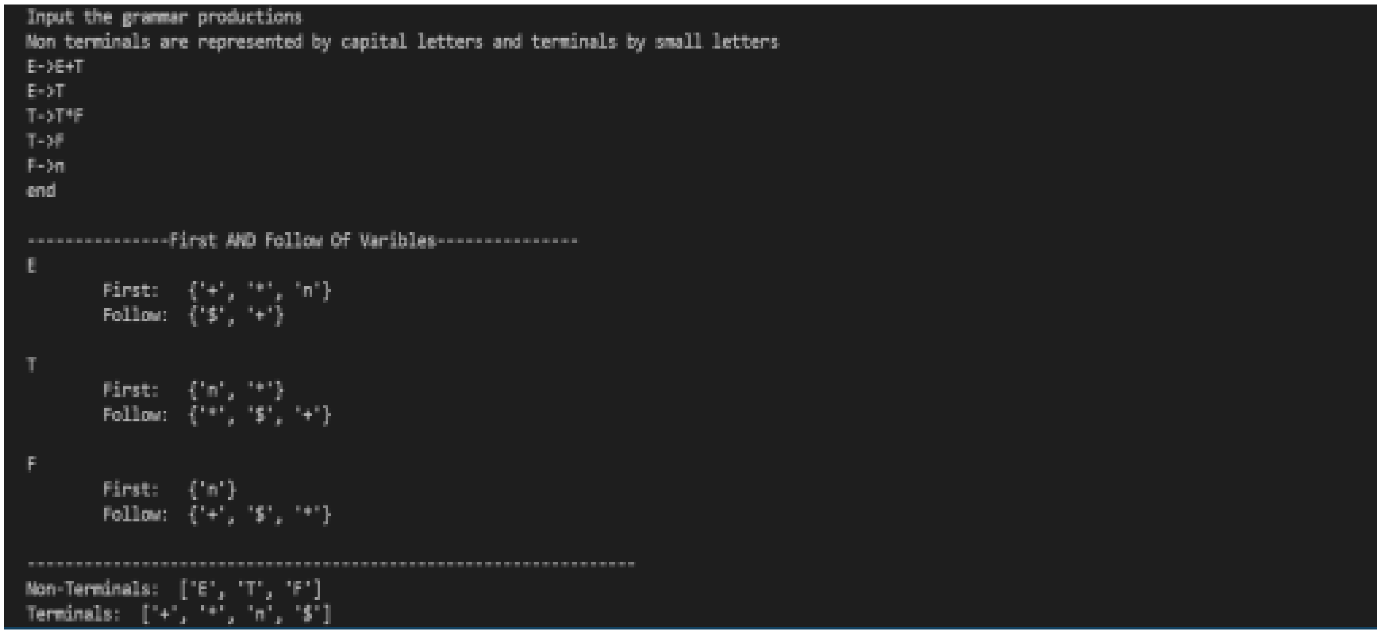
Problem statement:

Implementation of CLR(1) Parsing by first constructing LR(1) automaton,CLR(1) parse table and parsing the given input.

To run the file:

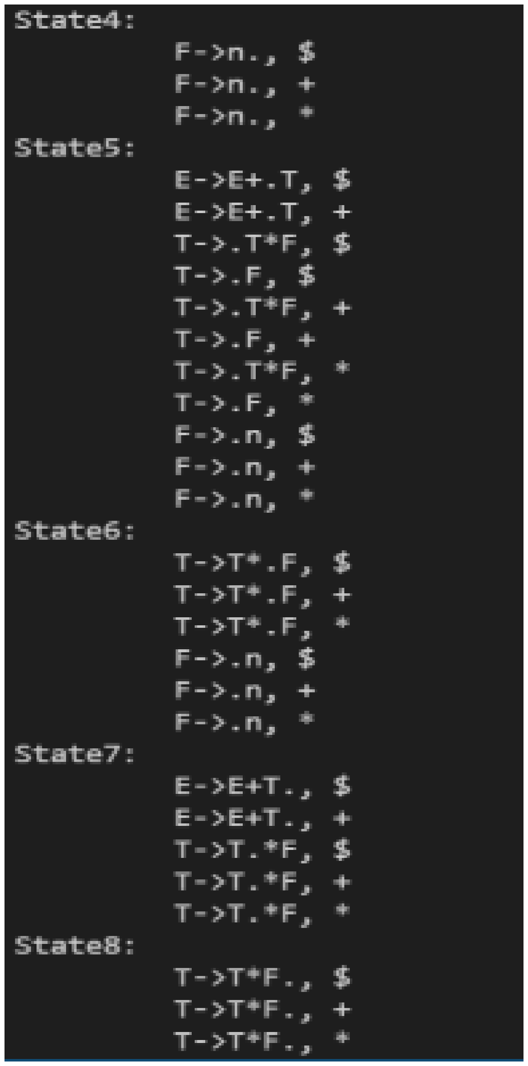
- ➢ Clone the files f1.py and f2.py.
- ➢ Open the terminal and run the command python f2.py
- ➢ Input the productions and you will get the parse table and after that input the string to find whether it is parsed by clr(1) or not.

Screenshots for Code of execution:

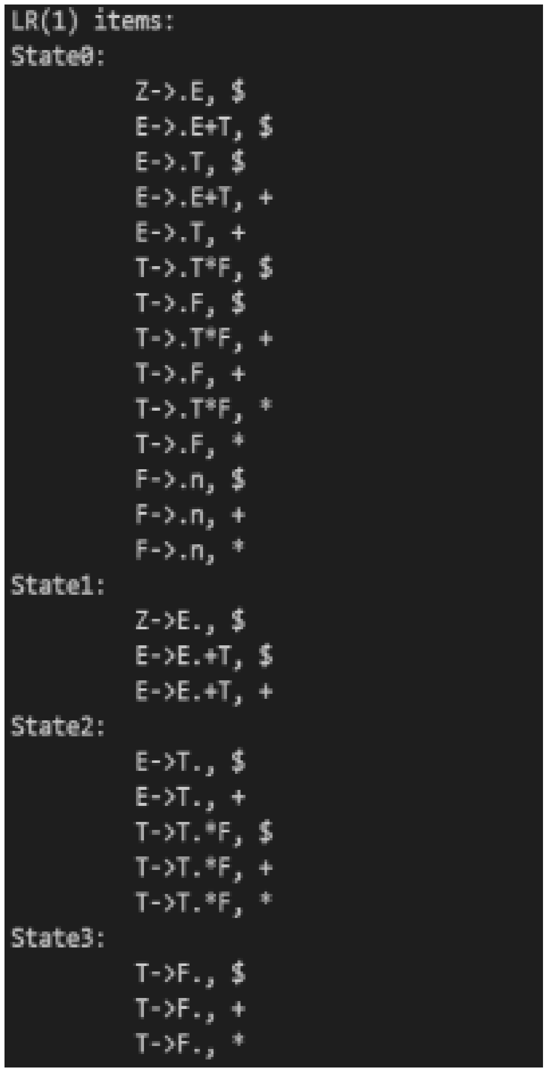


```
Input the grammar productions
Non terminals are represented by capital letters and terminals by small letters
E->E+T
E->T
T->T*F
T->F
F->n
end

---------------First AND Follow Of Varibles---------------
E
        First:   {'+', '*', 'n'}
        Follow:  {'$', '+'}

T
        First:   {'n', '*'}
        Follow:  {'*', '$', '+'}

F
        First:   {'n'}
        Follow:  {'+', '$', '*'}

----------------------------------------------------------------
Non-Terminals:  ['E', 'T', 'F']
Terminals:  ['+', '*', 'n', '$']
```

LR(1) items:

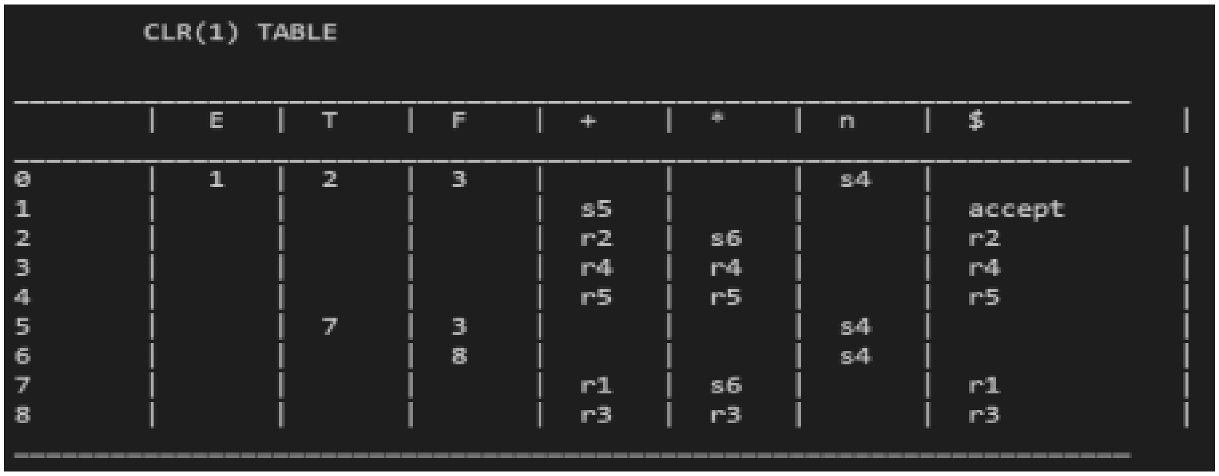


```
State4:
        F->n., $
        F->n., +
        F->n., *
State5:
        E->E+.T, $
        E->E+.T, +
        T->.T*F, $
        T->.F, $
        T->.T*F, +
        T->.F, +
        T->.T*F, *
        T->.F, *
        F->.n, $
        F->.n, +
        F->.n, *
State6:
        T->T*.F, $
        T->T*.F, +
        T->T*.F, *
        F->.n, $
        F->.n, +
        F->.n, *
State7:
        E->E+T., $
        E->E+T., +
        T->T.*F, $
        T->T.*F, +
        T->T.*F, *
State8:
        T->T*F., $
        T->T*F., +
        T->T*F., *
```



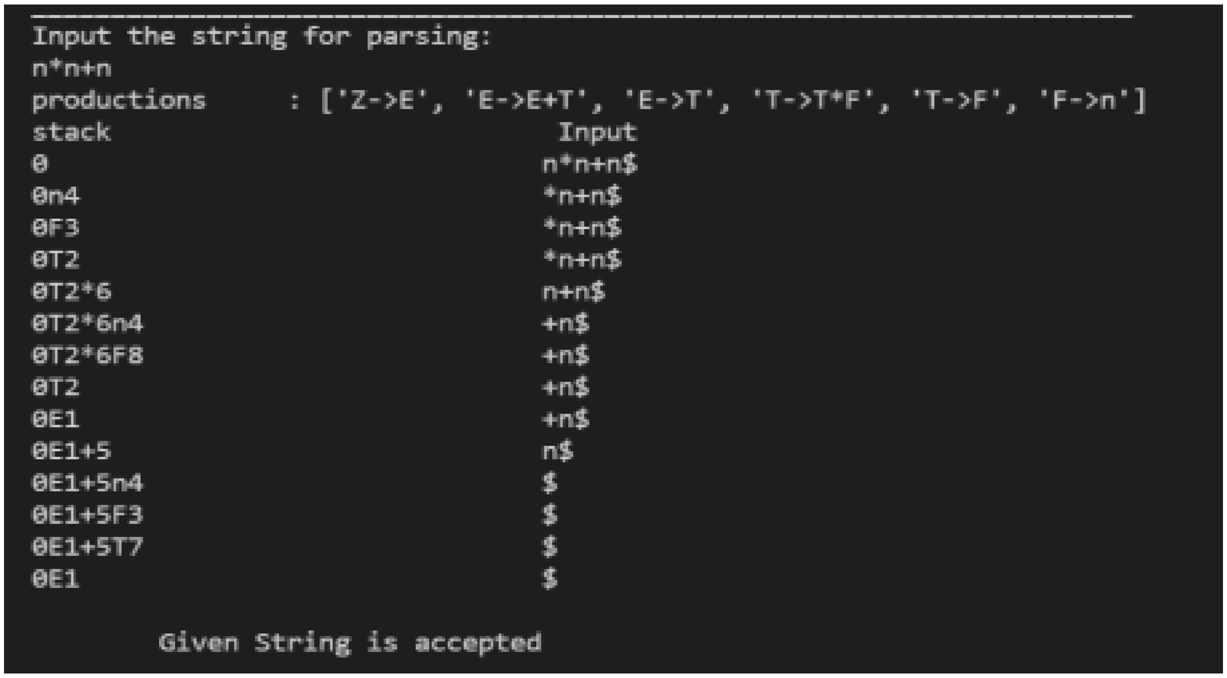
```
LR(1) items:
State0:
        Z->.E, $
        E->.E+T, $
        E->.T, $
        E->.E+T, +
        E->.T, +
        T->.T*F, $
        T->.F, $
        T->.T*F, +
        T->.F, +
        T->.T*F, *
        T->.F, *
        F->.n, $
        F->.n, +
        F->.n, *
State1:
        Z->E., $
        E->E.+T, $
        E->E.+T, +
State2:
        E->T., $
        E->T., +
        T->T.*F, $
        T->T.*F, +
        T->T.*F, *
State3:
        T->F., $
        T->F., +
        T->F., *
```

➢ Parse Table generated:



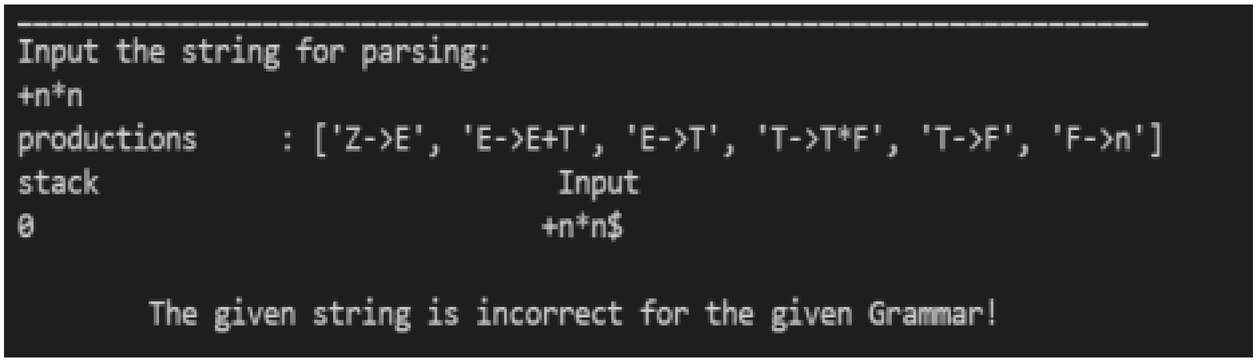
```
        CLR(1) TABLE

_____________________________________________________________________________
        |   E   |  T    |  F    |  +    |  *    |  n    |  $              |
_____________________________________________________________________________
0       |   1   |  2    |  3    |       |       |  s4   |                 |
1       |       |       |       |  s5   |       |       |  accept
2       |       |       |       |  r2   |  s6   |       |  r2             |
3       |       |       |       |  r4   |  r4   |       |  r4             |
4       |       |       |       |  r5   |  r5   |       |  r5             |
5       |       |  7    |  3    |       |       |  s4   |                 |
6       |       |       |  8    |       |       |  s4   |                 |
7       |       |       |       |  r1   |  s6   |       |  r1             |
8       |       |       |       |  r3   |  r3   |       |  r3             |
_____________________________________________________________________________
```

➢ Testing for the CLR(1) parsing with an example:(accepted)



```
--------------------------------------------------------------------------------
Input the string for parsing:
n*n+n
productions     : ['Z->E', 'E->E+T', 'E->T', 'T->T*F', 'T->F', 'F->n']
stack                           Input
0                              n*n+n$
0n4                            *n+n$
0F3                            *n+n$
0T2                            *n+n$
0T2*6                          n+n$
0T2*6n4                        +n$
0T2*6F8                        +n$
0T2                            +n$
0E1                            +n$
0E1+5                          n$
0E1+5n4                        $
0E1+5F3                        $
0E1+5T7                        $
0E1                            $

        Given String is accepted
```

➢ Testing for the CLR(1) parsing with an example:(Not accepted)



```
--------------------------------------------------------------------------------
Input the string for parsing:
+n*n
productions     : ['Z->E', 'E->E+T', 'E->T', 'T->T*F', 'T->F', 'F->n']
stack                           Input
0                              +n*n$

        The given string is incorrect for the given Grammar!
```

Explanation of the approach:

First, find the first and follow of the given non-terminals and then find out the closure items i.e. LR(1) items.Because for CLR(1) Parsing we will use this.

LR(k) item is defined to be an item using look ahead of length k.

So, the LR(1) item is composed of two parts: the LR(0) item and the lookahead associated with the item.

The first file (f1.py) computes the first and follow by distinguishing the terminals and non terminals.

The second file(f2.py) generates parse table and checks the given input whether it is accepted by CLR(1) or not.
