Exp:8: Computation of Leading and Trailing sets

AIM: To design a code to compute leading and trailing sets of the given grammar.

LANGUAGE USED: Python 3

ALGORITHM/PROCEDURE: -

- Epsilon is represented by ‘e’.
- Productions are of the form A=B, where ‘A’ is a single Non-Terminal and ‘B’ can be any combination of Terminals and Non- Terminals.
- Each production of a non-terminal is entered on a different line.
- Only Upper-Case letters are Non-Terminals and everything else is a terminal.
- Do not use ‘!’ or ‘$’ as they are reserved for special purposes.
- Grammar is taken as input and will be through an infinite loop (while=true).
-

EXPLANATION:

LEADING(A)

{

1. ‘a’ is in Leading(A) if A □ γaδ where γ is ε or any Non-Terminal

2. If’ ‘a’ is in Leading(B) and A □ Bα, then a in Leading(A)

}

Step 1 of algorithm leading, indicates how to add the first terminal occurring in the RHS of every production directly. Step 2 of the algorithm indicates to add the first terminal, through another non-terminal B to be included indirectly to the LEADING() of every non-terminal.

TRAILING (A)

{

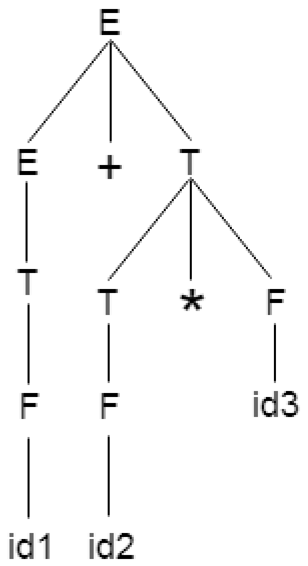
1. a is in Trailing(A) if A -> γaδ where δ is ε or any Non-Terminal

2. If a is in Trailing(B) and A -> αB, then a in Trailing(A)

}

Algorithm trailing is similar to algorithm leading and the only difference being, the symbol is looked from right to left as against left to right in algorithm leading. Step 1 of the algorithm trailing, indicates looking for the first terminal occurring in the RHS of a production from right side and thus adds the direct first symbol. The second step looks for adding the indirect first symbol from the right of the RHS of the production.

SPACE TREE DIAGRAM/ EXPLANATION:



SOURCE CODE: -

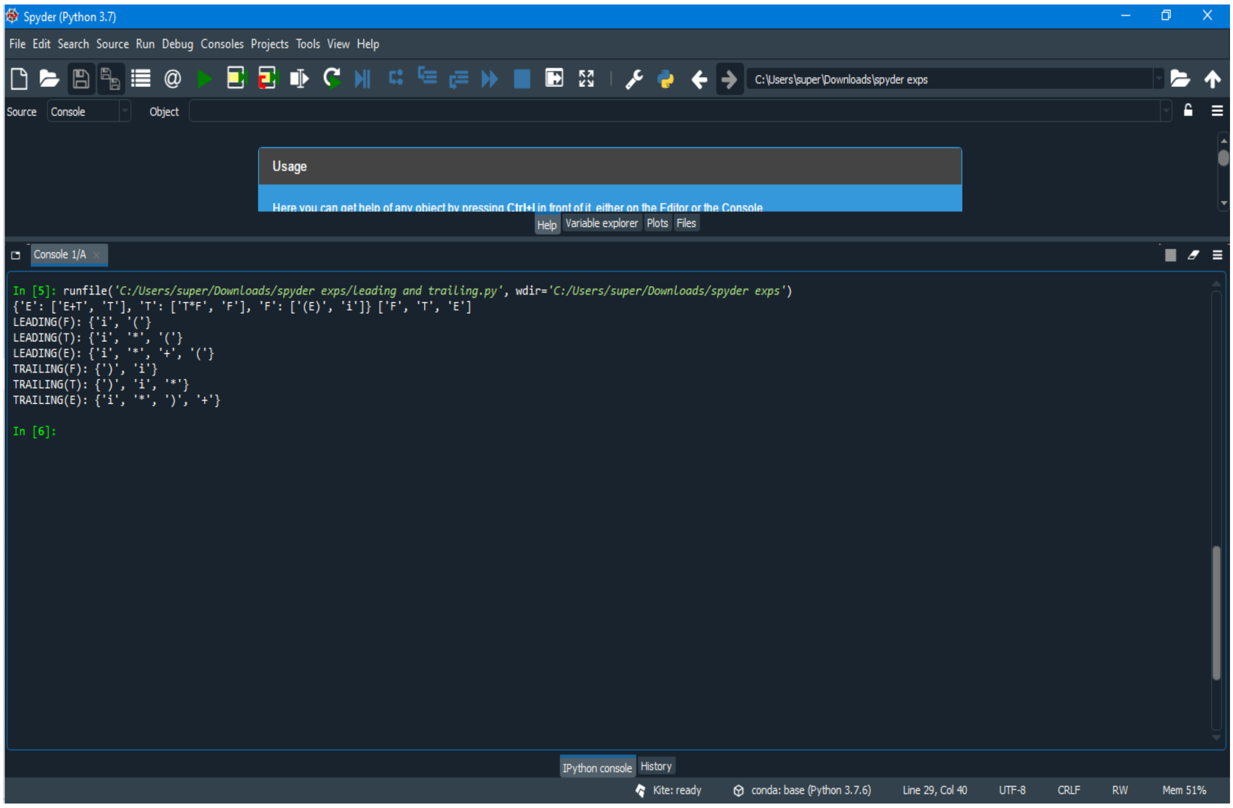
```
a = ["E=E+T",
     "E=T",
     "T=T*F",
     "T=F",
     "F=(E)",
     "F=i"]
rules = {}
terms = []
for i in a:
    temp = i.split("=")

    terms.append(temp[0])
    try:
        rules[temp[0]] += [temp[1]]
```

```
    except:
        rules[temp[0]] = [temp[1]]
terms = list(set(terms))
print(rules,terms)
def leading(gram, rules, term, start):
    s = []
    if gram[0] not in terms:
        return gram[0]
    elif len(gram) == 1:
        return [0]
    elif gram[1] not in terms and gram[-1] is not start:
        for i in rules[gram[-1]]:
            s+= leading(i, rules, gram[-1], start)
            s+= [gram[1]]
        return s
def trailing(gram, rules, term, start):
    s = []
    if gram[-1] not in terms:
        return gram[-1]
    elif len(gram) == 1:
        return [0]
    elif gram[-2] not in terms and gram[-1] is not start:

        for i in rules[gram[-1]]:
            s+= trailing(i, rules, gram[-1], start)
            s+= [gram[-2]]

             return s
leads = {}
trails = {}
for i in terms:
    s = [0]
```

```
    for j in rules[i]:
        s+=leading(j,rules,i,i)
    s = set(s)
    s.remove(0)
    leads[i] = s
    s = [0]
    for j in rules[i]:
        s+=trailing(j,rules,i,i)
    s = set(s)
    s.remove(0)
    trails[i] = s
for i in terms:
    print("LEADING("+i+"):",leads[i])
for i in terms:
    print("TRAILING("+i+"):",trails[i])
```

OUTPUT:

```
{'E': ['E+T', 'T'], 'T': ['T*F', 'F'], 'F': ['(E)', 'i']} ['F', 'T', 'E']
LEADING(F): {'i', '('}
LEADING(T): {'i', '*', '('}
LEADING(E): {'i', '*', '+', '('}
TRAILING(F): {')', 'i'}
TRAILING(T): {')', 'i', '*'}
TRAILING(E): {'i', '*', ')', '+'}
```



RESULT: Therefore, we successfully implemented a code for computing leading and trailing sets of the given grammar.