Aim

To implement left factoring from given grammar.

Program logic

Write a program to eliminate the Left Factoring using the given in Algorithm.

Sample Grammar Sample:

Rule:

A->αβ1 | αβ2

step 1: A -> αA'
step 2: A' -> β1 | β2

Example 1:

Given grammar: A → aAB / aBc / aAc

o/p: A → aA'

- A' → AB / Bc / Ac
- A → aA'
- A' → AD / Bc
- D → B / c

Example 2:

Given grammar: S → bSSaaS / bSSaSb / bSb / a

o/p: S → bSS' / a

- S' → SaaS / SaSb / b
- S → bSS' / a
- S' → SaA / b
- A → aS / Sb

Removing Left Factoring:

A grammar is said to be left factored when it is of the form –
A -> αβ1 | αβ2 | αβ3 | …… | αβn | γ

i.e the productions start with the same terminal (or set of terminals). On seeing the input α we cannot immediately tell which production to choose to expand A.
Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen to make the right choice.
For the grammar A -> αβ1 | αβ2 | αβ3 | …… | αβn | γ
The equivalent left factored grammar will be –
A -> αA' | γ
A' -> β1 | β2 | β3 | …… | βn

Lab Assignment

What is Left Factoring?

If more than one grammar production rules have a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.

Example

If a top-down parser encounters a production like

A ⟹ αβ | αγ | …

Then it cannot determine which production to follow to parse the string as both productions are starting from the same terminal (or non-terminal). To remove this confusion, we use a technique called **left factoring**.

Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.

Why to remove left Factoring?

Left recursion must be removed if the parser performs top-down parsing. Left Factoring is a grammar transformation technique. It consists in "factoring out" prefixes which are common to two or more productions

Define algorithm for left Factor Grammar.

For all A ∈ non-terminal, find the longest prefix a that occurs in two or more right-hand sides of A.

If a ^{1} ∈ then replace all of the A productions, A → a bI | a b2 | - - - | a bn | r

With

A → a AI | r AI → bI | b2| - - - | bn | ∈ Where, AI is a new element of non-terminal. Repeat until no common prefixes remain. It is easy to remove common prefixes by left factoring, creating new non-terminal.

For example, consider:

V → a b | a r Change to:

V → a VI VI → b | r

What are different rules for left factor.

First and Follow Sets

An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing T[A, t] = α with some production rule.

First Set

This set is created to know what terminal symbol is derived in the first position by a non-terminal. For example,

α → t β

That is α derives t (terminal) in the very first position. So, t ∈ FIRST(α).

Algorithm for calculating First set

Look at the definition of FIRST(α) set:

- if α is a terminal, then FIRST(α) = { α }.
- if α is a non-terminal and α → Ɛ is a production, then FIRST(α) = { Ɛ }.
- if α is a non-terminal and α → $\gamma 1\, \gamma 2\, \gamma 3 \ldots \gamma n$ and any FIRST(γ) contains t then t is in FIRST(α).

First set can be seen as:

$$\mathrm{FIRST}(\alpha) = \{\, t \mid \alpha \xrightarrow{*} t\beta \,\} \cup \{\, \varepsilon \mid \alpha \xrightarrow{*} \varepsilon \,\}$$

Follow Set

Likewise, we calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the productions of a non-terminal.

Algorithm for calculating Follow set:

- if α is a start symbol, then FOLLOW () = $
- if α is a non-terminal and has a production α → AB, then FIRST(B) is in FOLLOW(A) except Ɛ.
- if α is a non-terminal and has a production α → AB, where B Ɛ, then FOLLOW(A) is in FOLLOW(α).

Follow set can be seen as: FOLLOW(α) = { t | S *αt*}

Lab Assignment Program

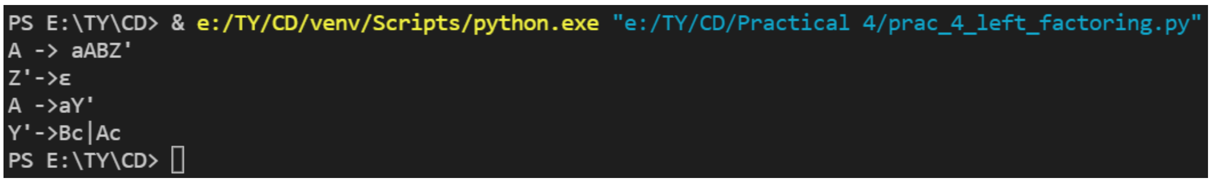
Write a program to implement left factoring from given grammar.

Code

```python
from itertools import takewhile
def groupby(ls):
    d = {}
    ls = [ y[0] for y in rules ]
    initial = list(set(ls))
    for y in initial:
        for i in rules:
            if i.startswith(y):
                if y not in d:
                    d[y] = []
                d[y].append(i)
    return d

def prefix(x):
    return len(set(x)) == 1


starting=""
rules=[]
common=[]
```

```
alphabetset=["A'","B'","C'","D'","E'","F'","G'","H'","I'","J'","K'","L'","M'",
"N'","O'","P'","Q'","R'","S'","T'","U'","V'","W'","X'","Y'","Z'"]


s= "A -> aAB|aBc|aAc"
while(True):
    rules=[]
    common=[]
    split=s.split("->")
    starting=split[0]
    for i in split[1].split("|"):
        rules.append(i)

#logic for taking commons out
    for k, l in groupby(rules).items():
        r = [l[0] for l in takewhile(prefix, zip(*l))]
        common.append(''.join(r))
#end of taking commons
    for i in common:
        newalphabet=alphabetset.pop()
        print(starting+"->"+i+newalphabet)
        index=[]
        for k in rules:
            if(k.startswith(i)):
                index.append(k)
        print(newalphabet+"->",end="")
        for j in index[:-1]:
            stringtoprint=j.replace(i,"", 1)+"|"
            if stringtoprint=="|":
                print("\u03B5","|",end="")
            else:
                print(j.replace(i,"", 1)+"|",end="")
        stringtoprint=index[-1].replace(i,"", 1)+"|"
        if stringtoprint=="|":
            print("\u03B5","",end="")
        else:
            print(index[-1].replace(i,"", 1)+"",end="")
        print("")
    break
```

Output



```
PS E:\TY\CD> & e:/TY/CD/venv/Scripts/python.exe "e:/TY/CD/Practical 4/prac_4_left_factoring.py"
A -> aABZ'
Z'->ε
A ->aY'
Y'->Bc|Ac
PS E:\TY\CD> 
```

Conclusion

Hence, we were able to implement left factoring from given grammar.