Aim

To check the different keywords and operators (tokens) in the given grammar/ Generate Lexical Analyzer (tokens from given source program)

Program Logic

1. Read the input Expression
2. Check whether input is alphabet or digits then store it as identifier
3. If the input contains operator store it as in symbol Table.
4. Check the input for keywords.
5. Check for special symbols from source program

Lab Assignment

1. What is token?

A token is a group of characters having collective meaning: typically, a word or punctuation mark, separated by a lexical analyser and passed to a parser. A lexeme is an actual character sequence forming a specific instance of a token, such as num. The pattern matches each string in the set.

2. What is lexeme?

A lexeme is a sequence of alphanumeric characters in a token. The term is used in both the study of language and in the lexical analysis of computer program compilation. In the context of computer programming, lexemes are part of the input stream from which tokens are identified.

3. What is the difference between token and lexeme?

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

- Identifiers
- keywords
- operators
- special symbols
- constants

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example: while(y > = t) = y -3

Will be represented by the set of pairs.

Lexeme	**Token**
while	while
(	lparen
y	identifier

>=	Comparison
t	identifier
)	Rparen
y	identifier
=	Assignment
y	identifier
-	Arithmetic
3	integer
;	Finish of a statement

4. Define phase and pass?

Pass: A pass refers to the traversal of a compiler through the entire program.

Phase: A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

5. What is the difference between phase and pass?

No.	Phase	Pass
1	The process of compilation is carried out in various step is called phase.	Various phases are logically grouped together to form a pass.
2	The phases of compilation are lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization and code generation.	The process of compilation can be carried out in a single pass or in multiple passes.

6. What is the difference between compiler and interpreter?

Compiler: It is a translator who takes input i.e., High-Level Language, and produces an output of low-level language i.e., machine or assembly language.

- A compiler is more intelligent than an assembler it checks all kinds of limits, ranges, errors, etc.
- But its program run time is more and occupies a larger part of memory. It has slow speed because a compiler goes through the entire program and then translates the entire program into machine codes.

Interpreter: An interpreter is a program that translates a programming language into a comprehensible language. –

- It translates only one statement of the program at a time.
- Interpreters often are smaller than compilers.

Compiler	Interpreter
Compiler scans the whole program in one go.	Translates program one statement at a time.
As it scans the code in one go, the errors (if any) are shown at the end together.	Considering it scans code one line at a time, errors are shown line by line.
Main advantage of compilers is its execution time.	Due to interpreters being slow in executing the object code, it is preferred less.
It converts the source code into object code.	It does not convert source code into object code instead it scans it line by line
It does not require source code for later execution.	It requires source code for later execution.
C, C++, C# etc.	Python, Ruby, Perl, SNOBOL, MATLAB, etc.

7. What is lexical analyser?

Lexical Analysis is the first phase of the compiler also known as a scanner. It converts the High-level input program into a sequence of Tokens.

- Lexical Analysis can be implemented with the **Deterministic finite Automata**.
- The output is a sequence of tokens that is sent to the parser for syntax analysis

Lab Assignment Program

Write a program to recognize:

1. Identifiers
2. Constants
3. Keywords

Code

Read.py

```
a = 12
b = 22
c = 20
d = a + b
m = 25
n = 10
j = m + n
```

Prac_02.py

```
import re

file = open("read.py")

operators = {
      '=': 'Assignment op','+': 'Addition op','-
': 'Subtraction op','/' : 'Division op'
      ,'*': 'Multiplication op','<': 'Lessthan op','>': 'Greaterthan op'}
operators_key = operators.keys()
```

```
data_type = {'int' : 'integer type', 'float': 'Floating point' , 'char' : 'Cha
racter type', 'long' : 'long int' }
data_type_key = data_type.keys()

punctuation_symbol = { ':' : 'colon', ';' : 'semi-
colon', '.' : 'dot' , ',' : 'comma' }
punctuation_symbol_key = punctuation_symbol.keys()

identifier = {
      'a':'id','b':'id','c':'id','d':'id','e':'id'
      ,'f':'id','g' :'id','h':'id','i':'id','j':'id'
      ,'k':'id','l':'id','m':'id','n':'id','o':'id'
      ,'p':'id','q':'id','r':'id','t':'id','u':'id'
      ,'v':'id','w':'id','x':'id','y':'id','z':'id'}

identifier_key = identifier.keys()

dataFlag = False

a=file.read()

count=0
program = a.split("\n")
for line in program:
    count = count + 1
    print("line#" , count, "\n" , line)

    tokens=line.split(' ')
    print("Tokens are " , tokens)
    print("Line#", count, "properties \n")
    for token in tokens:
        if token in operators_key:
            print("operator is ", operators[token])
        if token in data_type_key:
            print("datatype is", data_type[token])
        if token in punctuation_symbol_key:
            print (token, "Punctuation symbol is" , punctuation_symbol[token])
        if token in identifier_key:
            print (token, "Identifier is" , identifier[token])


    dataFlag=False
    print("---------------------------")
```

Output

```
[Running] python -u "e:\TY\CD\Prac_02.py"
line# 1
 a = 12
Tokens are  ['a', '=', '12', '']
Line# 1 properties

a Identifier is id
operator is  Assignment op
---------------------------
line# 2
 b = 22
Tokens are  ['b', '=', '22']
Line# 2 properties

b Identifier is id
operator is  Assignment op
---------------------------
line# 3
 c = 20
Tokens are  ['c', '=', '20']
Line# 3 properties

c Identifier is id
operator is  Assignment op
```

```
---------------------------
line# 4
 d = a + b
Tokens are  ['d', '=', 'a', '+', 'b']
Line# 4 properties

d Identifier is id
operator is  Assignment op
a Identifier is id
operator is  Addition op
b Identifier is id
---------------------------
line# 5
 m = 25
Tokens are  ['m', '=', '25']
Line# 5 properties

m Identifier is id
operator is  Assignment op
---------------------------
line# 6
 n = 10
Tokens are  ['n', '=', '10']
Line# 6 properties

n Identifier is id
operator is  Assignment op
```

```
--------------------------
line# 7
 j = m + n
Tokens are  ['j', '=', 'm', '+', 'n']
Line# 7 properties

j Identifier is id
operator is  Assignment op
m Identifier is id
operator is  Addition op
n Identifier is id
--------------------------
```

Conclusion

Hence, we were able to generate a Lexical Analyzer.

Aim

Write a program to implement left recursion.

Program logic

Check if the given grammar contains left recursion, if present then separate the production and start working on it.

In our example,

S-->S a/

S b

/c

/ d

Introduce a new nonterminal and write it at the last of every terminal. We produce a new nonterminal S'and write new production as,

S-->cS' / dS'

Write newly produced nonterminal in LHS and in RHS it can either produce or it can produce new production in which the terminals or non-terminals which followed the previous LHS will be replaced by new nonterminal at last.

S'-->?

/ aS'

/ bS'

So, after conversion the new equivalent production is

S-->cS' / dS'

S'-->? / aS' / bS'

Step by step elimination of this indirect left recursion

A -> Cd

B -> Ce

C -> A | B | f

In this case order would be A < B < C, and possible paths for recursion of non-terminal C would be

C=> A => Cd

and

C=> B => Ce

so new rules for C would be

C=> Cd | Ce | f

now you can simply just remove direct left recursion:

C=> fC'

C'=> dC' | eC' | eps

and the resulting non-recursive grammar would be:

A => Cd

B => Ce

C => fC'

C' => dC' | eC' | eps

Lab Assignment

1. What is Left Recursion?

- A production of grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having left recursion is called as Left Recursive Grammar.

Example-

$$S \rightarrow Sa / \in$$

(Left Recursive Grammar)

- Left recursion is a problematic situation for Top-down parsers.
- Therefore, left recursion must be eliminated from the grammar.

2. What is right recursion?

- A production of grammar is said to have **right recursion** if the rightmost variable of its RHS is same as variable of its LHS.
- A grammar containing a production having right recursion is called as Right Recursive Grammar.

Example-

$$S \rightarrow aS / \in$$

(Right Recursive Grammar)

- Right recursion does not create any problem for the Top-down parsers.
- Therefore, there is no need of eliminating right recursion from the grammar.

3. Why to remove left recursion?

Left recursion is a problematic situation for Top-down parsers. Therefore, left recursion has to be eliminated from the grammar.

4. Define algorithm for left recursion

Left recursion is eliminated by converting the grammar into a right recursive grammar.

If we have the left-recursive pair of productions-

$A \rightarrow A\alpha / \beta$

(Left Recursive Grammar)

where β does not begin with an A.

Then, we can eliminate left recursion by replacing the pair of productions with-

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \in$

(Right Recursive Grammar)

This right recursive grammar functions same as left recursive grammar.

5. What are different rules for left Recursion.

The production is left-recursive if the leftmost symbol on the right side is the same as the non-terminal on the left side. For example,

expr → expr + term.

If one were to code this production in a recursive-descent parser, the parser would go in an infinite loop.

We can eliminate the left-recursion by introducing new nonterminal and new productions rules.

Lab Assignment Program

Write a program to implement left recursion.

Code

```
gram = {}

def add(str):                                   #to rules together
    x = str.split("->")
    y = x[1]
    x.pop()
    z = y.split("|")
    x.append(z)
    gram[x[0]]=x[1]

def removeDirectLR(gramA, A):
```

```
    """gramA is dictonary"""
    temp = gramA[A]
    tempCr = []
    tempInCr = []
    for i in temp:
        if i[0] == A:
            #tempInCr.append(i[1:])
            tempInCr.append(i[1:]+[A+"'"])
        else:
            #tempCr.append(i)
            tempCr.append(i+[A+"'"])
    tempInCr.append(["e"])
    gramA[A] = tempCr
    gramA[A+"'"] = tempInCr
    return gramA


def checkForIndirect(gramA, a, ai):
    if ai not in gramA:
        return False
    if a == ai:
        return True
    for i in gramA[ai]:
        if i[0] == ai:
            return False
        if i[0] in gramA:
            return checkForIndirect(gramA, a, i[0])
    return False

def rep(gramA, A):
    temp = gramA[A]
    newTemp = []
    for i in temp:
        if checkForIndirect(gramA, A, i[0]):
            t = []
            for k in gramA[i[0]]:
                t=[]
                t+=k
                t+=i[1:]
                newTemp.append(t)

        else:
            newTemp.append(i)
    gramA[A] = newTemp
    return gramA

def rem(gram):
    c = 1
    conv = {}
```

```
    gramA = {}
    revconv = {}
    for j in gram:
        conv[j] = "A"+str(c)
        gramA["A"+str(c)] = []
        c+=1

    for i in gram:
        for j in gram[i]:
            temp = []
            for k in j:
                if k in conv:
                    temp.append(conv[k])
                else:
                    temp.append(k)
            gramA[conv[i]].append(temp)


    #print(gramA)
    for i in range(c-1,0,-1):
        ai = "A"+str(i)
        for j in range(0,i):
            aj = gramA[ai][0][0]
            if ai!=aj :
                if aj in gramA and checkForIndirect(gramA,ai,aj):
                    gramA = rep(gramA, ai)

    for i in range(1,c):
        ai = "A"+str(i)
        for j in gramA[ai]:
            if ai==j[0]:
                gramA = removeDirectLR(gramA, ai)
                break

    op = {}
    for i in gramA:
        a = str(i)
        for j in conv:
            a = a.replace(conv[j],j)
        revconv[i] = a

    for i in gramA:
        l = []
        for j in gramA[i]:
            k = []
            for m in j:
                if m in revconv:
                    k.append(m.replace(m,revconv[m]))
                else:
```

```
                    k.append(m)
            l.append(k)
        op[revconv[i]] = l

    return op

n = int(input("Enter No of Production: "))
for i in range(n):
    txt=input()
    add(txt)

result = rem(gram)

for x,y in result.items():
    print("The output after Left Reccursion is ::")
    print(f'{x} -> {y}')
```

Output

```
PS E:\TY\CD> & e:/TY/CD/venv/Scripts/python.exe "e:/TY/CD/Practical 3/prac_3_left_reccursion.py"
Enter No of Production: 3
E->ES|v
E->vE'
E->SE'|e
The output after Left Reccursion is ::
E -> [['S', 'E', "'"], ['e']]
PS E:\TY\CD>
```

Conclusion

Hence, we were able to implement left recursion.

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

$A \Longrightarrow \alpha\beta \mid \alpha\boldsymbol{\gamma} \mid \ldots$

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 → γ1 γ2 γ3 … γn and any FIRST(γ) contains t then t is in FIRST(α).

First set can be seen as:

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

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

```
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.

Name: Varun Khadayate	Subject: Compiler Design
Roll No: A016	Date of Submission: 3rd September 2021

Aim

To find the First and Follow of the given grammar

Program logic

First Function-

First(α) is a set of terminal symbols that begin in strings derived from α.

Example-

Consider the production rule-

$$A \rightarrow abc / def / ghi$$

Then, we have-

$$First(A) = \{ a , d , g \}$$

Rules For Calculating First Function-

Rule-01:

For a production rule $X \rightarrow \in$,

$$First(X) = \{ \in \}$$

Rule-02:

For any terminal symbol 'a',

$$First(a) = \{ a \}$$

Rule-03:

For a production rule $X \rightarrow Y_1Y_2Y_3$,

Calculating First(X)

- If $\in \notin First(Y_1)$, then $First(X) = First(Y_1)$
- If $\in \in First(Y_1)$, then $First(X) = \{ First(Y_1) - \in \} \cup First(Y_2Y_3)$

Calculating First(Y2Y3)

- If $\in \notin First(Y_2)$, then $First(Y_2Y_3) = First(Y_2)$
- If $\in \in First(Y_2)$, then $First(Y_2Y_3) = \{ First(Y_2) - \in \} \cup First(Y_3)$

Similarly, we can make expansion for any production rule $X \rightarrow Y_1Y_2Y_3.....Y_n$.

Follow Function-

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place $ in Follow(S).

Rule-02:

For any production rule A → αB,

$$Follow(B) = Follow(A)$$

Rule-03:

For any production rule A → αBβ,

- If ∈ ∉ First(β), then Follow(B) = First(β)
- If ∈ ∈ First(β), then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

Important Notes-

NOTE-01:

- ∈ may appear in the first function of a non-terminal.
- ∈ will never appear in the follow function of a non-terminal.

NOTE-02:

- Before calculating the first and follow functions, eliminate **Left Recursion** from the grammar, if present.

NOTE-03:

- We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

PRACTICE PROBLEMS BASED ON CALCULATING FIRST AND FOLLOW-

Problem-01:

Calculate the first and follow functions for the given grammar-

$$S \rightarrow aBDh$$

$$B \rightarrow cC$$

$$C \rightarrow bC / \in$$

$$D \rightarrow EF$$

$$E \rightarrow g / \in$$

$$F \rightarrow f / \in$$

Solution-

The first and follow functions are as follows-

First Functions-

- First(S) = { a }
- First(B) = { c }
- First(C) = { b , ∈ }
- First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }
- First(E) = { g , ∈ }
- First(F) = { f , ∈ }

Follow Functions-

- Follow(S) = { $ }
- Follow(B) = { First(D) – ∈ } ∪ First(h) = { g , f , h }
- Follow(C) = Follow(B) = { g , f , h }
- Follow(D) = First(h) = { h }
- Follow(E) = { First(F) – ∈ } ∪ Follow(D) = { f , h }
- Follow(F) = Follow(D) = { h }

Lab Assignment

What is First and Follow?

First(α) is a set of terminal symbols that begin in strings derived from α.

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Specify rules for first and follow?

Rules For Calculating First Function-

Rule-01:

For a production rule $X \rightarrow \in$,

$$First(X) = \{ \in \}$$

Rule-02:

For any terminal symbol ‘a’,

$$First(a) = \{ a \}$$

Rule-03:

For a production rule $X \rightarrow Y_1Y_2Y_3$,

Calculating First(X)

- If $\in \notin First(Y_1)$, then $First(X) = First(Y_1)$
- If $\in \in First(Y_1)$, then $First(X) = \{ First(Y_1) - \in \} \cup First(Y_2Y_3)$

Calculating First(Y2Y3)

- If $\in \notin First(Y_2)$, then $First(Y_2Y_3) = First(Y_2)$
- If $\in \in First(Y_2)$, then $First(Y_2Y_3) = \{ First(Y_2) - \in \} \cup First(Y_3)$

Similarly, we can make expansion for any production rule $X \rightarrow Y_1Y_2Y_3.....Y_n$.

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place $ in Follow(S).

Rule-02:

For any production rule $A \rightarrow \alpha B$,

$$Follow(B) = Follow(A)$$

Rule-03:

For any production rule $A \rightarrow \alpha B\beta$,

- If $\in \notin First(\beta)$, then $Follow(B) = First(\beta)$
- If $\in \in First(\beta)$, then $Follow(B) = \{ First(\beta) - \in \} \cup Follow(A)$

Define algorithm for first and follow?

Algorithm for calculating First 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 → γ1 γ2 γ3 … γn and any FIRST(γ) contains t then t is in FIRST(α).

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).

Lab Assignment Program

Write a program to implement first and follow from given grammar.

Code

```
gram = {
    "S":["aBDh"],
    "B":["cC"],
    "C":["bC","e"],
    "D":["EF"],
    "E":["g","e"],
    "F":["f","e"]
}

def removeDirectLR(gramA, A):
    """gramA is dictonary"""
    temp = gramA[A]
    tempCr = []
    tempInCr = []
    for i in temp:
        if i[0] == A:
            tempInCr.append(i[1:]+[A+"'"])
        else:
            tempCr.append(i+[A+"'"])
    tempInCr.append(["e"])
    gramA[A] = tempCr
    gramA[A+"'"] = tempInCr
    return gramA


def checkForIndirect(gramA, a, ai):
    if ai not in gramA:
        return False
    if a == ai:
        return True
    for i in gramA[ai]:
        if i[0] == ai:
            return False
        if i[0] in gramA:
            return checkForIndirect(gramA, a, i[0])
    return False

def rep(gramA, A):
    temp = gramA[A]
    newTemp = []
    for i in temp:
        if checkForIndirect(gramA, A, i[0]):
            t = []
            for k in gramA[i[0]]:
```

```
                t=[]
                t+=k
                t+=i[1:]
                newTemp.append(t)
        else:
            newTemp.append(i)
    gramA[A] = newTemp
    return gramA

def rem(gram):
    c = 1
    conv = {}
    gramA = {}
    revconv = {}
    for j in gram:
        conv[j] = "A"+str(c)
        gramA["A"+str(c)] = []
        c+=1

    for i in gram:
        for j in gram[i]:
            temp = []
            for k in j:
                if k in conv:
                    temp.append(conv[k])
                else:
                    temp.append(k)
            gramA[conv[i]].append(temp)

    for i in range(c-1,0,-1):
        ai = "A"+str(i)
        for j in range(0,i):
            aj = gramA[ai][0][0]
            if ai!=aj :
                if aj in gramA and checkForIndirect(gramA,ai,aj):
                    gramA = rep(gramA, ai)

    for i in range(1,c):
        ai = "A"+str(i)
        for j in gramA[ai]:
            if ai==j[0]:
                gramA = removeDirectLR(gramA, ai)
                break

    op = {}
    for i in gramA:
        a = str(i)
        for j in conv:
```

```
            a = a.replace(conv[j],j)
        revconv[i] = a

    for i in gramA:
        l = []
        for j in gramA[i]:
            k = []
            for m in j:
                if m in revconv:
                    k.append(m.replace(m,revconv[m]))
                else:
                    k.append(m)
            l.append(k)
        op[revconv[i]] = l

    return op

result = rem(gram)

def first(gram, term):
    a = []
    if term not in gram:
        return [term]
    for i in gram[term]:
        if i[0] not in gram:
            a.append(i[0])
        elif i[0] in gram:
            a += first(gram, i[0])
    return a

firsts = {}
for i in result:
    firsts[i] = first(result,i)
    print(f'First of ({i}):',firsts[i])


def follow(gram, term):
    a = []
    for rule in gram:
        for i in gram[rule]:
            if term in i:
                temp = i
                indx = i.index(term)
                if indx+1!=len(i):
                    if i[-1] in firsts:
                        a+=firsts[i[-1]]
                    else:
                        a+=[i[-1]]
                else:
```

```
                    a+=["e"]
                if rule != term and "e" in a:
                    a+= follow(gram,rule)
    return a

follows = {}
print("\n")
for i in result:
    follows[i] = list(set(follow(result,i)))
    if "e" in follows[i]:
        follows[i].pop(follows[i].index("e"))
    follows[i]+=["$"]
    print(f'Follow of ({i}):',follows[i])
```

Output

```
PS E:\TY\CD> & e:/TY/CD/venv/Scripts/python.exe "e:/TY/CD/Practical 5 and 6/prac_5_6_first_n_follow.py"
First of (S): ['a']
First of (B): ['c']
First of (C): ['b', 'e']
First of (D): ['g', 'e']
First of (E): ['g', 'e']
First of (F): ['f', 'e']


Follow of (S): ['$']
Follow of (B): ['h', '$']
Follow of (C): ['h', '$']
Follow of (D): ['h', '$']
Follow of (E): ['f', 'h', '$']
Follow of (F): ['h', '$']
PS E:\TY\CD>
```

Conclusion

Hence, we were able to implement first and follow of the given grammar.

Name: Varun Khadayate	Subject: Compiler Design
Roll No: A016	Date of Submission: 24th September 2021

Aim

To design predictive parser from given grammar. (LL1)

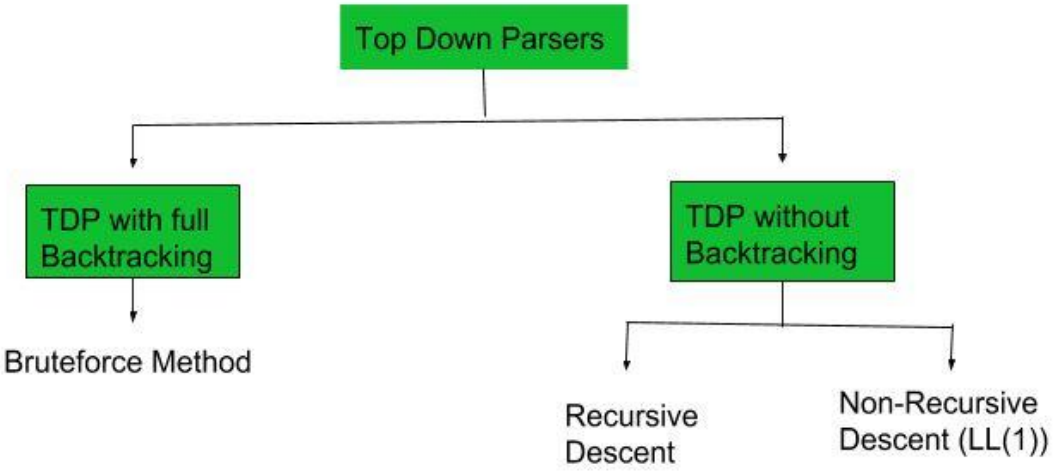
Program logic

1. First and Follow Sets
2. LL1 parsing Table
3. String parsing function which takes string as Input and outputs whether the string is accepted or rejected by the grammar

A top-down parser builds the parse tree from the top down, starting with the start non-terminal. There are two types of Top-Down Parsers:

1. Top-Down Parser with Backtracking
2. Top-Down Parsers without Backtracking

Top-Down Parsers without backtracking can further be divided into two parts:



LL (1) parsing is a top-down parsing method in the syntax analysis phase of compiler design. Required components for LL (1) parsing are input string, a stack, parsing table for given grammar, and parser. Here, we discuss a parser that determines that given string can be generated from a given grammar (or parsing table) or not.

Let given grammar is G = (V, T, S, P)

where V-variable symbol set, T-terminal symbol set, S- start symbol, P- production set.

LL (1) Parser algorithm:

Input

1. stack = S //stack initially contains only S.

2. input string = w$
   where S is the start symbol of grammar, w is given string, and $ is used for the end of string.
3. PT is a parsing table of given grammar in the form of a matrix or 2D array.

Output

Determines that given string can be produced by given grammar (parsing table) or not, if not then it produces an error.

Steps

```
1. while(stack is not empty) {

       // initially it is S
2.     A = top symbol of stack;

       //initially it is first symbol in string, it can be $ also
3.     r = next input symbol of given string;
4.     if (A∈T or A==$) {
5.         if(A==r){
6.             pop A from stack;
7.             remove r from input;
8.         }
9.         else
10.             ERROR();
11.     }
12.     else if (A∈V) {
13.         if(PT[A,r]= A⇢B1B2....Bk) {
14.             pop A from stack;

                // B1 on top of stack at final of this step
15.             push Bk,Bk-1......B1 on stack
16.         }
17.         else if (PT[A,r] = error())
18.             error();
19.     }
20. }
```

// if parser terminate without error ()

// then given string can generated by given parsing table.

p is the maximum of lengths of strings in RHS of all productions and

l is the length of given string and

l is very larger than p. if block at line 4 of algorithm always runs for O(1) time. else if block at line 12 in algorithm takes O(|P|*p) as upper bound for a single next input symbol. And while loop can run for more than l times, but we have considered the repeated while loop for a single next input symbol in O(|P|*p). So, the total time complexity is

T(l) = O(l)*O(|P|*p)

= O(l*|P|*p)

= O(l) { as l >>>|P|*p }

The time complexity of this algorithm is the order of length of the input string.

Comparison with Context-free language (CFL):

Languages in LL(1) grammar is a proper subset of CFL. Using the CYK algorithm we can find membership of a string for a given Context-free grammar (CFG). CYK takes O(l3) time for the membership test for CFG. But for LL(1) grammar we can do a membership test in O(l) time which is linear using the above algorithm. If we know that given CFG is LL(1) grammar then use LL(1) parser for parsing rather than CYK algorithm.

Time complexity

As we know that size of a grammar for a language is finite. Like, a finite number of variables, terminals, and productions. If grammar is finite then its LL(1) parsing table is also finite of size O(V*T). Let

Example

Let the grammar G = (V, T, S', P) is

*S' → **S$***
S → xYzS | a
Y → xYz | y

Parsing table (PT) for this grammar

	a	***x***	***y***	***z***	***$***
S'	*S' → S$*	*S' → S$*	*error*	*error*	*error*
S	*S → a*	*S → xYzS*	*error*	*error*	*error*
Y	*error*	*Y → xYz*	*Y → y*	*error*	*error*

Let string1 = xxyzza,

We have to add $ with this string,
We will use the above parsing algorithm, diagram for the process:

For string1 we got an empty stack, and while loop or algorithm terminated without error.
So, string1 belongs to language for given grammar G.

Let *string2 = xxyzzz,*

For string2, at the last stage as in the above diagram when the top of the stack is S and the next input symbol of the string is z, but in PT[S,z] = error. The algorithm terminated with an error.
So, string2 is not in the language of grammar G.

First Function-

First(α) is a set of terminal symbols that begin in strings derived from α.

Example-

Consider the production rule-

$$A \rightarrow abc / def / ghi$$

Then, we have-

$$First(A) = \{a, d, g\}$$

Rules For Calculating First Function-

Rule-01:

For a production rule $X \rightarrow \in$,

$$First(X) = \{\in\}$$

Rule-02:

For any terminal symbol 'a',

$$First(a) = \{ a \}$$

Rule-03:

For a production rule X → Y1Y2Y3,

Calculating First(X)

- If ∈ ∉ First(Y1), then First(X) = First(Y1)
- If ∈ ∈ First(Y1), then First(X) = { First(Y1) – ∈ } ∪ First(Y2Y3)

Calculating First(Y2Y3)

- If ∈ ∉ First(Y2), then First(Y2Y3) = First(Y2)
- If ∈ ∈ First(Y2), then First(Y2Y3) = { First(Y2) – ∈ } ∪ First(Y3)

Similarly, we can make expansion for any production rule X → Y1Y2Y3…..Yn.

Follow Function-

> Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place $ in Follow(S).

Rule-02:

For any production rule A → αB,

Follow(B) = Follow(A)

Rule-03:

For any production rule A → αBβ,

- If ∈ ∉ First(β), then Follow(B) = First(β)
- If ∈ ∈ First(β), then Follow(B) = { First(β) – ∈ } ∪ Follow(A)

Important Notes-

Note-01:

- ∈ may appear in the first function of a non-terminal.
- ∈ will never appear in the follow function of a non-terminal.

Note-02:

Before calculating the first and follow functions, eliminate Left Recursion from the grammar, if present.

Note-03:

We calculate the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

Practice problems based on calculating first and follow-

Problem-01:

Calculate the first and follow functions for the given grammar-

$$S \rightarrow aBDh$$

$$B \rightarrow cC$$

$$C \rightarrow bC / \in$$

$$D \rightarrow EF$$

$$E \rightarrow g / \in$$

$$F \rightarrow f / \in$$

Solution-

The first and follow functions are as follows-

First Functions-

- First(S) = { a }
- First(B) = { c }
- First(C) = { b , ∈ }
- First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }
- First(E) = { g , ∈ }
- First(F) = { f , ∈ }

Follow Functions-

- Follow(S) = {$ }
- Follow(B) = {First(D) – ∈} ∪ First(h) = {g , f , h }
- Follow(C) = Follow(B) = {g, f , h }
- Follow(D) = First(h) = {h }
- Follow(E) = {First(F) – ∈} ∪ Follow(D) = { f , h }
- Follow(F) = Follow(D) = {h }

LAB Assignment

What is LL1 Parser

A top-down parser that uses a one-token lookahead is called an LL(1) parser.

- The first L indicates that the input is read from left to right.
- The second L says that it produces a left-to-right derivation.
- And the 1 says that it uses one lookahead token. (Some parsers look ahead at the next 2 tokens, or even more than that.)

What is First and Follow?

First(α) is a set of terminal symbols that begin in strings derived from α.

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Specify rules for LL1 parser.

Specify rules for first and follow.

Rules For Calculating First Function-

Rule-01:

For a production rule $X \rightarrow \in$,

$$First(X) = \{ \in \}$$

Rule-02:

For any terminal symbol 'a',

$$First(a) = \{ a \}$$

Rule-03:

For a production rule $X \rightarrow Y_1Y_2Y_3$,

Calculating First(X)

- If $\in \notin First(Y_1)$, then $First(X) = First(Y_1)$
- If $\in \in First(Y_1)$, then $First(X) = \{ First(Y_1) - \in \} \cup First(Y_2Y_3)$

Calculating First(Y2Y3)

- If $\in \notin First(Y_2)$, then $First(Y_2Y_3) = First(Y_2)$
- If $\in \in First(Y_2)$, then $First(Y_2Y_3) = \{ First(Y_2) - \in \} \cup First(Y_3)$

Similarly, we can make expansion for any production rule $X \rightarrow Y_1Y_2Y_3.....Y_n$.

Rules For Calculating Follow Function-

Rule-01:

For the start symbol S, place $ in Follow(S).

Rule-02:

For any production rule $A \rightarrow \alpha B$,

$$Follow(B) = Follow(A)$$

Rule-03:

For any production rule $A \rightarrow \alpha B\beta$,

- If $\in \notin First(\beta)$, then $Follow(B) = First(\beta)$
- If $\in \in First(\beta)$, then $Follow(B) = \{ First(\beta) - \in \} \cup Follow(A)$

Define algorithm for LL1parser.

Input

1. stack = S //stack initially contains only S.
2. input string = w$
   where S is the start symbol of grammar, w is given string, and $ is used for the end of string.
3. PT is a parsing table of given grammar in the form of a matrix or 2D array.

Output

Determines that given string can be produced by given grammar(parsing table) or not, if not then it produces an error.

```
1. while(stack is not empty) {

       // initially it is S
2.     A = top symbol of stack;

       //initially it is first symbol in string, it can be $ also
3.     r = next input symbol of given string;
4.     if (A∈T or A==$) {
5.         if(A==r){
6.             pop A from stack;
7.             remove r from input;
8.         }
9.         else
10.             ERROR();
11.     }
12.     else if (A∈V) {
13.         if(PT[A,r]= A⇢B1B2....Bk) {
14.             pop A from stack;

                // B1 on top of stack at final of this step
15.             push Bk,Bk-1......B1 on stack
16.         }
17.         else if (PT[A,r] = error())
18.             error();
```

```
19.     }
20. }
```

Lab Assignment Program

Write a program to design predictive parser from given grammar. (LL1)

Code

```
gram = {
    "S":["aBDh"],
    "B":["cC"],
    "C":["bC","e"],
    "D":["EF"],
    "E":["g","e"],
    "F":["f","e"]
}

def removeDirectLR(gramA, A):
    temp = gramA[A]
    tempCr = []
    tempInCr = []
    for i in temp:
        if i[0] == A:
            tempInCr.append(i[1:]+[A+"'"])
        else:
            tempCr.append(i+[A+"'"])
    tempInCr.append(["e"])
    gramA[A] = tempCr
    gramA[A+"'"] = tempInCr
    return gramA

def checkForIndirect(gramA, a, ai):
    if ai not in gramA:
        return False
    if a == ai:
        return True
    for i in gramA[ai]:
        if i[0] == ai:
            return False
        if i[0] in gramA:
            return checkForIndirect(gramA, a, i[0])
    return False

def rep(gramA, A):
    temp = gramA[A]
    newTemp = []
    for i in temp:
```

```
        if checkForIndirect(gramA, A, i[0]):
            t = []
            for k in gramA[i[0]]:
                t=[]
                t+=k
                t+=i[1:]
                newTemp.append(t)
        else:
            newTemp.append(i)
    gramA[A] = newTemp
    return gramA

def rem(gram):
    c = 1
    conv = {}
    gramA = {}
    revconv = {}
    for j in gram:
        conv[j] = "A"+str(c)
        gramA["A"+str(c)] = []
        c+=1

    for i in gram:
        for j in gram[i]:
            temp = []
            for k in j:
                if k in conv:
                    temp.append(conv[k])
                else:
                    temp.append(k)
            gramA[conv[i]].append(temp)

    for i in range(c-1,0,-1):
        ai = "A"+str(i)
        for j in range(0,i):
            aj = gramA[ai][0][0]
            if ai!=aj :
                if aj in gramA and checkForIndirect(gramA,ai,aj):
                    gramA = rep(gramA, ai)

    for i in range(1,c):
        ai = "A"+str(i)
        for j in gramA[ai]:
            if ai==j[0]:
                gramA = removeDirectLR(gramA, ai)
                break

    op = {}
```

```
    for i in gramA:
        a = str(i)
        for j in conv:
            a = a.replace(conv[j],j)
        revconv[i] = a

    for i in gramA:
        l = []
        for j in gramA[i]:
            k = []
            for m in j:
                if m in revconv:
                    k.append(m.replace(m,revconv[m]))
                else:
                    k.append(m)
            l.append(k)
        op[revconv[i]] = l

    return op

result = rem(gram)
terminals = []
for i in result:
    for j in result[i]:
        for k in j:
            if k not in result:
                terminals+=[k]
terminals = list(set(terminals))

def first(gram, term):
    a = []
    if term not in gram:
        return [term]
    for i in gram[term]:
        if i[0] not in gram:
            a.append(i[0])
        elif i[0] in gram:
            a += first(gram, i[0])
    return a

print("First and Follow of the Given Grammer.")
firsts = {}
for i in result:
    firsts[i] = first(result,i)
    print(f'First of ({i}):',firsts[i])

def follow(gram, term):
    a = []
```

```
    for rule in gram:
        for i in gram[rule]:
            if term in i:
                temp = i
                indx = i.index(term)
                if indx+1!=len(i):
                    if i[-1] in firsts:
                        a+=firsts[i[-1]]
                    else:
                        a+=[i[-1]]
                else:
                    a+=["e"]
                if rule != term and "e" in a:
                    a+= follow(gram,rule)
    return a

follows = {}
print("\n")
for i in result:
    follows[i] = list(set(follow(result,i)))
    if "e" in follows[i]:
        follows[i].pop(follows[i].index("e"))
    follows[i]+=["$"]
    print(f'Follow of ({i}):',follows[i])

resMod = {}
for i in result:
    l = []
    for j in result[i]:
        temp = ""
        for k in j:
            temp+=k
        l.append(temp)
    resMod[i] = l

print("Transition Table for the given LL1 Grammer.")

tterm = list(terminals)
tterm.pop(tterm.index("e"))
tterm+=["$"]
pptable = {}
for i in result:
    for j in tterm:
        if j in firsts[i]:
            pptable[(i,j)]=resMod[i[0]][0]
        else:
            pptable[(i,j)]=""
    if "e" in firsts[i]:
```

```
        for j in tterm:
            if j in follows[i]:
                pptable[(i,j)]="e"
pptable[("F","i")] = "i"
toprint = f'{"": <10}'
for i in tterm:
    toprint+= f'|{i: <10}'
print(toprint)
for i in result:
    toprint = f'{i: <10}'
    for j in tterm:
        if pptable[(i,j)]!="":
            toprint+=f'|{i+"->"+pptable[(i,j)]: <10}'
        else:
            toprint+=f'|{pptable[(i,j)]: <10}'
    print(f'{"-":-<76}')
    print(toprint)
```

Output

```
PS E:\TY\CD> & e:/TY/CD/venv/Scripts/python.exe "e:/TY/CD/Practical 7/prac_7_predictive_parsr_LL1.py"
First and Follow of the Given Grammer.
First of (S): ['a']
First of (B): ['c']
First of (C): ['b', 'e']
First of (D): ['g', 'e']
First of (E): ['g', 'e']
First of (F): ['f', 'e']
Follow of (S): ['$']
Follow of (B): ['h', '$']
Follow of (C): ['h', '$']
Follow of (D): ['h', '$']
Follow of (E): ['h', 'f', '$']
Follow of (F): ['h', '$']
Transition Table for the given LL1 Grammer.
          |b         |f         |c         |a         |g         |h         |$
--------------------------------------------------------------------------------
S         |          |          |          |S->aBDh   |          |          |
--------------------------------------------------------------------------------
B         |          |          |B->cC     |          |          |          |
--------------------------------------------------------------------------------
C         |C->bC     |          |          |          |          |C->e      |C->e
--------------------------------------------------------------------------------
D         |          |          |          |          |D->EF     |D->e      |D->e
--------------------------------------------------------------------------------
E         |          |E->e      |          |          |E->g      |E->e      |E->e
--------------------------------------------------------------------------------
F         |          |F->f      |          |          |          |F->e      |F->e
```

Conclusion

Hence, we were able to design a predictive parser from given grammar. (LL1)

Name: Varun Khadayate	Subject: Compiler Design
Roll No: A016	Date of Submission: 24th September 2021

Aim

To design shift reduce parser from given grammar. (shift reduce)

Program logic

Shift Reduce parser

attempts for the construction of parse in a similar manner as done in bottom up parsing i.e. the parse tree is constructed from leaves(bottom) to the root(up). A more general form of shift reduce parser is LR parser.
This parser requires some data structures i.e.

- A input buffer for storing the input string.
- A stack for storing and accessing the production rules.

Basic Operations –

Shift

This involves moving of symbols from input buffer onto the stack.

Reduce

If the handle appears on top of the stack, then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.

Accept

If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.

Error

This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Example 1

Consider the grammar

S –> S + S
S –> S * S
S –> id

Perform Shift Reduce parsing for input string “id + id + id”.

Example 2

Consider the grammar

E –> 2E2
E –> 3E3
E –> 4

Perform Shift Reduce parsing for input string “32423”.

Example 3

Consider the grammar

S –> (L) | a
L –> L , S | S

Perform Shift Reduce parsing for input string “(a , (a , a)) “.

Stack	Input Buffer	Parsing Action
$	(a , (a , a)) $	Shift
$ (	a , (a , a)) $	Shift
$ (a	, (a , a)) $	Reduce S → a
$ (S	, (a , a)) $	Reduce L → S
$ (L	, (a , a)) $	Shift
$ (L ,	(a , a)) $	Shift
$ (L , (	a , a)) $	Shift
$ (L , (a	, a)) $	Reduce S → a
$ (L , (S	, a)) $	Reduce L → S
$ (L , (L	, a)) $	Shift
$ (L , (L ,	a)) $	Shift
$ (L , (L , a	)) $	Reduce S → a
$ (L , (L , S)	)) $	Reduce L →L , S
$ (L , (L	)) $	Shift
$ (L , (L)	) $	Reduce S → (L)
$ (L , S	) $	Reduce L → L , S
$ (L	) $	Shift
$ (L)	$	Reduce S → (L)

Stack	Input Buffer	Parsing Action
$ S	$	Accept

Lab Assignment

What is shift reduce Parser

A shift-reduce parser is a class of efficient, table-driven bottom-up parsing methods for computer languages and other notations formally defined by a grammar. The parsing methods most used for parsing programming languages, LR parsing and its variations, are shift-reduce methods.

What are different types?

There are two main categories of shift reduce parsing as follows:

1. Operator-Precedence Parsing
2. LR-Parser

Specify rules shift reduce parser.

Shift

This involves moving of symbols from input buffer onto the stack.

Reduce

If the handle appears on top of the stack, then, its reduction by using appropriate production rule is done i.e. RHS of production rule is popped out of stack and LHS of production rule is pushed onto the stack.

Accept

If only start symbol is present in the stack and the input buffer is empty then, the parsing action is called accept. When accept action is obtained, it is means successful parsing is done.

Error

This is the situation in which the parser can neither perform shift action nor reduce action and not even accept action.

Define algorithm for shift reduce parser.

1. Get the input expression and store it in the input buffer.
2. Read the data from the input buffer one at the time.
3. Using stack and push & pop operation shift and reduce symbols with respect to production rules available.
4. Continue the process till symbol shift and production rule reduce reaches the start symbol.
5. Display the Stack Implementation table with corresponding Stack actions with input symbols.

Lab Assignment Program

Write a program to design a shift reduce parser from given grammar. (shift reduce)

Code

```
gram = {
    "S":["S+S","S*S","id"]
```

```
}
starting_terminal = "S"
inp = "id+id*id"

stack = "$"
print(f'{"Stack": <15}'+"|"+f'{"Input Buffer": <15}'+"|"+f'Parsing Action')
print(f'{"-":-<50}')

while True:
    action = True
    i = 0
    while i<len(gram[starting_terminal]):
        if gram[starting_terminal][i] in stack:
            stack = stack.replace(gram[starting_terminal][i],starting_terminal
)
            print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Reduce S-
>{gram[starting_terminal][i]}')
            i=-1
            action = False
        i+=1
    if len(inp)>1:
        stack+=inp[0]
        inp=inp[1:]
        print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Shift')
        action = False

    if inp == "$" and stack == ("$"+starting_terminal):
        print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Accepted')
        break

    if action:
        print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Rejected')
        break
```

Output

```
PS E:\TY\CD> & e:/TY/CD/venv/Scripts/python.exe "e:/TY/CD/Practical 8/prac_8_shift_reduce_parser.py"
Stack          |Input Buffer    |Parsing Action
--------------------------------------------------
$i             |d+id*id         |Shift
$id            |+id*id          |Shift
$S             |+id*id          |Reduce S->id
$S+            |id*id           |Shift
$S+i           |d*id            |Shift
$S+id          |*id             |Shift
$S+S           |*id             |Reduce S->id
$S             |*id             |Reduce S->S+S
$S*            |id              |Shift
$S*i           |d               |Shift
$S*i           |d               |Rejected
```

Name: Varun Khadayate	Subject: Compiler Design
Roll No: A016	Date of Submission: 2nd October 2021

Aim

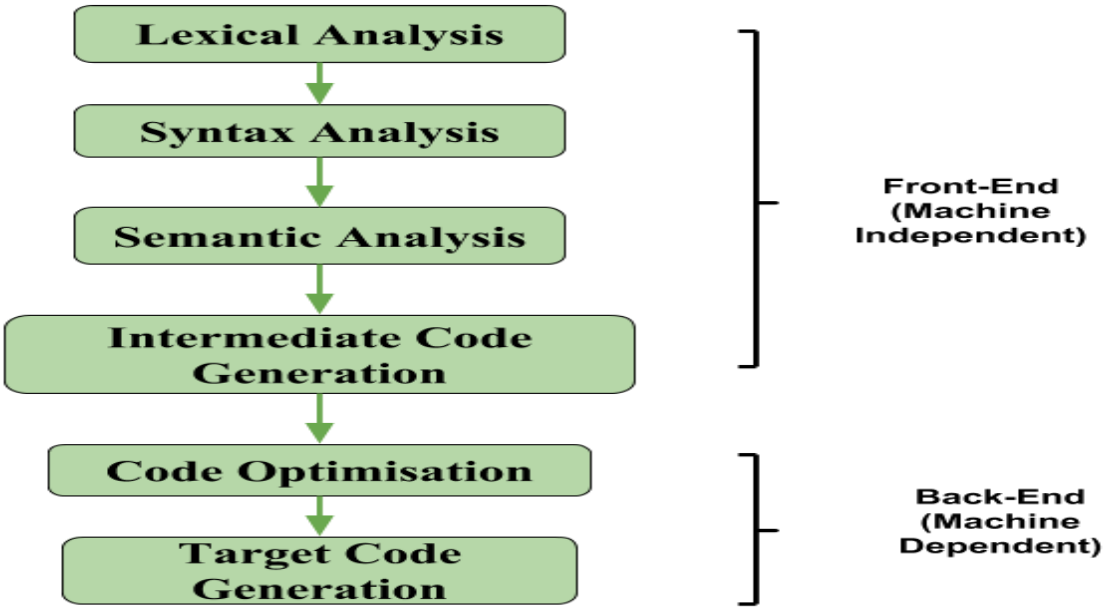
To implement intermediate code generation.

Program logic

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

The benefits of using machine independent intermediate code are:

- Because of the machine independent intermediate code, portability will be enhanced. For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
- Retargeting is facilitated
- It is easier to apply source code modification to improve the performance of source code by optimising the intermediate code.



If we generate machine code directly from source code then for n target machine, we will have n optimisers and n code generators but if we will have a machine independent intermediate code, we will have only one optimiser. Intermediate code can be either language specific (e.g., Bytecode for Java) or language. independent (three-address code).

The following are commonly used intermediate code representation:

Postfix Notation –

The ordinary (infix) way of writing the sum of a and b is with operator in the middle: a + b
The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

Example –

The postfix representation of the expression
(a – b) * (c + d) + (a – b)
is: ab – cd + *ab -+.

Three-Address Code –

A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z, here x, y, z will have address (memory location). Sometimes a statement might contain less than three references, but it is still called three address statement.

Example –

The three-address code for the expression
a + b * c + d:

T 1 = b * c
T 2 = a + T 1
T 3 = T 2 + d

T 1, T 2, T 3 are temporary variables.

Syntax Tree –

Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Example –

x = (a + b * c) / (a – b * c)

Lab Assignment Program

Implement Intermediate Code Generation.

Code

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+':1, '-':1, '*':2, '/':2}

def infix_to_postfix(formula):
```

```
    stack = []
    output = ''
    for ch in formula:
        if ch not in OPERATORS:
            output += ch
        elif ch == '(':
            stack.append('(')
        elif ch == ')':
            while stack and stack[-1] != '(':
                output += stack.pop()
            stack.pop()
        else:
            while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
                output += stack.pop()
            stack.append(ch)
    while stack:
        output += stack.pop()
    print(f'{output}')
    return output

def infix_to_prefix(formula):
    op_stack = []
    exp_stack = []
    for ch in formula:
        if not ch in OPERATORS:
            exp_stack.append(ch)
        elif ch == '(':
            op_stack.append(ch)
        elif ch == ')':
            while op_stack[-1] != '(':
                op = op_stack.pop()
                a = exp_stack.pop()
                b = exp_stack.pop()
                exp_stack.append( op+b+a )
            op_stack.pop()
        else:
            while op_stack and op_stack[-
1] != '(' and PRI[ch] <= PRI[op_stack[-1]]:
                op = op_stack.pop()
                a = exp_stack.pop()
                b = exp_stack.pop()
                exp_stack.append( op+b+a )
            op_stack.append(ch)

    while op_stack:
        op = op_stack.pop()
        a = exp_stack.pop()
        b = exp_stack.pop()
```

```
        exp_stack.append( op+b+a )
    print(f'{exp_stack[-1]}')
    return exp_stack[-1]

def three_add_code(pos):
    print("The Three Address Code Generation of expression ",expres," is::")
    exp_stack = []
    t = 1

    for i in pos:
        if i not in OPERATORS:
            exp_stack.append(i)
        else:
            print(f't{t} := {exp_stack[-2]} {i} {exp_stack[-1]}')
            exp_stack=exp_stack[:-2]
            exp_stack.append(f't{t}')
            t+=1

expres = input("INPUT THE EXPRESSION: ")
print("The Infix of the given expression is ::\n",expres)
print("The Prefix of ",expres," is::")
pre = infix_to_prefix(expres)
print("The Postfix of ",expres," is::")
pos = infix_to_postfix(expres)
three_add_code(pos)
```

Output

```
PS E:\TY\CD> & e:/TY/CD/venv/Scripts/python.exe "e:/TY/CD/Practical 9/prac_9_intermediate_code_generation.py"
INPUT THE EXPRESSION: (a+(b*c))/(a-(b*c))
The Infix of the given expression is ::
 (a+(b*c))/(a-(b*c))
The Prefix of  (a+(b*c))/(a-(b*c))  is::
/+a*bc-a*bc
The Postfix of  (a+(b*c))/(a-(b*c))  is::
abc*+abc*-/
The Three Address Code Generation of expression  (a+(b*c))/(a-(b*c))  is::
t1 := b * c
t2 := a + t1
t3 := b * c
t4 := a - t3
t5 := t2 / t4
```

Name: Varun Khadayate	Subject: Compiler Design
Roll No: A016	Date of Submission: 12th October 2021

Aim

Implement operator precedence parser/ LR Parser.

Lab Assignment Program

Implement operator precedence parser/ LR Parser.

Code

```
gram = {
    "S":["CC"],
    "C":["aC","d"]
}
start = "S"
terms = ["a","d","$"]

non_terms = []
for i in gram:
    non_terms.append(i)
gram["S'"]= [start]

new_row = {}
for i in terms+non_terms:
    new_row[i]=""

non_terms += ["S'"]
stateTable = []

def closure(term, I):
    if term in non_terms:
        for i in gram[term]:
            I+=[(term,"."+i)]
    I = list(set(I))
    for i in I:
        if "." != i[1][-1] and i[1][i[1].index(".")+1] in non_terms and
i[1][i[1].index(".")+1] != term:
            I += closure(i[1][i[1].index(".")+1], [])
    return I

Is = []
Is+=set(closure("S'", []))

print("\t\t\tGoto Steps")
countI = 0
omegaList = [set(Is)]
while countI<len(omegaList):
    newrow = dict(new_row)
```

```
    vars_in_I = []
    Is = omegaList[countI]
    countI+=1
    for i in Is:
        if i[1][-1]!=".":
            ind = i[1].index(".")
            vars_in_I+=[i[1][ind+1]]
    vars_in_I = list(set(vars_in_I))
    for i in vars_in_I:
        In = []
        for j in Is:
            if "."+i in j[1]:
                rep = j[1].replace("."+i,i+".")
                In+=[(j[0],rep)]
        if (In[0][1][-1]!="."):
            temp = set(closure(i,In))
            if temp not in omegaList:
                omegaList.append(temp)
            if i in non_terms:
                newrow[i] = str(omegaList.index(temp))
            else:
                newrow[i] = "s"+str(omegaList.index(temp))
            print(f'Goto(I{countI-1},{i}):{temp} That is
I{omegaList.index(temp)}')
        else:
            temp = set(In)
            if temp not in omegaList:
                omegaList.append(temp)
            if i in non_terms:
                newrow[i] = str(omegaList.index(temp))
            else:
                newrow[i] = "s"+str(omegaList.index(temp))
            print(f'Goto(I{countI-1},{i}):{temp} That is
I{omegaList.index(temp)}')

    stateTable.append(newrow)
print("\n\n\t\t\tList of I's")
for i in omegaList:
    print(f'I{omegaList.index(i)}: {i}')

I0 = []
for i in list(omegaList[0]):
    I0 += [i[1].replace(".","")]
print(I0)

for i in omegaList:
    for j in i:
        if "." in j[1][-1]:
```

```
            if j[1][-2]=="S":
                stateTable[omegaList.index(i)]["$"] = "Accept"
                break
            for k in terms:
                stateTable[omegaList.index(i)][k] =
"r"+str(I0.index(j[1].replace(".","")))
print("\n\t\t\tState Table")

print(f'{" ": <9}',end="")
for i in new_row:
    print(f'|{i: <11}',end="")

print(f'\n{"-":-<66}')
for i in stateTable:
    print(f'{"I("+str(stateTable.index(i))+")": <9}',end="")
    for j in i:
        print(f'|{i[j]: <10}',end=" ")
    print()
```

Output

```
PS E:\TY\CD> & e:/TY/CD/venv/Scripts/python.exe "e:/TY/CD/Practical 10/prac_10_LR_Parser.py"
                    Goto Steps
Goto(I0,a):{('C', 'a.C'), ('C', '.aC'), ('C', '.d')} That is I1
Goto(I0,d):{('C', 'd.')} That is I2
Goto(I0,S):{("S'", 'S.')} That is I3
Goto(I0,C):{('C', '.aC'), ('S', 'C.C'), ('C', '.d')} That is I4
Goto(I1,a):{('C', 'a.C'), ('C', '.aC'), ('C', '.d')} That is I1
Goto(I1,d):{('C', 'd.')} That is I2
Goto(I1,C):{('C', 'aC.')} That is I5
Goto(I4,a):{('C', 'a.C'), ('C', '.aC'), ('C', '.d')} That is I1
Goto(I4,d):{('C', 'd.')} That is I2
Goto(I4,C):{('S', 'CC.')} That is I6


                    List of I's
I0: {('C', '.aC'), ('S', '.CC'), ("S'", '.S'), ('C', '.d')}
I1: {('C', 'a.C'), ('C', '.aC'), ('C', '.d')}
I2: {('C', 'd.')}
I3: {("S'", 'S.')}
I4: {('C', '.aC'), ('S', 'C.C'), ('C', '.d')}
I5: {('C', 'aC.')}
I6: {('S', 'CC.')}
['aC', 'CC', 'S', 'd']

                    State Table
         |a          |d          |$          |S          |C
 ------------------------------------------------------------------
 I(0)    |s1         |s2         |           |3          |4
 I(1)    |s1         |s2         |           |           |5
 I(2)    |r3         |r3         |r3         |           |
 I(3)    |           |           |Accept     |           |
 I(4)    |s1         |s2         |           |           |6
 I(5)    |r0         |r0         |r0         |           |
 I(6)    |r1         |r1         |r1         |           |
PS E:\TY\CD>
```