Compiler Construction Lab CSL-323

PROJECT REPORT

"TEXT TO LEXIFY TRANSFORMER"

Tayyaba Imam (02-134202-056)
Mania Imam (02-134212-013)
Shavana Yousuf (02-134212-022)

Computer Science Department

Bahria University, Karachi Campus

Fall 2023

TEXT TO LEXIFY TRANSFORMER

ABSTRACT

Our project introduces "Lexify," an imperative programming language inspired by C++ and Python. Designed for modular programming rather than realistic production, Lexify emphasizes clarity, expanded scope, and simplified structure. The compilation process involves six layers, including a Lexical Analyzer, Parser, Scanner, Intermediate Code Generator, Code Optimization, and Code Generation.

Lexify enforces case sensitivity, necessitating precise typing of keywords, variables, and function names. The compilation process employs various stages to ensure accurate transformation, error checking, and constraint handling. The Symbol Table plays a vital role in storing relevant information at each stage, preventing errors from propagating. The Lexify compiler facilitates the breakdown of complex programs into manageable modules, fostering a systematic and structured approach to programming.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Background

This project addresses the need for a specialized programming language focusing on modular programming. "Lexify" is inspired by C++ and Python, with modifications for enhanced clarity and simplified structure. While not intended for production, Lexify serves educational purposes, allowing developers to break down complex programs into manageable modules. By enforcing case sensitivity and precise typing, Lexify promotes code reliability. The project emphasizes error detection and constraint handling in each compilation stage to ensure accurate output. This background sets the stage for exploring Lexify's design, implementation, and functionality in subsequent sections.

1.2 Aims and Objectives

The primary objective of the project is to align with its educational purpose and defined scope. The overarching aim is to create a user-friendly programming environment that prioritizes the development of coding skills over intricate syntax details. This objective is rooted in the belief that simplicity in learning is crucial for beginners, removing unnecessary barriers to entry.

The project seeks to empower programmers by providing a language, "Lexify," that facilitates easy modularization of code. This, in turn, fosters a clear and organized approach to programming. The emphasis on modularity and clarity is a key component of the project's broader goal — to offer an accessible and engaging learning experience for individuals embarking on their coding journey. By achieving these objectives, the project endeavors to contribute to the educational landscape of programming languages, making coding more approachable and comprehensible for learners.

1.3 Scope of Project

The project's scope is comprehensive, encompassing crucial aspects that define its objectives and impact. Firstly, the primary emphasis is on the design and implementation of "Lexify," a programming language introducing distinctive data types and syntax constructs. The

intentional structuring of Lexify aims to empower programmers to easily modularize their code, instilling good coding practices right from the beginning.

Moreover, the project is distinctly oriented toward education, targeting beginners and students entering the coding domain. By creating Lexify as a foundational tool for learning programming, the project seeks to contribute to accessible coding education. It aspires to bridge the gap between non-programmers and coding enthusiasts by providing a user-friendly environment that fosters a clear understanding of programming concepts. In doing so, the project aims to make a significant impact on coding education, facilitating an easier entry into the world of programming for individuals at the onset of their coding journey.

1.4 Environmental Aspects of Project

The project's environment is shaped by technological and contextual factors:

Technological Environment:

- Programming Tools: Choices impact Lexify's efficiency and functionality.
- Development Platforms: Consideration for cross-platform accessibility.
- Integration: Compatibility with existing technologies.

Contextual Environment:

- Educational Focus: Emphasis on simplicity for learning programming.
- User Demographics: Tailored for beginners and students.
- Community Engagement: Open-source principles and collaboration.

CHAPTER 2

DESIGN AND METHODOLOGY

2.1 Equipment / Materials Used

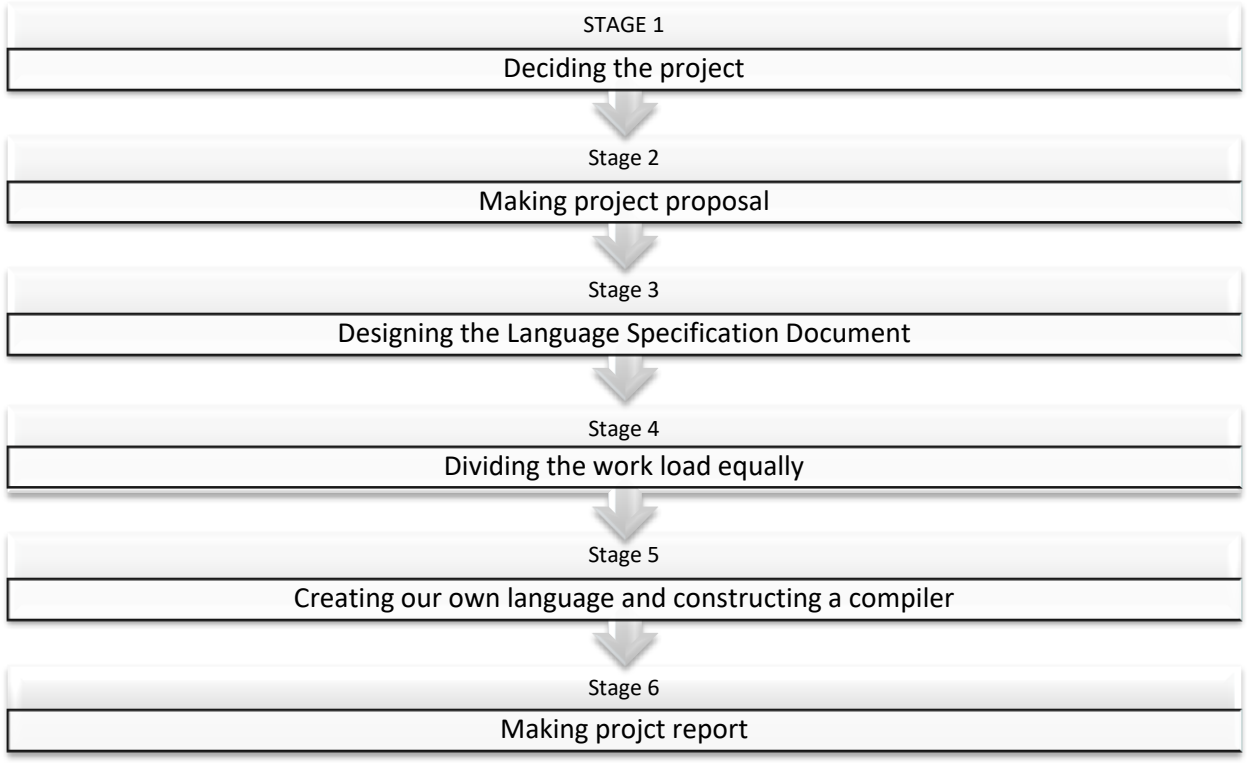
Development Software:

- PyCharm Community Edition for language design and compiler implementation.
- Compilers, interpreters, and debugging tools for Lexify development.

Programming Languages:

- Python

2.2 Flow Chart



CHAPTER 3

DESIGN IMPLMENTATION

3.1 Implementation

In the implementation phase, we begin by developing the Lexical Analyzer, the initial phase of our compiler. Utilizing Deterministic Finite Automata (DFA), we convert high-level input programs into a sequence of tokens, representing identifiers, keywords, and punctuation. This phase seamlessly integrates with subsequent stages.

Moving forward, our focus shifts to the Syntax Analyzer, receiving token streams from the Lexical Analyzer. It parses the source code against production rules, implementing error-recovering strategies to gracefully handle errors. This results in the generation of a parse tree, providing a structured representation of the source code.

Following the Syntax Analyzer, we proceed to the Semantic Analyzer. Leveraging the parse tree and the symbol table, we ensure semantic consistency with our language definition. Gathering and storing type information in the syntax tree or symbol table, this phase identifies and handles semantic errors like type mismatches and undeclared variables.

The integrated approach of Lexical, Syntax, and Semantic Analysis ensures a thorough examination of the source code, detecting and resolving errors at various levels. Continuous testing and refinement throughout these implementation phases guarantee the reliability and effectiveness of our compiler.

CHAPTER 4

RESULTS AND DISCUSSIONS

4.1 Results (CODE):

- **Lexer:**

```
import re
PN_Keywords=
"yap|dawg|sike|also|cap|facts|salty|base|func|con|bussin|purr|
or|ghost|rent-
free|girlboss|loop|gaslight|slay|iostream|reverse|main|chill|n
on|twocents|levels|fosho|case"
PN_Datatypes = "dec|num|calm|rizz|bin|girlboss|depression"
PN_ADDSUBOperators = "plus|minus"
PN_UnaryOperators = "inc|dec"
PN_DIVMULOperators = "mul|div|mod"
PN_RelationalOperators                                    =
"smallerthan|largerthan|notequal|equalequal"
PN_LogicalOperators = "and|or"
PN_AssignOperators                                        =
"plusequals|minusequals|divequals|modequals"
PN_EqualsOperator = "equals"
PN_Identifiers = "^[A-Z]+[a-zA-Z0-9$@#]*_"
tokenset = []


def tokenize():
    myfile = open("Lexify.txt", "r")
    count = 0
    patt = "##*"
    Output = open('LexifyOutput.txt', 'w')
```

```
    for inputline in myfile:
        if (re.search(patt, inputline)):
            continue
        else:

            Output.write(inputline)
    Output.close()
    Output = open('LexifyOutput.txt', 'r')
    while True:

        count = count + 1
        line = Output.readline()
        if not line:
            break

        WB = [' ', ',', '(', '{', '[', ']', '}', ')', ';', '=',
'+', '-', '*', ':', '>', '<', '\'', '\"', '.', '!', '|']
        i = 0
        wordHolder = ""
        tokens = []
        while (i < len(line) - 1):

            if line[i] in WB:
                # print(line[i] , line[i]=='=', not opHolder ,
wordHolder)
                if re.search(PN_Keywords, wordHolder):
                    tokens.append([wordHolder,      wordHolder,
count])
                elif re.search(PN_Identifiers, wordHolder):
                    tokens.append(['Identifier',    wordHolder,
count])
                elif  re.search(PN_ADDSUBOperators,  wordHolder):
```

```
                    tokens.append(['ADDSUB           Operator',
wordHolder, count])
                elif re.search(PN_UnaryOperators, wordHolder):
                    tokens.append(['Unary            Operator',
wordHolder, count])
                elif  re.search(PN_DIVMULOperators,  wordHolder):
                    tokens.append(['DIVMUL           Operator',
wordHolder, count])
                elif          re.search(PN_RelationalOperators,
wordHolder):
                    tokens.append(['Relational       Operator',
wordHolder, count])
                elif             re.search(PN_LogicalOperators,
wordHolder):
                    tokens.append(['Logical          Operator',
wordHolder, count])
                elif  re.search(PN_AssignOperators,  wordHolder):
                    tokens.append(['Assign           Operator',
wordHolder, count])
                elif re.search(PN_EqualsOperator, wordHolder):
                    tokens.append(['Equals           Operator',
wordHolder, count])
                elif re.search(PN_Datatypes, wordHolder):
                    tokens.append(['DT', wordHolder, count])

                if line[i] != ' ':
                    if line[i] == '=':
                        tokens.append(['AssignOP', line[i]])
                    if line[i] != '"' and line[i] != "'":
                        # Punctuator
                        tokens.append([line[i],        line[i],
count])
```

```
                wordHolder = ""

            else:
                # do nothing if the wordholder is empty and
the first value is an integer
                # check for integer value and not add it into
the wordholder
                if line[i] >= '0' and line[i] <= '9' and not
wordHolder:
                    k = 1
                else:
                    wordHolder += line[i]

                    # to check if any string const is being
read
            if i < len(line) and line[i] == '"' or line[i] ==
"'":
                tempStr = line[i]
                val = line[i]
                i += 1
                found = False
                while (i < len(line)):
                    if (line[i] == val):
                        tempStr += line[i]
                        found = True
                        break
                    tempStr += line[i]
                    i += 1
                if found:
                    tokens.append(['String_const',     tempStr,
count])
```

```
            # to check if any int/float const is being read
            if not wordHolder and i < len(line) and line[i] >=
'0' and line[i] <= '9':
                isFloat = False
                tempVal = line[i]
                while ((i + 1 < len(line) and (line[i + 1] >=
'0' and line[i + 1] <= '9')) or (
                        line[i + 1] == '.' and (i + 2 <
len(line) and line[i + 2] >= '0' and line[i + 2] <= '9'))):
                    if line[i + 1] == '.':
                        isFloat = True
                    tempVal += line[i + 1]
                    i += 1
                if isFloat:
                    tokens.append(['Float_const',      tempVal,
count])
                else:
                    tokens.append(['int_const',        tempVal,
count])
            i += 1

        if wordHolder:
            if re.search(PN_Keywords, wordHolder):
                tokens.append([wordHolder, wordHolder, count])
            elif re.search(PN_Identifiers, wordHolder):
                tokens.append(['Identifier',        wordHolder,
count])
            elif re.search(PN_ADDSUBOperators, wordHolder):
                tokens.append(['ADDSUB  Operator',  wordHolder,
count])
            elif re.search(PN_UnaryOperators, wordHolder):
```

```
                tokens.append(['Unary   Operator',   wordHolder,
count])
            elif re.search(PN_DIVMULOperators, wordHolder):
                tokens.append(['DIVMUL  Operator',  wordHolder,
count])
            elif  re.search(PN_RelationalOperators,  wordHolder):
                tokens.append(['Relational           Operator',
wordHolder, count])
            elif re.search(PN_LogicalOperators, wordHolder):
                tokens.append(['Logical  Operator',  wordHolder,
count])
            elif re.search(PN_AssignOperators, wordHolder):
                tokens.append(['Assign  Operator',  wordHolder,
count])
            elif re.search(PN_EqualsOperator, wordHolder):
                tokens.append(['Equals  Operator',  wordHolder,
count])
            elif re.search(PN_Datatypes, wordHolder):
                tokens.append(['DT', wordHolder, count])
            else:
                print("Lexical Error @ line", count, " invalid
ID")

        tokenset.append(tokens)
        print(tokens)
    Output.close()


print("\tTOKENIZE")
tokenize()
Output = open('TOKENS_output.txt', 'w')
for token in tokenset:
```

```
        Output.write("{token1}\n".format(token1=token))
    Output.close()
    tk = []
    print("\nShow all Class Part")
    PNClasspart = open('LexifyClassParts.txt','w')
    for x in range(len(tokenset)):
        tk = (tokenset[x])
        x += 1
        for i in range(len(tk)):
            PNClasspart.write(tk[i][0] + "\n")
```

- **Syntax:**

```
import re
output_file_name = 'CFG_output.txt'
file = open('LexifyClassParts.txt','r')
file = file.read()
string = file
DataType = ["rizz", "num", "dec", "calm", "DT"]
Keyword = ["iostream","insert","for", "key", "if", "switch",
"case", "ret", "while", "print", "cout"]
Relational = ["Relational_Operator"]
const = ["st_const","deci_const", "digit_const"]
Arithmetic=["ADDSUB_operator"]
loops = ["aslongas","since","act"]
incre = ["++", "--"]
Iden=["Identifier"]
Equals = ["equals"]
Punctuators = ["(", ")", "{", "}", "!", "|",","]
def parseTree(value):
    if value in Arithmetic:
        return "Arithmetic OP"
    if value in Punctuators:
```

```
            return value
        if value in loops:
            return value
        if value in DataType:
            return "DataType"
        if value in Relational:
            return "Relational Op"
        if value in incre:
            return "Incre Op"
        if value in Keyword:
            return value
        if value in Iden:
            return "ID"
        if value in Equals:
            return "equals"
        if value in const:
            return value
        if (value == "-"):
            return "Terminator"
        else:
            return "Error"
stri = string.split();
print(stri)
print('\n')
cfg = []
for char in stri:
    cfg.append(parseTree(char))
print(cfg)
with open(output_file_name, 'w') as output_file:
    output_file.write('\n'.join(cfg))

print(f"CFGs saved to {output_file_name}")
```

- **Recursive Descent Parser:**

```
def read_fsa_table(filename_list):
    all_fsa_table=[]
    all_fsa_char=[]
    for x in filename_list:
        file=open(x,"r")
        lines=file.readlines()
        file.close()
        lines=[ x.replace("\n","").split('\t') for x in lines]
        characters={ x[1] for x in lines}
        all_fsa_table.append(lines)
        all_fsa_char.append(characters)
    return all_fsa_table,all_fsa_char

def read_input(filename):
    file=open(filename)
    data=file.readline()
    file.close()
    return data.split()

def
fsm_in_parallel(all_lines,all_characters,input_list,e_states  ,
s_state):
    lis=[]
    seprators={"=","+",'minus',"*","&","!","|"}

keywords=['num','dec','calm','rizz','bin','girlboss','loop','f
osho','dawg','con','gaslight','non','sike','bussin','slay','bo
ujee','drip,''purr','depression','basic','also','or','reverse'
,'levels','cap','facts','exclude','chill','salty','ghost','yap
','twocents','base','rent-
```

```
free','vibe','woke','plus','minus','mul','divmod','smallerthan
','largerthan','equalsset','girlboss','slay']
    fsa_acceptance=[]
    for j in input_list:
        print("Processing String: ",j)
        input_lexeme_list=list(j)
        current_states=s_state
        fsa_acceptance=[ True for x in range(len(all_lines))]
        seprator_flag=False
        for index,char in enumerate(input_lexeme_list):
            if seprator_flag:
                seprator_flag=False
            if char in seprators:
                try:
                    if input_lexeme_list[index+1] in seprators:
                        print(f"     ------          Seperator:
{char}{input_lexeme_list[index+1]} ------ ")
                        seprator_flag=True

lis.append(f"{char}{input_lexeme_list[index+1]}")
                        break
                    else:
                        print(" ------ Seperator:",char,"   ---
--- ")
                        lis.append(char)
                        seprator_flag=True
                        break
                except:
                    print(" ------ Seperator:",char,"   ------
")
                    seprator_flag=True
                    lis.append(char)
```

```
                    break

            for fsa in range(len(all_lines)):

                if char not in all_characters[fsa] :
                    fsa_acceptance[fsa]=False

                if fsa_acceptance[fsa]==False:
                    continue
                print(f"{fsa+1}         current         state:
",current_states[fsa])
                print("input processing: ",char)
                next_states=set()
                for current_state in current_states[fsa]:

                    for y in all_lines[fsa]:

                        if(y[0]==current_state and y[1]==char):
                            next_states.add(y[2])

                print("next states: ", next_states,"\n...")
                current_states[fsa]=next_states
        if seprator_flag==True:
            continue
        for i in range(len(all_lines)):
            if fsa_acceptance[i]==False:
                print("-----")
            else:
                isAccepted=False
                for x in current_states[i]:
                    if x in e_states[i]:
                        isAccepted=True
```

```
                        break
                if i==0 and isAccepted:
                    if j in keywords:
                        print("----- Keyword! string accepted!
------")
                        lis.append("k")
                    else:
                        print(f"  ------  {i+1}    It  is  an
identifer ------ ")
                        lis.append("i")
                    continue

                print("------")
    return lis


files=["identifier.txt","integer.txt"]
all_lines,all_characters=read_fsa_table(files)
input_list=read_input("input.txt")
end_states=[["1"],["1"]]
start_states=[{"0"},{"0"}]

data=fsm_in_parallel(all_lines,all_characters,input_list,end_s
tates,start_states)

input=""
for x in data:
    if (x=="i" or x=="+"):
        input=input+x
input=input+"$"
print(input)
```

```
# input = "ii$" #$ is end of input,,
i=0
charc = input[i] # we will start with first charcter
rval = True # This is global flag to check whether parsing was succeed
# we are declaring input as global, so that it is accessable to all functions

# This function will match input charatcer with Grammar Charatcer
def match(var): # here charc point to the charcter, that need to be matched
    global i
    global input
    global charc
    global rval
    if(var == input[i] and var != "$"):
        # if the character is matched with string character, but it should not be the end character
         i=i+1  # You need to go to next character
         charc =  input[i]
         rval = True
         return rval #Its Fine Go Ahead,,as no error is their
    else:
        #print("\n There is an Error in the input string")
        rval = False
        return rval  # It means their is an Error, You need to quite

def E(): #This function implemnts E non-terminal
    global i
    global input
```

```
    global charc
    global rval
    if( charc == "i"):
        rval = match("i")
        if rval and charc !="$": #It means its True and we are
not at end of input
            rval = E_bar()  # Then proceed further in parsing
        else:
            return rval
    else:
        return False
    return rval

def E_bar():
    global i
    global input
    global charc
    global rval
    if( charc == "+"):
        rval = match("+")
        #rval = match("i")
        if rval: # then match next i
            rval = match("i")
            if rval and charc != "$": # we are not at end of
input
                rval = E_bar()
            else:
                return rval
        else:
            return rval # return rvale

    else:
```

```
        rval = False
        return rval # simply return to previous function in
recusrrion
    return rval

if __name__ == "__main__": # This is the main Function
    #global rval
    rval = E()
    if( rval): # we reached end of parsing, and still rval is
True
        print("\n Parsing Succeed")
        print("\n End---")
```

- **Shift Reduce Parser:**

```
gram = {
    "E":["2E2","3E3","4"]
}
starting_terminal = "E"
inp = "2324232$"
"""
# example 2
"23233332$"
gram = {
    "S":["S+S","S*S","i"]
}
starting_terminal = "S"
inp = "i+i*i"
"""
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
```

- **Type Checking:**

```
import re
```

```
print("\n\t\t\t\t\t**********************************************
**********************")
print ("\t\t\t\t\t\t\t\tTYPE CHECKING")
print("\t\t\t\t\t**********************************************
******************\n")
inp = input("\n\nEnter your input: ")
dt = ["num", "calm", "rizz","dec"]
tok = []
tok = inp.split(" ")
print(tok, "\n")

def checkk(typee, value):
    if typee == "digit" and re.match("^[0-9]+$", value):
        return True
    elif typee == "st" and re.match('^"[\w]+"$', value):
        return True
    elif  typee  ==  "ascii"  and  re.match("^'[A-Za-z0-9]'$",
value):
        return True
    else:
        return False

def main():
    error = False
    flag = True
    i = 0
    namee = ""
    tipe = ""
    while i < len(tok):
        if tok[i] in dt:
            flag = True
```

```
            while i < len(tok) and flag == True:
                if tok[i] in dt:
                    tipe = tok[i]
                    i += 1

                else:
                    print("ERROR:  Datatype  expected  but  got
something else")
                    error = True
                    i += 1

                if  re.match("^[A-Z]+[a-zA-Z0-9$@#]*_",  tok[i]):
                    namee = tok[i]
                    i += 1

                else:
                    print("ERROR: Invalid Variable Name")
                    error = True
                    i += 1

                if tok[i] == "equals":
                    i += 1

                else:
                    print("ERROR:  'equals'  expected  but  got
something else")
                    error = True
                    i += 1

                if checkk(tipe, tok[i]):
                    tipe = ""
                    i += 1
```

```
                    print("No Error")

                else:
                    print("ERROR:",   namee,   "'s   Datatype
Mismatch")
                    error = True
                    i += 1

                if tok[i] == "-" or tok[i] == "":
                    i + 1
                    flag = False

                else:
                    print("ERROR:",   namee,   "'s   Terminator
missing")
                    error = True
                    i += 1

    else:
        i += 1

main()
```

- **Symbol Table:**

```
import re
from beautifultable import BeautifulTable

mySymbolTable = BeautifulTable()
tokens = []
symbolTable = []
```

```
class Token:
    lineNo = 0
    valuePart = '-'
    classPart = '-'


class SymbolTable:
    scope = 0
    classPart = '-'
    valuePart = '-'

    def __init__(self, scope, classPart, valuePart):
        self.scope = scope
        self.classPart = classPart
        self.valuePart = valuePart


class LexicalAnalyser:
    def __init__(self):
        wordBreakers = [' ', ',', '(', '{', '[', ']', '}', ')',
';', '=', '+', '-', '*', ':', '>', '<', '\'', '\"', '.',
                        '!', '|', ]
        code = open('L0.txt', 'r')
        lineNo = 1
        while True:
            i = 0
            fileLine = code.readline()
            if not fileLine:
                break
            length = 1 if len(fileLine) == 1 else len(fileLine)
- 1
            word = ""
```

```
            while i < length:
                if fileLine[i] in wordBreakers:
                    wordBreaker = fileLine[i]
                    if len(word) <= 1:
                        if (i + 1 < length) and ((fileLine[i]
== "plus" and fileLine[i + 1] == "plus")
                                                 or
(fileLine[i] == "minus" and fileLine[i + 1] == "minus")
                                                 or
(fileLine[i] == "=" and fileLine[i + 1] == "=")
                                                 or ((

fileLine[i] == "mul"

or fileLine[i] == "div"

or fileLine[i] == "plus"

or fileLine[i] == "minus"
                                                     )
                                                     and
fileLine[i + 1] == "equals"
                                                 )
                        ):
                            wordBreakers += fileLine[i + 1]
                            i += 1
                    if word != "":
                        self.createToken(word, lineNo)
                    self.createToken(wordBreaker, lineNo)
                    word = ""
                else:
                    word += fileLine[i]
```

```
                i += 1
            lineNo += 1

    def createToken(self, word, lineNo):
        hasError = False
        PN_Datatypes                                        =
"dec|num|calm|rizz|bin|girlboss|depression"
        PN_ADDSUBOperators = "plus|minus"
        PN_UnaryOperators = "inc|dec"
        PN_DIVMULOperators = "mul|div|mod"
        PN_RelationalOperators                              =
"smallerthan|largerthan|notequal|equalequal"
        PN_LogicalOperators = "and|or"
        PN_AssignOperators                                  =
"plusequals|minusequals|divequals|modequals"
        PN_EqualsOperator = "="
        PN_Punctuators = ["(", "{", "[", ")", "}", "]", ",",
"-", "|", "!", ""]
        PN_Identifiers = "^[A-Z]+[a-zA-Z0-9$@#]*_"

        token = Token()
        if word in PN_Datatypes:
            token.classPart = "DT"
        elif word in PN_Punctuators:
            token.classPart = word
        elif word in PN_UnaryOperators:
            token.classPart = "INCDEC"
        elif word in PN_AssignOperators:
            token.classPart = "AO"
        elif word in PN_ADDSUBOperators:
            token.classPart = "ADDSUB"
        elif word in PN_DIVMULOperators:
```

```
                token.classPart = "DIVMOD"
            elif word in PN_RelationalOperators:
                token.classPart = "RO"
            elif re.search(PN_Identifiers, word):
                token.classPart = "ID"
            elif self.isValidConstant(word, lineNo):
                token.classPart = "const"
            elif word == " ":
                return
            token.lineNo = lineNo
            token.valuePart = word
            tokens.append(token)

        def isValidConstant(self, word, lineNo):
            digit_const = r"^[+-]?\d+$"
            deci_const = r"^[+-]?\d+(\.\d+)?$"
            ascii_const = r"^(\\.|[ ^\\'])'$"
            st_const = r'^"(\\.|[ ^\\"])*"$'
            if re.match(digit_const, word):
                return True
            elif re.match(deci_const, word):
                return True
            elif re.match(ascii_const, word):
                return True
            elif re.match(st_const, word):
                return True
            else:
                return False


    if __name__ == "__main__":
        analyser = LexicalAnalyser()
```

```
print("\t\t\t\t\t**********************************************
******************")
    print("\t\t\t\t\t\t\t\tSYMBOL TABLE")

print("\t\t\t\t\t**********************************************
******************\n")
    mySymbolTable.append_column("Scope", [])
    mySymbolTable.append_column("Type", [])
    mySymbolTable.append_column("ID", [])
    scope = 0
    type = ""

    for val in tokens:

        if val.valuePart == "{":
            scope += 1
        if val.classPart == "DT":
            type = val.valuePart
        if val.classPart == "ID" and type != "":
            table = SymbolTable(scope, val.valuePart, type)
            type = ""
            symbolTable.append(table)
        if val.valuePart == "}":
            scope -= 1
        if scope < 0:
            scope = 0
    for item in symbolTable:
        mySymbolTable.append_row([item.scope,   item.valuePart,
item.classPart])
print(mySymbolTable)
```

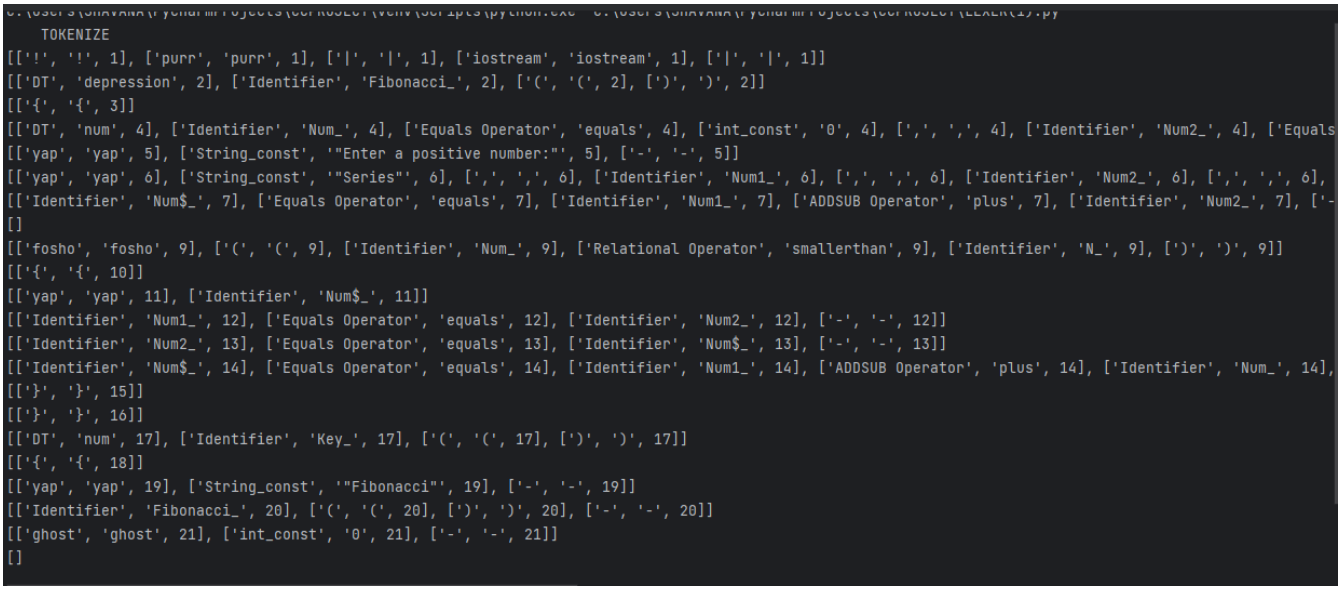
- **Intermediate Code Generator:**

```
# Token list
tokens = [
    ['!', '!', 1], ['purr', 'purr', 1], ['|', '|', 1],
['iostream', 'iostream', 1], ['|', '|', 1],
    ['DT', 'depression', 2], ['Identifier', 'Fibonacci_', 2],
['(', '(', 2], [')', ')', 2],
    ['{', '{', 3],
    ['DT', 'num', 4], ['Identifier', 'Num_', 4], ['Equals
Operator', 'equals', 4], ['int_const', '0', 4], [',', ',', 4],
    ['Identifier', 'Num2_', 4], ['Equals Operator', 'equals',
4], ['int_const', '1', 4], [',', ',', 4],
    ['Identifier', 'Num$_', 4], ['Equals Operator', 'equals',
4], ['int_const', '0', 4], [',', ',', 4],
    ['Identifier', 'N_', 4], ['-', '-', 4],
    ['yap', 'yap', 5], ['String_const', '"Enter a positive
number:"', 5], ['-', '-', 5],
    ['yap', 'yap', 6], ['String_const', '"Series"', 6], [',',
',', 6], ['Identifier', 'Num1_', 6], [',', ',', 6],
    ['Identifier', 'Num2_', 6], [',', ',', 6], ['String_const',
'",", 6], ['-', '-', 6],
    ['Identifier', 'Num$_', 7], ['Equals Operator', 'equals',
7], ['Identifier', 'Num1_', 7], ['ADDSUB Operator', 'plus', 7],
    ['Identifier', 'Num2_', 7], ['-', '-', 7],
    [],
    ['fosho', 'fosho', 9], ['(', '(', 9], ['Identifier',
'Num_', 9], ['Relational Operator', 'smallerthan', 9],
    ['Identifier', 'N_', 9], [')', ')', 9],
    ['{', '{', 10],
    ['yap', 'yap', 11], ['Identifier', 'Num$_', 11],
    ['Identifier', 'Num1_', 12], ['Equals Operator', 'equals',
12], ['Identifier', 'Num2_', 12], ['-', '-', 12],
```

```
    ['Identifier', 'Num2_', 13], ['Equals Operator', 'equals',
13], ['Identifier', 'Num$_', 13], ['-', '-', 13],
    ['Identifier', 'Num$_', 14], ['Equals Operator', 'equals',
14], ['Identifier', 'Num1_', 14],
    ['ADDSUB Operator', 'plus', 14], ['Identifier', 'Num_',
14], ['-', '-', 14],
    ['}', '}', 15],
    ['}', '}', 16],
    ['DT', 'num', 17], ['Identifier', 'Key_', 17], ['(', '(',
17], [')', ')', 17],
    ['{', '{', 18],
    ['yap', 'yap', 19], ['String_const', '"Fibonacci"', 19],
['-', '-', 19],
    ['Identifier', 'Fibonacci_', 20], ['(', '(', 20], [')',
')', 20], ['-', '-', 20],
    ['ghost', 'ghost', 21], ['int_const', '0', 21], ['-', '-',
21],
    []
]

def generate_intermediate_code(tokens):
    intermediate_code = []
    for token in tokens:

        if token:
            # Extract relevant information from the token
            token_type = token[0]
            token_value = token[1]
            line_number = token[2]

            if token_type == 'Identifier':
```

```
                intermediate_code.append(f'{token_value}      =
{line_number}')
            elif token_type == 'ADDSUB Operator':
                intermediate_code.append(f'{token_value}      =
{line_number}')
            elif token_type == 'Relational Operator':
                intermediate_code.append(f'if {line_number}:')
            elif token_type == 'String_const':

intermediate_code.append(f'print({token_value})')
            elif token_type == 'yap':

intermediate_code.append(f'print({token_value})')
            elif token_type == '}':
                intermediate_code.append('}')
print("Intermediate Code Generated:")
result = generate_intermediate_code(tokens)
for line in result:
    print(line)
```

4.2 Screenshots of the system:



```
['!', 'purr', '|', 'iostream', '|', 'DT', 'Identifier', '(', ')', '{', 'DT', 'Identifier', 'Equals', 'Operator', 'int_const', ',', 'Identifier', 'Equals',


['!', 'Error', '|', 'iostream', '|', 'DataType', 'ID', '(', ')', '{', 'DataType', 'ID', 'Error', 'Error', 'Error', ',', 'ID', 'Error', 'Error', 'Error', ',
CFGs saved to CFG_output.txt
```

```
Processing String:  =
 ------ Seperator: =   ------
Processing String:  +
 ------ Seperator: +   ------
Processing String:  equals
1 current state:  {'0'}
input processing:  e
next states:  {'1'}
...
1 current state:  {'1'}
input processing:  q
next states:  {'1'}
...
1 current state:  {'1'}
input processing:  u
next states:  {'1'}
...
1 current state:  {'1'}
input processing:  a
next states:  {'1'}
...
1 current state:  {'1'}
input processing:  l
next states:  {'1'}
```

```
...
 ------ 1  It is an identifer ---
-----
Processing String:  &
 ------ Seperator: &   ------
Processing String:  a
1 current state:  {'1'}
input processing:  a
next states:  {'1'}
...
 ------ 1  It is an identifer ---
-----
Processing String:  +
 ------ Seperator: +   ------
Processing String:  fosho
1 current state:  {'1'}
input processing:  f
next states:  {'1'}
...
1 current state:  {'1'}
input processing:  o
next states:  {'1'}
```

```
1 current state:  {'1'}
input processing:  o
next states:  {'1'}
...
----- Keyword! string accepted! ------
-----
+ii+$

Process finished with exit code 0
```

```
Stack          |Input Buffer   |Parsing Action
-----------------------------------------------
$2             |324232$        |Shift
$23            |24232$         |Shift
$232           |4232$          |Shift
$2324          |232$           |Shift
$232E          |232$           |Reduce S->4
$232E2         |32$            |Shift
$23E           |32$            |Reduce S->2E2
$23E3          |2$             |Shift
$2E            |2$             |Reduce S->3E3
$2E2           |$              |Shift
$E             |$              |Reduce S->2E2
$E             |$              |Accepted

Process finished with exit code 0
```

```
                ****************************************************************
                            TYPE CHECKING
                ****************************************************************



Enter your input: riz_ equals 5
['riz_', 'equals', '5']
```

```
                    ************************************************
                                    SYMBOL TABLE
                    ************************************************

C:\Users\SHAVANA\PycharmProjects\CCPROJECT\venv\Lib\site-packages\be
  warnings.warn(message, FutureWarning)
C:\Users\SHAVANA\PycharmProjects\CCPROJECT\venv\Lib\site-packages\be
  warnings.warn(message, FutureWarning)
+-------+------------+------------+
| Scope |    Type    |     ID     |
+-------+------------+------------+
|   0   | depression | Fibonacci_ |
+-------+------------+------------+
|   1   |    num     |   Num1_    |
+-------+------------+------------+
|   1   |    num     |   Num2_    |
+-------+------------+------------+
|   1   |    num     |   Num$_    |
+-------+------------+------------+
|   1   |    num     |     N_     |
+-------+------------+------------+
|   0   |    num     |    Key_    |
+-------+------------+------------+
```

```
Intermediate Code Generated:
Fibonacci_ = 2
Num_ = 4
Num2_ = 4
Num$_ = 4
N_ = 4
print(yap)
print("Enter a positive number:")
print(yap)
print("Series")
Num1_ = 6
Num2_ = 6
print(",")
Num$_ = 7
Num1_ = 7
plus = 7
Num2_ = 7
Num_ = 9
if 9:
N_ = 9
print(yap)
Num$_ = 11
Num1_ = 12
Num2_ = 12
```

```
Num$_ = 11
Num1_ = 12
Num2_ = 12
Num2_ = 13
Num$_ = 13
Num$_ = 14
Num1_ = 14
plus = 14
Num_ = 14
}
}
Key_ = 17
print(yap)
print("Fibonacci")
Fibonacci_ = 20

Process finished with exit code 0
```

TOKENS_output - Notepad

File Edit Format View Help

```
[['!', '!', 1], ['purr', 'purr', 1], ['|', '|', 1], ['iostream', 'iostream', 1], ['|', '|', 1]]
[['DT', 'depression', 2], ['Identifier', 'Fibonacci_', 2], ['(', '(', 2], [')', ')', 2]]
[['{', '{', 3]]
[['DT', 'num', 4], ['Identifier', 'Num_', 4], ['Equals Operator', 'equals', 4], ['int_const', '0', 4], [',', ',', 4], ['Identifier', 'Num2_', 4], ['Equals Operator', 'equals', 4],
['int_const', '1', 4], [',', ',', 4], ['Identifier', 'Num$_', 4], ['Equals Operator', 'equals', 4], ['int_const', '0', 4], [',', ',', 4], ['Identifier', 'N_', 4], ['-', '-', 4]]
[['yap', 'yap', 5], ['String_const', '"Enter a positive number:"', 5], ['-', '-', 5]]
[['yap', 'yap', 6], ['String_const', '"Series"', 6], [',', ',', 6], ['Identifier', 'Num1_', 6], [',', ',', 6], ['Identifier', 'Num2_', 6], [',', ',', 6], ['String_const', '",","', 6], ['-',
'-', 6]]
[['Identifier', 'Num$_', 7], ['Equals Operator', 'equals', 7], ['Identifier', 'Num1_', 7], ['ADDSUB Operator', 'plus', 7], ['Identifier', 'Num2_', 7], ['-', '-', 7]]
[]
[['fosho', 'fosho', 9], ['(', '(', 9], ['Identifier', 'Num_', 9], ['Relational Operator', 'smallerthan', 9], ['Identifier', 'N_', 9], [')', ')', 9]]
[['{', '{', 10]]
[['yap', 'yap', 11], ['Identifier', 'Num$_', 11]]
[['Identifier', 'Num1_', 12], ['Equals Operator', 'equals', 12], ['Identifier', 'Num2_', 12], ['-', '-', 12]]
[['Identifier', 'Num2_', 13], ['Equals Operator', 'equals', 13], ['Identifier', 'Num$_', 13], ['-', '-', 13]]
[['Identifier', 'Num$_', 14], ['Equals Operator', 'equals', 14], ['Identifier', 'Num1_', 14], ['ADDSUB Operator', 'plus', 14], ['Identifier', 'Num_', 14], ['-', '-', 14]]
[['}', '}', 15]]
[['}', '}', 16]]
[['DT', 'num', 17], ['Identifier', 'Key_', 17], ['(', '(', 17], [')', ')', 17]]
[['{', '{', 18]]
[['yap', 'yap', 19], ['String_const', '"Fibonacci"', 19], ['-', '-', 19]]
[['Identifier', 'Fibonacci_', 20], ['(', '(', 20], [')', ')', 20], ['-', '-', 20]]
[['ghost', 'ghost', 21], ['int_const', '0', 21], ['-', '-', 21]]
[]
```

LexifyClassParts - Notepad

File Edit Format View Help

```
!
purr
|
iostream
|
DT
Identifier
(
)
{
DT
Identifier
Equals Operator
int_const
,
Identifier
Equals Operator
int_const
,
Identifier
Equals Operator
int_const
,
Identifier
-
yap
String_const
-
yap
String_const
,
Identifier
,
Identifier
,
String_const
-
Identifier
Equals Operator
Identifier
ADDSUB Operator
```