**Lexical Analysis Code**

# Define the token types

INT = 'INT'

PLUS = 'PLUS'

EQ = 'EQ'

MINUS = 'MINUS'

MULT = 'MULT'

DIV = 'DIV'

LPAREN = 'LPAREN'

RPAREN = 'RPAREN'

IDF = 'IDF'

EOF = 'EOF'

# Define a dictionary of keywords

KEYWORDS = {

    'if': 'IF',

    'else': 'ELSE',

    'while': 'WHILE',

    'print': 'PRINT',

}

# Define a function to tokenize the input string

def tokenize(input\_string):

    tokens = []

    pos = 0

    while pos < len(input\_string):

        current\_char = input\_string[pos]

        # Integer token

        if current\_char.isdigit():

            value = ''

            while pos < len(input\_string) and input\_string[pos].isdigit():

                value += input\_string[pos]

                pos += 1

            tokens.append(('INT', int(value)))

        # Identifier or keyword token

        elif current\_char.isalpha() or current\_char == '\_':

            value = ''

            while pos < len(input\_string) and (input\_string[pos].isalnum() or input\_string[pos] == '\_'):

                value += input\_string[pos]

                pos += 1

            token\_type = KEYWORDS.get(value, IDF)

            tokens.append((token\_type, value))

        # Operator tokens

        elif current\_char == '+':

            tokens.append((PLUS, current\_char))

            pos += 1

        elif current\_char == '-':

            tokens.append((MINUS, current\_char))

            pos += 1

        elif current\_char == '\*':

            tokens.append((MULT, current\_char))

            pos += 1

        elif current\_char == '/':

            tokens.append((DIV, current\_char))

            pos += 1

        elif current\_char == '(':

            tokens.append((LPAREN, current\_char))

            pos += 1

        elif current\_char == ')':

            tokens.append((RPAREN, current\_char))

            pos += 1

        elif current\_char == '=':

            tokens.append((EQ, current\_char))

            pos += 1

        # Ignore whitespace

        elif current\_char.isspace():

            pos += 1

        # Invalid input

        else:

            print(f"Invalid input: {current\_char}")

            return []

    tokens.append((EOF, None))

    return tokens

# Test the tokenizer

data = '''

sum = 2 + 3 - 5

result = sum \* 4

if( x = 9)

'''

tokens = tokenize(data)

print(tokens)

**Output :-**

[('IDF', 'sum'), ('EQ', '='), ('INT', 2), ('PLUS', '+'), ('INT', 3), ('MINUS', '-'), ('INT', 5), ('IDF', 'result'), ('EQ', '='), ('IDF', 'sum'), ('MULT', '\*'), ('INT', 4), ('IF',

'if'), ('LPAREN', '('), ('IDF', 'x'), ('EQ', '='), ('INT', 9), ('RPAREN', ')'), ('EOF', None)]

**Symbol Table Code**

# Define a dictionary to store the symbol table entries

symbol\_table = {}

# Define data type bytes

data\_types = {"int": 4, "char": 1, "bool": 2, "float": 4}

# Define a function to add a new entry to the symbol table

def add\_entry(

    name, type, object\_address, dimension\_num, line\_declaration, line\_references

):

    symbol\_table[name] = {

        "Type": type,

        "Object Address": object\_address,

        "Dimension Num": dimension\_num,

        "Line Declaration": line\_declaration,

        "Line References": line\_references,

    }

# Define a function to parse the input code and generate the symbol table

def parse\_code(input\_code):

    lines = input\_code.split("\n")

    current\_line = 1

    current\_address = 0

    for line in lines:

        words = line.split()

        for i, word in enumerate(words):

            if word == "int" or word == "float" or word == "bool" or word == "char":

                # Found a variable declaration

                name = words[i + 1]

                type = word

                object\_address = current\_address

                dimension\_num = 0

                line\_declaration = current\_line

                line\_references = [current\_line]

                add\_entry(

                    name,

                    type,

                    object\_address,

                    dimension\_num,

                    line\_declaration,

                    line\_references,

                )

                typeValue = data\_types[word]

                current\_address += typeValue

                if (

                    len(words) > i + 2

                    and words[i + 2].startswith("[")

                    and words[i + 2].endswith("]")

                ):

                    # Found an array declaration

                    typeValue = data\_types[word]

                    dimension\_str = words[i + 2][1:-1]

                    dimension\_num = len(dimension\_str.split(","))

                    current\_address += typeValue \* dimension\_num

            elif word in symbol\_table:

                # Found a variable reference

                symbol\_table[word]["Line References"].append(current\_line)

        current\_line += 1

# Test the code with the input example and print out the resulting symbol table

input\_code = """

int arr[3,8,5];

float y;

bool z;

arr[0] = 1;

arr[1] = 2;

arr[2] = 3;

char m;

float x = arr[0] + arr[1];

if (x > y) {

    z = true;

} else {

    z = false;

}

int result = x \* arr[2];

for (int i = 0; i < result; i++) {

    print(i);

}

"""

parse\_code(input\_code)

# Print out the resulting symbol table in table format

print(

    "| {:<16} | {:<16} | {:<16} | {:<16} | {:<16} | {:<16} |".format(

        "Name",

        "Type",

        "Object Address",

        "Dimension Num",

        "Line Declaration",

        "Line References",

    )

)

print(

    "|------------------|------------------|------------------|------------------|------------------|------------------|"

)

for name, entry in symbol\_table.items():

    type = entry["Type"]

    object\_address = entry["Object Address"]

    dimension\_num = entry["Dimension Num"]

    line\_declaration = entry["Line Declaration"]

    line\_references = ", ".join(map(str, entry["Line References"]))

    print(

        "| {:<16} | {:<16} | {:<16} | {:<16} | {:<16} | {:<16} |".format(

            name, type, object\_address, dimension\_num, line\_declaration, line\_references

        )

    )

**Parse Tree Code**

from lark import Lark, Tree

grammar = """

    start: expr

    expr: atom | expr "+" atom

    atom: NUMBER | "(" expr ")"

    %import common.NUMBER

    %import common.WS

    %ignore WS

"""

def print\_tree(tree, level=0):

    print("  " \* level + tree.data)

    for child in tree.children:

        if isinstance(child, Tree):

            print\_tree(child, level=level + 1)

        else:

            print("  " \* (level + 1) + child)

parser = Lark(grammar)

input\_str = "3 + (4 + 5)"

parse\_tree = parser.parse(input\_str)

print\_tree(parse\_tree)

**Parse Table Code**

def calculate\_first(grammar):

    first = {non\_terminal: set() for non\_terminal in grammar}

    while True:

        updated = False

        for non\_terminal, productions in grammar.items():

            for production in productions:

                first\_set = {production[0]} if production[0] not in grammar else first[production[0]]

                first\_set -= {'epsilon'}

                for symbol in production[1:]:

                    if symbol not in grammar:

                        break

                    first\_set |= first[symbol] - {'epsilon'}

                else:

                    if 'epsilon' in first\_set:

                        first\_set |= {'epsilon'}

                if first\_set - first[non\_terminal]:

                    first[non\_terminal] |= first\_set

                    updated = True

        if not updated:

            break

    return first

def calculate\_follow(grammar, first):

    follow = {non\_terminal: set() for non\_terminal in grammar}

    start\_symbol = list(grammar.keys())[0]

    follow[start\_symbol] |= {'$'}

    while True:

        updated = False

        for non\_terminal, productions in grammar.items():

            for production in productions:

                for i, symbol in enumerate(production):

                    if symbol in grammar:

                        rest = production[i+1:]

                        first\_rest = {r[0] for r in rest if r[0] in grammar} | {'epsilon'}

                        for r in rest:

                            if r[0] in grammar and 'epsilon' in first[r[0]] and 'epsilon' in first\_rest:

                                first\_rest |= first[r[0]] - {'epsilon'}

                            else:

                                first\_rest -= {'epsilon'}

                                break

                        else:

                            first\_rest -= {'epsilon'}

                            first\_rest |= follow[non\_terminal]

                        if first\_rest - follow[symbol]:

                            follow[symbol] |= first\_rest

                            updated = True

        if not updated:

            break

    return follow

def create\_parse\_table(grammar, first, follow):

    parse\_table = {}

    for non\_terminal, productions in grammar.items():

        parse\_table[non\_terminal] = {}

        for terminal in grammar[non\_terminal]:

            if terminal != 'FOLLOW':

                parse\_table[non\_terminal][terminal] = []

        for production in productions:

            first\_set = []

            for symbol in production:

                if symbol in grammar:

                    first\_set += [x for x in first[symbol] if x != 'epsilon']

                    if 'epsilon' not in first[symbol]:

                        break

                else:

                    first\_set.append(symbol)

                    break

            else:

                first\_set += follow[non\_terminal]

            for terminal in first\_set:

                if terminal in parse\_table[non\_terminal]:

                    parse\_table[non\_terminal][terminal].append(production)

                else:

                    parse\_table[non\_terminal][terminal] = [production]

            if 'epsilon' in first\_set:

                for terminal in follow[non\_terminal]:

                    if terminal in parse\_table[non\_terminal]:

                        parse\_table[non\_terminal][terminal].append(production)

                    else:

                        parse\_table[non\_terminal][terminal] = [production]

    return parse\_table

grammar = {

    'S': ['A B', 'C'],

    'A': ['A a', 'b'],

    'B': ['b'],

    'C': ['A C', 'd']

}

first = calculate\_first(grammar)

follow = calculate\_follow(grammar, first)

parse\_table = create\_parse\_table(grammar, first, follow)

print('first set \n',first, '\n')

print('follow set \n',follow, '\n')

print('parse table \n',parse\_table, '\n')