*Lexer code :*

*# Define a function to tokenize the input string*

def tokenize(input\_string):

    tokens = []

    position = 0

*while* position < len(input\_string):

        char = input\_string[position]

*if* char.isdigit():

*# Parse integer*

            start = position

*# because if i have more than one intger like 12345 in the begin of the code*

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

                position += 1

            tokens.append(('INTEGER', input\_string[start:position]))

*elif* char == '+':

            tokens.append(('PLUS', '+'))

            position += 1

*elif* char == '-':

            tokens.append(('MINUS', '-'))

            position += 1

*elif* char == '\*':

            tokens.append(('MULTIPLY', '\*'))

            position += 1

*elif* char == '/':

            tokens.append(('DIVIDE', '/'))

            position += 1

*elif* char == '(':

            tokens.append(('LPAREN', '('))

            position += 1

*elif* char == ')':

            tokens.append(('RPAREN', ')'))

            position += 1

*elif* char.isspace():

*# Skip whitespace*

            position += 1

*else*:

*raise* ValueError(f"Invalid character: {char}")

*return* tokens

*# Test the tokenizer with an example input string*

input\_string = "3 + 4 \* 5 - 6 / 2"

tokens = tokenize(input\_string)

print(tokens)

*output :*

PS C:\Users\HELAL\Desktop\Compiler\prictical> & C:/Users/HELAL/AppData/Local/Programs/Python/Python310/python.exe c:/Users/HELAL/Desktop/Compiler/prictical/lexer.py

[('INTEGER', '3'), ('PLUS', '+'), ('INTEGER', '4'), ('MULTIPLY', '\*'), ('INTEGER', '5'), ('MINUS', '-'), ('INTEGER', '6'), ('DIVIDE', '/'), ('INTEGER', '2')]

PS C:\Users\HELAL\Desktop\Compiler\prictical>

*Porse tree :*

*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)

*output:*

PS C:\Users\HELAL\Desktop\Compiler\prictical> & C:/Users/HELAL/AppData/Local/Programs/Python/Python310/python.exe c:/Users/HELAL/Desktop/Compiler/prictical/parse\_tree.py

start

  expr

    expr

      atom

        3

    atom

      expr

        expr

          atom

            4

        atom

          5

PS C:\Users\HELAL\Desktop\Compiler\prictical>

*Symbol table:*

*# 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 row in table*

                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"]

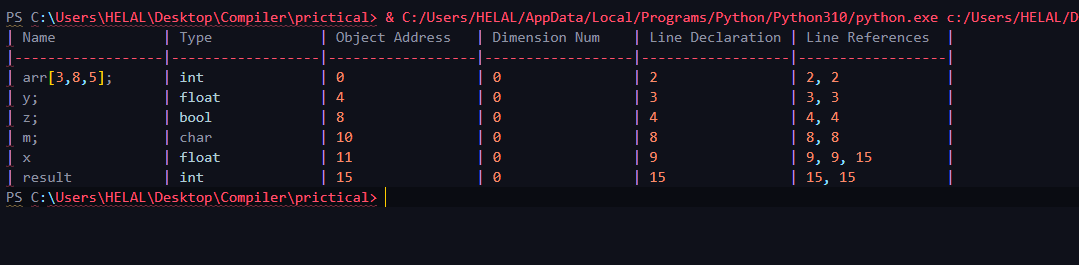
    line\_references = ", ".join(map(str, entry["Line References"]))

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

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

    )

*Output :*

**

*Parse tree:*

def calculate\_first(grammar):

    first = {}

*for* non\_terminal *in* grammar:

        first[non\_terminal] = set()

*while* True:

        updated = False

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

*for* production *in* productions:

*if* production[0] not in grammar:

*if* production[0] not in first[non\_terminal]:

                        first[non\_terminal].add(production[0])

                        updated = True

*else*:

*for* symbol *in* production:

*if* symbol not in first[non\_terminal]:

                            first[non\_terminal].update(first[symbol])

*if* 'epsilon' not in first[symbol]:

*break*

*if* symbol == production[-1]:

                                first[non\_terminal].add('epsilon')

                                updated = True

*if* not updated:

*break*

*return* first

def calculate\_follow(grammar, first):

    follow = {}

*for* non\_terminal *in* grammar:

        follow[non\_terminal] = set()

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

    follow[start\_symbol].add('$')

*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 = set()

*for* s *in* rest:

*if* s in grammar:

                                first\_s = first[s]

                                first\_rest |= first\_s - {'epsilon'}

*if* 'epsilon' not in first\_s:

*break*

*else*:

                                first\_rest.add(s)

*break*

*else*:

                            first\_rest |= follow[non\_terminal]

*if* not follow[symbol].issuperset(first\_rest):

                            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)

print('follow set \n',follow)

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

*output :*

PS C:\Users\HELAL\Desktop\Compiler\prictical> & C:/Users/HELAL/AppData/Local/Programs/Python/Python310/python.exe c:/Users/HELAL/Desktop/Compiler/prictical/frist.py

first set

 {'S': {'b', 'd'}, 'A': {'b'}, 'B': {'b'}, 'C': {'b', 'd'}}

follow set

 {'S': {'$'}, 'A': {' '}, 'B': {'$'}, 'C': {'$'}}

parse table

 {'S': {'A B': [], 'C': [], 'b': ['A B', 'C'], 'd': ['C']}, 'A': {'A a': [], 'b': ['A a', 'b']}, 'B': {'b': ['b']}, 'C': {'A C': [], 'd': ['d'], 'b': ['A C']}}

PS C:\Users\HELAL\Desktop\Compiler\prictical>