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_de
claration,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 :

```
PS C:\Users\HELAL\Desktop\Compiler\prictical> & C:/Users/HELAL/AppData/Local/Programs/Python/Python310/python.exe c:/Users/HELAL/D
| Name             | Type             | Object Address   | Dimension Num    | Line Declaration | Line References  |
|------------------|------------------|------------------|------------------|------------------|------------------|
| arr[3,8,5];      | int              | 0                | 0                | 2                | 2, 2             |
| y;               | float            | 4                | 0                | 3                | 3, 3             |
| z;               | bool             | 8                | 0                | 4                | 4, 4             |
| m;               | char             | 10               | 0                | 8                | 8, 8             |
| x                | float            | 11               | 0                | 9                | 9, 9, 15         |
| result           | int              | 15               | 0                | 15               | 15, 15           |
PS C:\Users\HELAL\Desktop\Compiler\prictical>
```

Parse table

```
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(productio
n)
                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(produ
ction)
                    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>
```