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

```python
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')
```