UNIVERSITY OF KARACHI
UBIT

COMPILER CONSTRUCTION
LAB DOCUMENTATION

GROUP 1
MEMBERS NAME

TAHRIM BILAL (B21110006153)

YUMNA MUBEEN (B21110006165)

SUBMITTED TO: MA'AM ATTIA AGHA

INPUT FILE:

```
$$ This is a
multi line comment $$

$ loop and conditionals(if-else)
void myFunction() {
    double i = 1;
    loop(i < 5) {
        if (i == 3) {
            br
        };
        else{
            i = i + 2
        };
        i = i + 1
        prnt(i);
    };
};

$ variable initialize
str name;
double age;
double score = 10.0;
str greeting = "Hello world!";

$ input
void getInput() {
    inp("Enter your name: ");
};
```

```
$ function initialization
double calculate(double a, double b) {
    double c = a + b;
    retn (c);
};

$ array
double[] scores = {1,2,3};
str[] names = {"Alice" , "Bob"};

$ inheritence
pub class Animal {
    void makeSound() {
        prnt("Animal sound");
    };
};

pub class Dog extends Animal {
    void bark() {
        prnt("Dog barks!");
    };
};


$ object
Dog dog = new Dog();
```

```
$ object calling
dog.bark();

$ function calling
getInput();
calculate(55, 45);
```

LEXICAL ANALYZER:

CODE:

```
import re

# Pattern for identifiers
identifier = r'^(A[a-zA-Z0-9^]*|[a-zA-Z][a-zA-Z0-9^]*)$'
# Pattern for double
double = r'\b[+-]?(\d*\.\d+|\d+\.?)([eE][+-]?\d+)?\b'
# Pattern for string
A = r"[\\'|\"|\\]"  # \",\',\\
B = r"[bntro]"    # with or without backslash
C = r"[@+:!.]"       # Not allowed with a backslash
D = r"[a-zA-Z\s+_=]" # Letters , space ,_,+,= allowed
char_const = rf"(\\{A}|\\{B}|{B}|{C}|{D})"
str_pattern = rf'^"({char_const})*"$'
# Dictionaries for keywords, operators, and punctuators
keywords = {
    'double': 'DT',    # DT = Data Type
    'str': 'String',
    'void': 'void',
    'loop': 'Loop',
    'br': 'Break',
    'if': 'if',
    'else': 'else',
    'prnt': 'Print',
    'inp': 'input',
    'retn': 'return',
    'class': 'class',
    'extends': 'extends',
    'supr': 'Super',
    'this': 'This',
    'pub': 'AM',        # Access Modifier
    'pri': 'AM',
    'final': 'Final',
    'new': 'new',
    'arr': 'Array',
    'and': 'And',
    'or': 'Or',
    'not': 'Not'
}
operators = {
    "+": "PM",       # PM = Plus Minus
    "-": "PM",
    "*": "MDM",      # MDM = Multiple Divide Modulo
    "/": "MDM",
    "%": "MDM",
    "<": "ROP",      # ROP = Relational Operator
    ">": "ROP",
    "<=": "ROP",
    ">=": "ROP",
    "!": "ROP",
    "!=": "ROP",
    "==": "ROP",
    "++": "INC_DEC",
    "--": "INC_DEC",
    "=": "=",
}
punctuators = {
    '(': '(',
    ')': ')',
    '{': '{',
    '}': '}',
    '[': '[',
    ']': ']',
    '.': '.',
    ';': ';',
    ',': ','
}
class Token:    # Token Class
    def __init__(self, value, token_type, line):
        self.value = value
        self.type = token_type
        self.line = line
    def __repr__(self):
        return f"Token Set (value='{self.value}', type='{self.type}', line={self.line})"
```

```
def read_file(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()  # Read all lines from the file
    return lines
# Token Function for Dot (.)
def handle_dot_token(token, line_number):
    token_classes = []
    segments = re.split(r'(\.)', token)
    i = 0
    while i < len(segments):
        if not segments[i].strip():
            i += 1
            continue

        if segments[i].isdigit() and i + 2 < len(segments) and segments[i + 1] == '.' and segments[i + 2].isdigit():
            # Handle cases like '123.456'
            token_classes.append(Token(f"{segments[i]}.{segments[i + 2]}", 'DOUBLE', line_number))
            i += 3
        elif segments[i] == '.':
            # Handle cases like '.123' or '.'
            if i + 1 < len(segments) and re.fullmatch(r'\d+([eE][+-]?\d+)?', segments[i + 1]):
                token_classes.append(Token(f".{segments[i + 1]}", 'DOUBLE', line_number))
                i += 2
            else:
                token_classes.append(Token(segments[i], 'DOT', line_number))
                i += 1
        else:
            # Handle tokens without dots
            if re.fullmatch(double, segments[i]):
                token_classes.append(Token(segments[i], 'DOUBLE', line_number))
            elif re.fullmatch(identifier, segments[i]):
                token_classes.append(Token(segments[i], 'IDENTIFIER', line_number))
            else:
                token_classes.append(Token(segments[i], 'INVALID LEXEME', line_number))
            i += 1
    return token_classes
# Token Function for operators or punctuators
def handle_mixed_token(token, line_number):
    token_classes = []
    if '.' in token:
        token_classes.extend(handle_dot_token(token, line_number))
    else:
        i = 0
        while i < len(token):
            matched = False
            for op in sorted(operators.keys(), key=len, reverse=True):
                if token[i:i + len(op)] == op:
                    token_classes.append(Token(op, operators[op], line_number))
                    i += len(op)
                    matched = True
                    break
            for p in sorted(punctuators.keys(), key=len, reverse=True):
                if token[i:i + len(p)] == p:
                    token_classes.append(Token(p, punctuators[p], line_number))
                    i += len(p)
                    matched = True
                    break
            if not matched:    # if not operators and punctuators
                start = i
                while i < len(token) and (token[i].isalnum() or token[i] == '^'):
                    i += 1
                potential_token = token[start:i]
                if re.fullmatch(double, potential_token):
                    token_classes.append(Token(potential_token, 'DOUBLE', line_number))
                elif re.fullmatch(identifier, potential_token):
                    token_classes.append(Token(potential_token, 'IDENTIFIER', line_number))
                else:
                    token_classes.append(Token(potential_token, 'INVALID LEXEME', line_number))
    return token_classes
def classify_tokens(lines):
    token_classes = []
    inside_multiline_comment = False
    inside_string = False
    string_literal = ''
    line_number_of_string_start = 0
    for line_number, line in enumerate(lines, start=1):
        if inside_multiline_comment:      # for Multi Line comment $$
            if '$$' in line:
                inside_multiline_comment = False
```

```
                line = line.split('$$', 1)[1]
            else:
                continue
        if '$$' in line and line.strip().startswith('$$'):
            inside_multiline_comment = True
            continue
        if '$' in line and not inside_multiline_comment:      # for Single Lin comment $
            line = line.split('$', 1)[0]
        # Breakwords
        tokens = re.findall(r'(?:[^\s\(\)\{\}\[\],;"]+|[;,(){}\[\]]|"(?:\\.|[^"\\])*"?)', line)
        i = 0
        while i < len(tokens):
            token = tokens[i]
            if inside_string:          # for String
                string_literal += " " + token
                if token.endswith('"') and not token.endswith(r'\"'):
                    inside_string = False
                    string_literal = string_literal.strip()
                    if string_literal.endswith('""'):
                        string_literal = string_literal[:-1].strip()
                    if re.fullmatch(str_pattern, string_literal):
                        token_classes.append(Token(string_literal, 'String', line_number_of_string_start))
                    else:
                        token_classes.append(Token(string_literal, 'INVALID STRING', line_number_of_string_start))
                    string_literal = ''
                i += 1
                continue
            if token.startswith('"'):
                inside_string = True
                line_number_of_string_start = line_number
                string_literal = token
                if token.endswith('"') and not token.endswith(r'\"'):
                    inside_string = False
                    string_literal = string_literal.strip()
                    if re.fullmatch(str_pattern, string_literal):
                        token_classes.append(Token(string_literal, 'String', line_number_of_string_start))
                    else:
                        token_classes.append(Token(string_literal, 'INVALID STRING', line_number_of_string_start))
                    string_literal = ''
                i += 1
                continue
            if any(op in token for op in operators) or any(p in token for p in punctuators):
                token_classes.extend(handle_mixed_token(token, line_number))
            elif token in keywords:
                token_classes.append(Token(token, keywords[token], line_number))
            elif re.fullmatch(double, token):
                token_classes.append(Token(token, 'DOUBLE', line_number))
            elif token in operators:
                token_classes.append(Token(token, operators[token], line_number))
            elif token in punctuators:
                token_classes.append(Token(token, punctuators[token], line_number))
            elif re.fullmatch(identifier, token):
                token_classes.append(Token(token, 'IDENTIFIER', line_number))
            else:
                token_classes.append(Token(token, 'INVALID LEXEME', line_number))
            i += 1
    if inside_string:
        token_classes.append(Token(string_literal.strip(), 'INVALID LEXEME', line_number_of_string_start))
    return token_classes
def output_results(token_classes):
    for token in token_classes:
        print(token)

file_path = "input.txt"
lines = read_file(file_path)
token_classes = classify_tokens(lines)
output_results(token_classes)
tokenize = []
tokenize = classify_tokens(lines)
```

SYNTAX ANALYZER:

CODE:

```
from Lexical_Analyzer import tokenize

class Token:
    def __init__(self, value_part, class_part, line_number):
        # Initialize a token with value, class type, and line number
        self.value_part = value_part
        self.class_part = class_part
        self.line_number = line_number
    def __repr__(self):
        # String representation for debugging
        return f"Token(value='{self.value_part}', type='{self.class_part}', line={self.line_number})"
class Parser:
    def __init__(self, tokens):
        # Initialize parser with list of tokens
        self.tokens = tokens
        self.current_index = 0  # Track current position in tokens
        self.current_token = tokens[0] if tokens else None  # Start with first token if available
    def eat(self, *token_types):
        #  token if it eates one of the expected types
        if self.current_token and self.current_token.class_part in token_types:
            consumed_token = self.current_token  # Store the token before advancing
            self.current_index += 1
            # Update current token or set to None if end of tokens
            self.current_token = self.tokens[self.current_index] if self.current_index < len(self.tokens) else None
            return consumed_token
        else:
            # Raise error if expected token type does not eat
            raise Exception(f"Syntax error at line {self.current_token.line_number}: expected one of {token_types}, got
{self.current_token.class_part if self.current_token else 'EOF'}")
    def parse_program(self):
        """Parse a program consisting of multiple statements."""
        statements = []  # List to store parsed statements
        while self.current_index < len(self.tokens):  # Process all tokens
            statements.append(self.parse_statement())  # Parse each statement
        return statements
    def parse_statement(self):
        """Parse a statement based on the defined grammar."""
        # Parse 'break' statement
        if self.current_token.value_part == "br":
            self.eat("Break")
            return {"type": "Break"}
        # Parse 'this' or 'supr' access statements
        if self.current_token.value_part in {"supr", "this"}:
            return self.parse_access()
        # Parse data type declarations or array declarations
        if self.current_token.class_part in {"DT", "String", "void"}:
            data_type = self.eat("DT", "String", "void").value_part
            # Check for array declaration
            if self.current_token.class_part == "[":
                self.eat("[")
                self.eat("]")
                array_name = self.eat("IDENTIFIER").value_part
                # Check for array initialization
                if self.current_token.class_part == "=":
                    return self.parse_array_initialization(data_type, array_name)
                else:
                    self.eat(";")
                    return {
                        'type': 'array_declaration',
                        'data_type': data_type,
                        'name': array_name
                    }
            # Parse variable or function declaration
            elif self.current_token.class_part == "IDENTIFIER":
                identifier_name = self.eat("IDENTIFIER").value_part
                # Check for function declaration
                if self.current_token.class_part == "(":
                    return self.parse_function(identifier_name, data_type)
                else:
                    # Parse variable initialization or declaration
                    if self.current_token.class_part == "=":
                        return self.parse_var_initialization(data_type, identifier_name)
                    else:
                        return self.parse_var_declaration(data_type, identifier_name)
```

```
        # Handle identifier for object or function calls
        elif self.current_token.class_part == "IDENTIFIER":
            next_token = self.tokens[self.current_index + 1]
            # Object declaration or calling
            if next_token.class_part == "IDENTIFIER":
                return self.parse_object()
            elif next_token.class_part == "DOT":
                return self.parse_object_calling()
            elif next_token.class_part == "(":
                return self.parse_function_calling()
        # Parse control structures: if, loop, print, input, return
        elif self.current_token.class_part == "if":
            return self.parse_conditional()
        elif self.current_token.class_part == "Loop":
            return self.parse_loop()
        elif self.current_token.class_part == "Print":
            return self.parse_print()
        elif self.current_token.class_part == "input":
            return self.parse_input()
        elif self.current_token.class_part == "AM":
            return self.parse_class()
        elif self.current_token.class_part == "void":
            return self.parse_function()
        elif self.current_token.class_part == "return":
            return self.parse_return()
        else:
            # Raise error if unexpected token found
            raise Exception(f"Unexpected token: {self.current_token.class_part}")
    def parse_function(self, identifier_name, data_type):
        """Parse a function definition with parameters and a body block."""
        self.eat("(")  # Match opening parenthesis
        parameters = []  # List to store function parameters
        while self.current_token.class_part != ")":
            # Parse parameter type and name
            if self.current_token.class_part in {"DT", "String"}:
                param_type = self.eat("DT", "String").class_part
                param_name = self.eat("IDENTIFIER").value_part
                parameters.append({"type": param_type, "name": param_name})
                if self.current_token.class_part == ",":
                    self.eat(",")  # Match comma separating parameters
            else:
                raise Exception(f"Expected parameter type, got: {self.current_token.class_part}")
        self.eat(")")  # Match closing parenthesis
        block = self.parse_block()  # Parse function body block
        return {
            'type': 'function_definition',
            'name': identifier_name,
            'data_type': data_type,
            'parameters': parameters,
            'body': block
        }
    def parse_var_declaration(self, data_type, variable_name):
        """Parse variable declaration: <data_type> <variable_name>;"""
        self.eat(";")  # Match semicolon ending declaration
        return {'type': 'var_declaration', 'data_type': data_type, 'name': variable_name}

    def parse_var_initialization(self, data_type, variable_name):
        """Parse variable initialization, e.g., 'double c = a + b;' or 'double c = 3.14;'."""
        self.eat("=")  # Match assignment operator

        # Try parsing based on the expected data type
        if data_type == "double":
            # Parse either a complex expression or a simple double literal
            if self.current_token.class_part in ["IDENTIFIER", "NUMBER", "(", "+", "-", "*", "/"]:
                expression = self.parse_expression()
            else:
                expression = self.parse_Double()  # Directly parse as a double if it's a literal
        elif data_type == "str":
            expression = self.parse_String()  # Parse as a string literal for `str` type
        else:
            raise Exception(f"Unsupported data type: {data_type}")

        self.eat(";")  # Match semicolon to end initialization statement

        return {
            'type': 'variable_initialization',
            'data_type': data_type,
            'identifier': variable_name,
            'value': expression
```

```python
        }
    def parse_Double(self):
        """Parse a double/number."""
        # Check if current token is a number and return its value
        if self.current_token.class_part == "DOUBLE":
            return {'type': 'number', 'value': self.eat("DOUBLE").value_part}
        else:
            raise Exception(f"Expected a NUMBER, but got {self.current_token.class_part}")
    def parse_String(self):
        """Parse a string."""
        # Check if current token is a string and return its value
        if self.current_token.class_part == "String":
            return {'type': 'string', 'value': self.eat("String").value_part}
        else:
            raise Exception(f"Expected a STRING, but got {self.current_token.class_part}")
    def parse_expression(self):
        """Parse an expression that can involve identifiers, numbers, and specified operators."""
        left = self.parse_term()  # Parse the left part of the expression
        # Continue parsing if operator is present
        while self.current_token and self.current_token.class_part in ["ROP", "PM", "MDM", "INC_DEC", "="]:
            operator_type = self.current_token.class_part
            operator_value = self.current_token.value_part
            self.eat(self.current_token.class_part)  # Match operator
            right = self.parse_term()  # Parse the right part of the expression
            left = {"type": operator_type, "left": left, "operator": operator_value, "right": right}

        return left
    def parse_term(self):
        """Parse a term which can be an identifier or a number."""
        # Check for identifier or number and return appropriate type
        if self.current_token.class_part == "IDENTIFIER":
            identifier = self.current_token.value_part
            self.eat("IDENTIFIER")
            return {"type": "identifier", "name": identifier}
        elif self.current_token.class_part == "DOUBLE":
            value = self.current_token.value_part
            self.eat("DOUBLE")
            return {"type": "number", "value": value}
        else:
            raise Exception(f"Unexpected token in term: {self.current_token}")
    def parse_loop(self):
        """Parse loop structure: loop (...) { statements }."""
        self.eat("Loop")  # Match 'Loop' keyword
        self.eat("(")  # Match opening parenthesis
        condition = self.parse_expression()  # Parse loop condition
        self.eat(")")  # Match closing parenthesis
        block = self.parse_block()  # Parse loop body block
        return {"type": "loop", "condition": condition, "block": block}
    def parse_conditional(self):
        """Parse conditional: if (...) { statements }."""
        self.eat("if")  # Match 'if' keyword
        self.eat("(")  # Match '(' symbol for condition start
        condition = self.parse_expression()  # Parse the condition expression inside 'if'
        self.eat(")")  # Match ')' symbol for condition end
        true_block = self.parse_block()  # Parse the 'if' block when condition is true
        false_block = None
        if self.current_token and self.current_token.value_part == "else":
            self.eat("else")  # Match 'else' keyword if it exists
            false_block = self.parse_block()  # Parse the 'else' block
        return {'type': 'conditional', 'condition': condition, 'true_block': true_block, 'false_block': false_block}
    def parse_block(self):
        """Parse block: { statements }."""
        self.eat("{")  # Match '{' symbol for block start
        statements = []
        while self.current_token and self.current_token.value_part != "}":
            if self.current_token.class_part == "IDENTIFIER":
                statements.append(self.parse_expression())  # Parse expression for identifiers
            else:
                statements.append(self.parse_statement())  # Parse other types of statements
        self.eat("}")  # Match '}' symbol for block end
        self.eat(";")  # Match ';' symbol to end the block
        return {'type': 'block', 'statements': statements}
    def parse_print(self):
        """Parse print: prnt (...);"""
        self.eat("Print")  # Match 'Print' keyword
        self.eat("(")  # Match '(' for start of print statement
        # variable = self.parse_Identifier()  # Parse identifier to print
        if self.current_token.class_part == "String":
            variable = self.parse_String()  # Parse string input
        elif self.current_token.class_part == "IDENTIFIER":
            variable = self.parse_Identifier()  # Parse identifier input
```

```
        else:
            raise Exception(f"Unexpected token: {self.current_token.class_part}")  # Raise exception if unexpected token found

        self.eat(")")  # Match ')' to end print statement
        self.eat(";")  # Match ';' to end print statement
        return {'type': 'print', 'variable': variable}
    def parse_input(self):
        """Parse imput: inp(...);"""
        self.eat("input")  # Match 'input' keyword
        self.eat("(")  # Match '(' symbol for input start

        if self.current_token.class_part == "String":
            variable = self.parse_String()  # Parse string input
        elif self.current_token.class_part == "IDENTIFIER":
            variable = self.parse_Identifier()  # Parse identifier input
        else:
            raise Exception(f"Unexpected token: {self.current_token.class_part}")  # Raise exception if unexpected token found

        self.eat(")")  # Match ')' to end input statement
        self.eat(";")  # Match ';' to end input statement
        return {'type': 'input', 'variable': variable}
    def parse_Identifier(self):
        if self.current_token.class_part == "IDENTIFIER":
            return {'type': 'variable', 'name': self.eat("IDENTIFIER").value_part}  # Return identifier if found
        else:
            raise Exception(f"Unexpected token: {self.current_token.class_part}")  # Raise exception if identifier not found
    def parse_return(self):
        """Parse return: retn(...);"""
        self.eat("return")  # Match 'return' keyword
        self.eat("(")  # Match '(' symbol for return start
        variable = self.parse_Identifier()  # Parse identifier to return
        self.eat(")")  # Match ')' to end return statement
        self.eat(";")  # Match ';' to end return statement
        return {'type': 'retrun', 'variable': variable}
    def parse_array_declaration(self, data_type):
        """Parse array declaration: <DT>[] <id>;"""
        self.eat("[")  # Match '[' for array declaration
        self.eat("]")  # Match ']' to complete array syntax
        variable_name = self.parse_Identifier()  # Parse array identifier
        self.eat(";")  # Match ';' to end array declaration
        return {'type': 'array_declaration', 'data_type': data_type, 'name': variable_name}
    def parse_array_initialization(self, data_type, array_name):
        """Parse array initialization: <identifier> = { <elements> };"""
        self.eat("=")  # Match '=' for initialization
        self.eat("{")  # Match '{' to start elements initialization
        elements = []
        while self.current_token.class_part != "}":
            if data_type == "double" and self.current_token.class_part == "DOUBLE":
                elements.append(float(self.eat("DOUBLE").value_part))  # Add double elements
            elif data_type == "str" and self.current_token.class_part == "String":
                elements.append(self.eat("String").value_part)  # Add string elements
            else:
                raise SyntaxError(f"Unexpected type in array initialization at line {self.current_token.line_number}")  # Handle
unexpected types

            if self.current_token.class_part == ",":
                self.eat(",")  # Match ',' to continue with more elements

        self.eat("}")  # Match '}' to end elements
        self.eat(";")  # Match ';' to end array initialization

        return {
            'type': 'array_initialization',
            'data_type': data_type,
            'name': array_name,
            'elements': elements
        }
    def parse_class(self):
        self.eat("AM")  # Match access modifier (AM)
        self.eat("class")  # Match 'class' keyword
        if self.current_index < len(self.tokens) and self.tokens[self.current_index].class_part == "IDENTIFIER":
            class_name = self.eat("IDENTIFIER").value_part  # Get class name
            if (self.current_index < len(self.tokens) and
                self.tokens[self.current_index].value_part == "extends"):
                self.eat("extends")  # Match 'extends' keyword for inheritance
                parent_class_name = self.eat("IDENTIFIER").value_part  # Get parent class name
                block = self.parse_block()  # Parse class block
                return {
                    'pub/pri': 'Access Modifier',
                    'type': 'inheritance',
                    'name': class_name,
```

```
                'parent': parent_class_name,
                'block': block
            }
        else:
            block = self.parse_block()  # Parse block without inheritance
            return {
                'pub/pri': 'Access Modifier',
                'type': 'class',
                'name': class_name,
                'block': block,
            }
def parse_access(self):
    """Parse `super.identifier` or `this.identifier` access syntax."""
    if self.current_token.value_part in {"supr", "this"}:  # Check for 'super' or 'this' keyword
        access_type = self.current_token.value_part  # Store the access type ('super' or 'this')
        self.eat("IDENTIFIER")  # Move past 'super' or 'this'
        self.eat("DOT")  # Move past the dot ('.') symbol
        identifier = self.eat("IDENTIFIER").value_part  # Store the accessed identifier name
        return {
            'type': f'{access_type}_access',  # Specify access type
            'identifier': identifier  # Store identifier name
        }
    else:
        raise SyntaxError(f"Expected 'super' or 'this' access, got '{self.current_token.value_part}'")
def parse_arguments(self):
    """Parse arguments for constructors or method calls."""
    arguments = []
    while True:
        if self.current_token.class_part == "IDENTIFIER":  # Check if argument is an identifier
            arguments.append(self.eat("IDENTIFIER").value_part)
        elif self.current_token.class_part in {"NUMBER", "STRING_LITERAL"}:  # Check if argument is number or string
            arguments.append(self.eat("NUMBER", "STRING_LITERAL").value_part)
        else:
            raise Exception(f"Unexpected argument type: {self.current_token.class_part}")
        if self.current_token.class_part == ",":  # Move past commas in argument list
            self.eat(",")
        elif self.current_token.class_part == ")":  # End of argument list
            break
        else:
            raise Exception(f"Expected ',' or ')', got '{self.current_token.class_part}'")
    return arguments
def parse_object(self):
    """Parse an object creation statement."""
    class_type = self.eat("IDENTIFIER").value_part  # Store class type for object creation
    object_name = self.eat("IDENTIFIER").value_part  # Store object name
    self.eat("=")  # Move past the assignment operator
    if self.current_token.value_part != "new":  # Expect the keyword 'new' for object instantiation
        raise Exception(f"Expected 'new', got {self.current_token.value_part}")
    self.eat("new")
    new_class_type = self.eat("IDENTIFIER").value_part  # Store the class type after 'new'
    if new_class_type != class_type:  # Check if class types eat
        raise Exception(f"Class name miseat: expected '{class_type}', got '{new_class_type}'")
    self.eat("(")
    parameters = []
    if self.current_token.class_part != ")":  # Parse parameters if present
        parameters = self.parse_arguments()
    self.eat(")")
    self.eat(";")  # End of statement
    return {
        "type": "object_creation",
        "class_type": class_type,
        "object_name": object_name,
        "parameters": parameters
    }
def parse_object_calling(self):
    """Parse an object method call."""
    object_name = self.eat("IDENTIFIER").value_part  # Get the object name
    self.eat("DOT")  # Move past the dot ('.') symbol
    object_name = self.eat("IDENTIFIER").value_part  # Get the method name
    self.eat("(")
    parameters = []
    if self.current_token.class_part != ")":  # Parse parameters if present
        parameters = self.parse_arguments()

    self.eat(")")
    self.eat(";")  # End of statement
    return {
        "type": "method_call",
        "object_name": object_name,
        "object_name": object_name,
        "parameters": parameters
```

```
        }
    def parse_function_calling(self):
        """Parse a function call based on the given tokens."""
        function_name = self.current_token.value_part  # Store function name
        self.eat("IDENTIFIER")
        self.eat("(")
        arguments = []
        if self.current_token.class_part != ")":  # Parse arguments if present
            while True:
                if self.current_token.class_part == "IDENTIFIER":  # Check if argument is identifier
                    arguments.append(self.is_identifier())
                elif self.current_token.class_part == "STRING":  # Check if argument is string
                    arguments.append(self.eat("STRING").value_part)
                elif self.current_token.class_part == "DOUBLE":  # Check if argument is double
                    arguments.append(self.eat("DOUBLE").value_part)
                else:
                    raise Exception(f"Unexpected argument type: {self.current_token.class_part}")

                if self.current_token.class_part == ",":  # Move past commas
                    self.eat(",")
                elif self.current_token.class_part == ")":  # End of argument list
                    break

        self.eat(")")
        self.eat(";")  # End of statement
        return {
            "type": "function_calling",
            "name": function_name,
            "arguments": arguments
        }

parser = Parser(tokenize)
ast = parser.parse_program()
# print(ast)  # Output the abstract syntax tree (AST)
```

OUTPUT:


```
[
    {
        "type": "function_definition",
        "name": "myFunction",
        "data_type": "void",
        "parameters": [],
        "body": {
            "type": "block",
            "statements": [
                {
                    "type": "variable_initialization",
                    "data_type": "double",
                    "identifier": "i",
                    "value": {
                        "type": "number",
                        "value": "1"
                    }
                },
                {
                    "type": "loop",
                    "condition": {
                        "type": "ROP",
                        "left": {
                            "type": "identifier",
                            "name": "i"
                        },
                        "operator": "<",
                        "right": {
                            "type": "number",
                            "value": "5"
                        }
                    },
                    "block": {
                        "type": "block",
                        "statements": [
                            {
                                "type": "conditional",
                                "condition": {
                                    "type": "ROP",
                                    "left": {
                                        "type": "identifier",
```

```
                        "name": "i"
                    },
                    "operator": "==",
                    "right": {
                        "type": "number",
                        "value": "3"
                    }
                },
                "true_block": {
                    "type": "block",
                    "statements": [
                        {
                            "type": "Break"
                        }
                    ]
                },
                "false_block": {
                    "type": "block",
                    "statements": [
                        {
                            "type": "PM",
                            "left": {
                                "type": "=",
                                "left": {
                                    "type": "identifier",
                                    "name": "i"
                                },
                                "operator": "=",
                                "right": {
                                    "type": "identifier",
                                    "name": "i"
                                }
                            },
                            "operator": "+",
                            "right": {
                                "type": "number",
                                "value": "2"
                            }
                        }
                    ]
                }
            },
            {
                "type": "PM",
                "left": {
                    "type": "=",
                    "left": {
                        "type": "identifier",
                        "name": "i"
                    },
                    "operator": "=",
                    "right": {
                        "type": "identifier",
                        "name": "i"
                    }
                },
                "operator": "+",
                "right": {
                    "type": "number",
                    "value": "1"
                }
            },
            {
                "type": "print",
                "variable": {
                    "type": "variable",
                    "name": "i"
                }
            }
        ]
    }
  }
 ]
}
},
{
    "type": "var_declaration",
    "data_type": "str",
    "name": "name"
},
{
```

```
        "type": "var_declaration",
        "data_type": "double",
        "name": "age"
    },
    {
        "type": "variable_initialization",
        "data_type": "double",
        "identifier": "score",
        "value": {
            "type": "number",
            "value": "10.0"
        }
    },
    {
        "type": "variable_initialization",
        "data_type": "str",
        "identifier": "greeting",
        "value": {
            "type": "string",
            "value": "\"Hello world!\""
        }
    },
    {
        "type": "function_definition",
        "name": "getInput",
        "data_type": "void",
        "parameters": [],
        "body": {
            "type": "block",
            "statements": [
                {
                    "type": "input",
                    "variable": {
                        "type": "string",
                        "value": "\"Enter your name: \""
                    }
                }
            ]
        }
    },
    {
        "type": "function_definition",
        "name": "calculate",
        "data_type": "double",
        "parameters": [
            {
                "type": "DT",
                "name": "a"
            },
            {
                "type": "DT",
                "name": "b"
            }
        ],
        "body": {
            "type": "block",
            "statements": [
                {
                    "type": "variable_initialization",
                    "data_type": "double",
                    "identifier": "c",
                    "value": {
                        "type": "PM",
                        "left": {
                            "type": "identifier",
                            "name": "a"
                        },
                        "operator": "+",
                        "right": {
                            "type": "identifier",
                            "name": "b"
                        }
                    }
                },
                {
                    "type": "retrun",
                    "variable": {
                        "type": "variable",
                        "name": "c"
                    }
                }
```

```
            ]
        }
    },
    {
        "type": "array_initialization",
        "data_type": "double",
        "name": "scores",
        "elements": [
            1.0,
            2.0,
            3.0
        ]
    },
    {
        "type": "array_initialization",
        "data_type": "str",
        "name": "names",
        "elements": [
            "\"Alice\"",
            "\"Bob\""
        ]
    },
    {
        "pub/pri": "Access Modifier",
        "type": "class",
        "name": "Animal",
        "block": {
            "type": "block",
            "statements": [
                {
                    "type": "function_definition",
                    "name": "makeSound",
                    "data_type": "void",
                    "parameters": [],
                    "body": {
                        "type": "block",
                        "statements": [
                            {
                                "type": "print",
                                "variable": {
                                    "type": "string",
                                    "value": "\"Animal sound\""
                                }
                            }
                        ]
                    }
                }
            ]
        }
    },
    {
        "pub/pri": "Access Modifier",
        "type": "inheritance",
        "name": "Dog",
        "parent": "Animal",
        "block": {
            "type": "block",
            "statements": [
                {
                    "type": "function_definition",
                    "name": "bark",
                    "data_type": "void",
                    "parameters": [],
                    "body": {
                        "type": "block",
                        "statements": [
                            {
                                "type": "print",
                                "variable": {
                                    "type": "string",
                                    "value": "\"Dog barks!\""
                                }
                            }
                        ]
                    }
                }
            ]
        }
    },
    {
        "type": "object_creation",
```

```
        "class_type": "Dog",
        "object_name": "dog",
        "parameters": []
    },
    {
        "type": "method_call",
        "object_name": "bark",
        "parameters": []
    },
    {
        "type": "function_calling",
        "name": "getInput",
        "arguments": []
    },
    {
        "type": "function_calling",
        "name": "calculate",
        "arguments": [
            "55",
            "45"
        ]
    }
]
```

SEMANTIC ANALYZER:

CODE:

```
class SemanticAnalyzer:
    def __init__(self, ast):
        self.ast = ast
        self.symbol_table = {}
        self.functions = {}
        self.classes = {}
        self.current_scope = None

    def analyze(self):
        for statement in self.ast:
            self.analyze_statement(statement, inherited=None)
    def analyze_statement(self, statement, inherited):
        if "type" not in statement:
            raise Exception("Unknown statement type")
        if statement["type"] == "var_declaration":
            return self.analyze_var_declaration(statement, inherited)
        elif statement["type"] == "var_initialization":
            return self.analyze_var_initialization(statement, inherited)
        elif statement["type"] == "class":
            return self.analyze_class(statement, inherited)
        elif statement["type"] == "if":
            return self.analyze_conditional(statement, inherited)
        elif statement["type"] == "function_calling":
            return self.analyze_function_calling(statement, inherited)
        elif statement["type"] == "function":
            return self.analyze_function(statement, inherited)
        elif statement["type"] == "loop":
            return self.analyze_loop(statement, inherited)
        elif statement["type"] == "expression":
            return self.analyze_expression(statement, inherited)
        elif statement["type"] == "print":
            return self.analyze_print(statement, inherited)
        elif statement["type"] == "input":
            return self.analyze_input(statement, inherited)
        elif statement["type"] == "object":
            return self.analyze_object(statement, inherited)
        elif statement["type"] == "object_calling":
            return self.analyze_object_calling(statement, inherited)
        elif statement["type"] == "array_declaration":
            return self.analyze_array_declaration(statement, inherited)
        elif statement["type"] == "array_initialization":
            return self.analyze_array_initialization(statement, inherited)
    def analyze_var_declaration(self, statement, inherited):
        var_name = statement['name']
        if var_name in self.symbol_table:
            raise Exception(f"Variable '{var_name}' is already declared.")
```

```
        var_type = inherited if inherited else statement.get('data_type', 'double')
        if var_type not in ['double', 'string']:
            raise Exception(f"Unsupported type '{var_type}' for variable '{var_name}'.")
        self.symbol_table[var_name] = {'type': var_type}
        statement['attributes'] = {'type': var_type}
        return var_type
    def analyze_var_initialization(self, statement, inherited):
        var_name = statement['name']
        if var_name not in self.symbol_table:
            raise Exception(f"Variable '{var_name}' used without declaration.")
        value_type = self.analyze_expression(statement['value'], inherited)
        declared_type = self.symbol_table[var_name]['type']
        if value_type != declared_type:
            raise Exception(f"Type mismatch: '{var_name}' declared as '{declared_type}', initialized with '{value_type}'.")
        return value_type
    def analyze_function(self, statement, inherited):
        func_name = statement["name"]
        if func_name in self.functions:
            raise Exception(f"Function '{func_name}' is already declared.")
        params = statement.get("parameters", [])
        return_type = statement.get("data_type", "void")
        self.functions[func_name] = {
            "parameters": params,
            "return_type": return_type
        }
        self.analyze_block(statement['body'], inherited=return_type)
        return return_type
    def analyze_expression(self, expression, inherited):
        if isinstance(expression, dict):
            if "type" in expression:
                if expression["type"] == "number":
                    return 'double'
                elif expression["type"] == "string":
                    return 'string'
                elif expression["type"] == "value":
                    return 'double'
                elif expression["type"] == "double":
                    return 'double'
                else:
                    raise Exception(f"Unknown expression type: {expression['type']}")
            else:
                raise Exception("Invalid expression format")
        else:
            raise Exception("Invalid expression type")
    def analyze_conditional(self, statement, inherited):
        condition_type = self.analyze_expression(statement['condition'], inherited)
        self.analyze_block(statement['true_block'], inherited)
        if 'false_block' in statement:
            self.analyze_block(statement['false_block'], inherited)
        return condition_type
    def analyze_loop(self, statement, inherited):
        condition_type = self.analyze_expression(statement['condition'], inherited)
        self.analyze_block(statement['block'], inherited)
        return condition_type
    def analyze_function_calling(self, statement, inherited):
        func_name = statement["name"]
        if func_name not in self.functions:
            raise Exception(f"Function '{func_name}' is called but not defined.")
        expected_params = self.functions[func_name]["parameters"]
        actual_args = statement.get("arguments", [])
        if len(expected_params) != len(actual_args):
            raise Exception(f"Function '{func_name}' expects {len(expected_params)} arguments, but got {len(actual_args)}.")
        for arg in actual_args:
            self.analyze_expression(arg, inherited)
        return self.functions[func_name]['return_type']
    def analyze_object_calling(self, statement, inherited):
        object_name = statement["object_name"]
        method_name = statement["method_name"]
        if object_name not in self.symbol_table:
            raise Exception(f"Object '{object_name}' used without declaration.")
        if method_name not in self.symbol_table[object_name]["methods"]:
            raise Exception(f"Method '{method_name}' does not exist in object '{object_name}'.")
        return self.symbol_table[object_name]["methods"][method_name]["return_type"]
    def analyze_object(self, statement, inherited):
        class_name = statement["class_type"]
        if class_name not in self.classes:
            raise Exception(f"Class '{class_name}' is not declared.")
        return "object"
    def analyze_array_declaration(self, statement, inherited):
        array_name = statement['name']
        array_type = statement['data_type']
```

```
        self.symbol_table[array_name] = {'type': array_type, 'is_array': True}
        return array_type
    def analyze_array_initialization(self, statement, inherited):
        array_name = statement['name']
        if array_name not in self.symbol_table:
            raise Exception(f"Array '{array_name}' used without declaration.")

        declared_type = self.symbol_table[array_name]['type']
        elements = statement['value']['elements']
        for element in elements:
            element_type = self.analyze_expression(element, inherited)
            if element_type != declared_type:
                raise Exception(f"Type mismatch: Array '{array_name}' expects elements of type '{declared_type}', "
                                f"but got '{element_type}'.")
        return declared_type
    def analyze_class(self, statement, inherited):
        class_name = statement['class_name']
        if class_name in self.classes:
            raise Exception(f"Class '{class_name}' is already declared.")
        self.classes[class_name] = statement
        self.analyze_block(statement['block'], inherited)
        return 'class'
    def analyze_print(self, statement, inherited):
        var_name = statement["variable"]["name"]
        if var_name not in self.symbol_table:
            raise Exception(f"Variable '{var_name}' used without declaration.")
        return self.symbol_table[var_name]['type']
    def analyze_input(self, statement, inherited):
        var_name = statement["variable"]["name"]
        if var_name not in self.symbol_table:
            raise Exception(f"Variable '{var_name}' used without declaration.")
        return self.symbol_table[var_name]['type']
    def analyze_block(self, block, inherited):
        for stmt in block['statements']:
            self.analyze_statement(stmt, inherited)
    def print_attributed_ast(self):
        import json
        print(json.dumps(self.ast, indent=2))
ast = [
# variable_declaration
    {
        "type": "variable_declaration",
        "data_type": "string",
        "name": "hello",
    },
# variable_initialization
    {
        "type": "variable_initialization",
        "data_type": "double",
        "name": "x",
        "value": {"type": "number", "value": 0}
    },
# function_declaration
    {
        "type": "function_declaration",
        "name": "myFunction",
        "arguments": [
            {"type": "double", "name": "number"}
        ],
        "body": {
            "statements": []
        }
    },
# function call
    {
        "type": "function_call",
        "name": "myFunction",
        "arguments": [
            {"type": "value", "value": 5},
        ]
    },
# loop
    {
        "type": "loop",
        "condition": {"type": "value", "value": 1},
        "block": {
            "statements": []
        }
    },
# conditions if-else
    {
```

```
        "type": "if",
        "condition": {"type": "double", "name": "x"},
        "true_block": {
        "statements": []
        },
        "false_block": {
            "statements": []
        }
    },
# array_declaration
    {
        "type": "array_declaration",
        "data_type": "double",
        "name": "arr1",
    },
# array_initialization
    {
        "type": "array_initialization",
        "name": "arr1",
        "value": {
            "type": "array_initialization",
            "elements": [
                {"type": "number", "value": 1.0},
                {"type": "number", "value": 2.0},
                {"type": "number", "value": 3.0},
            ]
        }
    },
# class
    {
        "type": "class",
        "class_name": "myClass",
        "block": {
            "statements": []
        }
    },
# object
    {
        "type": "object",
        "class_type": "myClass",
        "name": "myObject"
    },
# object call
    {
        "type": "object_call",
        "object_name": "myObject",
        "method_name": "myFunction",
        "arguments": [
            {"type": "value", "value": 22},
            {"type": "value", "value": "code"}
        ]
    }
]
analyzer = SemanticAnalyzer(ast)
try:
    analyzer.analyze()
    print("Semantic analysis passed!")
    analyzer.print_attributed_ast()  # Print the attributed AST
except Exception as e:
    print(f"Semantic analysis error: {e}")
```

OUTPUT:

Semantic analysis passed!

[
  {
    "type": "variable_declaration",
    "data_type": "string",
    "name": "hello"
  },
  {
    "type": "variable_initialization",
    "data_type": "double",
    "name": "x",
    "value": {
      "type": "number",
      "value": 0
    }
  },
  {
    "type": "function_declaration",
    "name": "myFunction",
    "arguments": [
      {
        "type": "double",
        "name": "number"
      }
    ],
    "body": {
      "statements": []
    }
  },
  {
    "type": "function_call",
    "name": "myFunction",
    "arguments": [
      {
        "type": "value",

```
      "value": 5
    }
  ]
},
{
  "type": "loop",
  "condition": {
    "type": "value",
    "value": 1
  },
  "block": {
    "statements": []
  }
},
{
  "type": "if",
  "condition": {
    "type": "double",
    "name": "x"
  },
  "true_block": {
    "statements": []
  },
  "false_block": {
    "statements": []
  }
},
{
  "type": "array_declaration",
  "data_type": "double",
  "name": "arr1"
},
{
  "type": "array_initialization",
  "name": "arr1",
```

```
  "value": {
    "type": "array_initialization",
    "elements": [
      {
        "type": "number",
        "value": 1.0
      },
      {
        "type": "number",
        "value": 2.0
      },
      {
        "type": "number",
        "value": 3.0
      }
    ]
  }
},
{
  "type": "class",
  "class_name": "myClass",
  "block": {
    "statements": []
  }
},
{
  "type": "object",
  "class_type": "myClass",
  "name": "myObject"
},
{
  "type": "object_call",
  "object_name": "myObject",
  "method_name": "myFunction",
  "arguments": [
```

```
      {
        "type": "value",
        "value": 22
      },
      {
        "type": "value",
        "value": "code"
      }
    ]
  }
]
```

SUMMARY:

LEXICAL:

Lexical analyzer defines regular expressions to identify tokens such as identifiers, doubles, and string patterns. It sets up patterns for different components including dictionaries to store keywords , operators and punctuators.

SYNTAX:

Syntax analyzer defines a Token class for token details like value, type, and line number. A Parser class is initialized with a list of tokens and likely processes these tokens according to predefined grammar rules. The parser uses methods to check syntax structures and produce a syntax tree

SEMANTIC:

Semantic analyzer takes an abstract syntax tree (AST) as input. It initializes tables to store symbols, functions, and classes and uses a method (analyze) to process each statement in the AST. Methods within this class handle variable declaration, type checking, and scope management, ensuring that the code follows semantic rules.

.