PROJECT REPORT

Compiler Construction Lab (CSL-323)

PROJECT TITLE:

BS(CS) – 5B

Group Members

Name	Enrollment
1. Rooma Siddiqui	02-134222-023
2. Danish Siddiqui	02-134222-018
3. Abiha Jawaid	02-134222-077

Submitted to:

Ma’am Mehwish Saleem

BAHRIA UNIVERSITY KARACHI CAMPUS

Department of Computer Science

INTRODUCTION

The Scribble programming language has been conceptualized as a minimalistic and beginner-friendly programming language. With a focus on simplicity and clarity, it introduces basic programming concepts in a structured yet accessible manner. The primary aim of the Scribble project is to provide an interpreter capable of understanding and executing code written in this language, thereby empowering novice programmers to explore the fundamentals of programming.

PROBLEM STATEMENT

Programming languages often present a steep learning curve for beginners due to their complexity and vast range of features. This project addresses the gap by offering Scribble, a language designed with simplicity at its core. The aim is to create a language with an intuitive syntax and minimal constructs to aid in the learning process. An interpreter for Scribble ensures the language is not just theoretical but also executable, further enhancing the learning experience.

METHODOLOGY

The project is divided into three main components: **Lexer**, **Parser**, and **Interpreter**. These components work together to tokenize, parse, and execute Scribble code.

1. Lexer

The lexer (lexical analyzer) converts the raw text of the Scribble code into tokens. Tokens are fundamental units, such as:

- **Keywords**: `make`, `if`, `while`
- **Operators**: `+`, `-`, `*`, `/`, `<`, `>`, `and`, `or`, `not`
- **Identifiers**: Variable names like `a`, `b`, `x`
- **Literals**: Numeric constants like `5`, `10`

Key Functions:

- **Input Reading**: Processes the input script character by character.
- **Token Identification**: Matches text with predefined patterns (e.g., regular expressions) to classify tokens.
- **Output**: A sequential list of tokens for the parser.

2. Parser

The parser builds a **syntax tree** from the token stream provided by the lexer. This tree represents the grammatical structure of the Scribble program and organizes the code hierarchically based on its syntax.

Key Functions:

- **Syntax Validation**: Ensures the code follows the rules of the Scribble language.
- **Tree Construction**: Represents programming constructs like assignments, expressions, and control structures in a tree format.
- **Error Handling**: Detects and reports invalid syntax.

3. Interpreter

The interpreter traverses the syntax tree to execute the Scribble code. It evaluates expressions, manages variable assignments, and handles logical control flows such as `if` statements and `while` loops.

Features Supported:

1. **Variable Assignments**: Example: `make a = 5` assigns 5 to variable `a`.
2. **Arithmetic Operations**: Supports `+`, `-`, `*`, `/`, with proper order of precedence.
3. **Boolean Operations**: Includes `and`, `or`, `not`, as well as comparisons (`<`, `>`, `equal to`, `not equal to`).
4. **Control Statements**:

PROJECT SCOPE

Current Features:

1. Tokenization of basic Scribble syntax.
2. Parsing and syntax validation for:
    - Arithmetic expressions
    - Boolean expressions
    - Conditional statements
    - Loop structures
3. Execution of code with variable handling and dynamic computation.
4. Comprehensive testing with over 20 test cases covering variable assignments, arithmetic, logical operators, precedence rules, and complex expressions.

Limitations:

- No support for user-defined functions or complex data structures.
- Execution is sequential without support for multithreading or parallelism.

CODE

#token.py

```
class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __repr__(self):
        return f"{self.type}({self.value})"


class Integer(Token):
    def __init__(self, value):
        super().__init__("INT", value)


class Float(Token):
    def __init__(self, value):
        super().__init__("FLT", value)


class Operation(Token):
    def __init__(self, value):
        super().__init__("OP", value)


class Declaration(Token):
    def __init__(self, value):
        super().__init__("DECL", value)


class Variable(Token):
    def __init__(self, value):
        super().__init__("VAR", value)


class Boolean(Token):
    def __init__(self, value):
        super().__init__("BOOL", value)


class Comparison(Token):
    def __init__(self, value):
        super().__init__("COMP", value)


class Reserved(Token):
    def __init__(self, value):
        super().  init  ("RSV", value)


class Emoji(Token):
    def __init__(self, value):
```

```
        super().__init__("EMOJI", value)


class String(Token):
    def __init__(self, value):
        super().__init__("STR", value)


class Keyword(Token):
    def __init__(self, value):
        super().__init__("KEY", value)


class Command(Token):
    def __init__(self, value):
        super().__init__("CMD", value)


# Alias tokens for user-defined keywords, i.e., when users define their own
shorthand for commands
class Alias(Token):
    def __init__(self, value):
        super().__init__("ALIAS", value)
```

#lexer.py

```
from tokens import Integer, Float, Operation, Declaration, Variable, Boolean,
Comparison, Reserved


# make varname = 50

class Lexer:
    # while <expr> do <statement>
    # while <expr> do <statement>
    digits = "0123456789"
    letters = "abcdefghijklmnopqrstuvwxyz"
    operations = ["+", "-", "*", "/", "(", ")", "=", "plus", "minus",
"times", "divided by"]
    stopwords = [" "]
    declarations = ["make", "create", "set", "define"]
    boolean = ["and", "or", "not"]
    comparisons = [">", "<", ">=", "<=", "?=", "greater", "less", "equal",
"greater than", "less than",
                   "greater or equal", "less or equal", "equal to", "not
equal"]
    specialCharacters = "><=?"
    reserved = ["if", "elif", "else", "do", "while"]

    def __init__(self, text):
        self.text = text
        self.idx = 0
        self.tokens = []
```

```
        self.char = self.text[self.idx]
        self.token = None

    def tokenize(self):
        while self.idx < len(self.text):
            if self.char in Lexer.digits:
                self.token = self.extract_number()

            elif self.char in Lexer.operations:
                self.token = Operation(self.char)
                self.move()

            elif self.char in Lexer.stopwords:
                self.move()
                continue

            elif self.char in Lexer.letters:
                word = self.extract_word()

                if word in Lexer.declarations:
                    self.token = Declaration(word)
                elif word in Lexer.boolean:
                    self.token = Boolean(word)
                elif word in Lexer.reserved:
                    self.token = Reserved(word)
                else:
                    self.token = Variable(word)

            elif self.char in Lexer.specialCharacters:
                comparisonOperator = ""
                while self.char in Lexer.specialCharacters and self.idx <
len(self.text):
                    comparisonOperator += self.char
                    self.move()

                self.token = Comparison(comparisonOperator)

            self.tokens.append(self.token)

        return self.tokens

    def extract_number(self):
        number = ""
        isFloat = False
        while (self.char in Lexer.digits or self.char == ".") and (self.idx <
len(self.text)):
            if self.char == ".":
                isFloat = True
            number += self.char
            self.move()

        return Integer(number) if not isFloat else Float(number)

    def extract_word(self):
        word = ""
        while self.char in Lexer.letters and self.idx < len(self.text):
            word += self.char
```

```
            self.move()

        return word

    def move(self):
        self.idx += 1
        if self.idx < len(self.text):
            self.char = self.text[self.idx]
```

#parser.py

```
class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.idx = 0
        self.token = self.tokens[self.idx]

    def factor(self):
        if self.token.type == "INT" or self.token.type == "FLT":
            return self.token
        elif self.token.value == "(":
            self.move()
            expression = self.boolean_expression()
            return expression
        elif self.token.value == "not":
            operator = self.token
            self.move()
            output = [operator, self.boolean_expression()]
            return output
        elif self.token.type.startswith("VAR"):
            return self.token
        elif self.token.value in ["+", "plus", "-", "minus"]:
            operator = self.token
            self.move()
            operand = self.boolean_expression()
            return [operator, operand]

    def term(self):
        left_node = self.factor()
        self.move()
        while self.token.value in ["*", "times", "/", "divided"]:
            operator = self.token
            if operator.value == "divided":
                self.move()
                if self.token.value == "by":
                    operator.value += " by"
            self.move()
            right_node = self.factor()
            self.move()
            left_node = [left_node, operator, right_node]
        return left_node
```

```
    def if_statement(self):
        self.move()
        condition = self.boolean_expression()
        if self.token.value == "do":
            self.move()
            action = self.statement()
            return condition, action
        elif self.tokens[self.idx - 1].value == "do":
            action = self.statement()
            return condition, action

    def if_statements(self):
        conditions = []
        actions = []
        if_statement = self.if_statement()
        conditions.append(if_statement[0])
        actions.append(if_statement[1])
        while self.token.value == "elif":
            if_statement = self.if_statement()
            conditions.append(if_statement[0])
            actions.append(if_statement[1])
        if self.token.value == "else":
            self.move()
            self.move()
            else_action = self.statement()
            return [conditions, actions, else_action]
        return [conditions, actions]

    def while_statement(self):
        self.move()
        condition = self.boolean_expression()
        if self.token.value == "do":
            self.move()
            action = self.statement()
            return [condition, action]
        elif self.tokens[self.idx - 1].value == "do":
            action = self.statement()
            return [condition, action]

    def comp_expression(self):
        left_node = self.expression()
        while self.token.type == "COMP" or self.token.value in ["greater",
"less", "equal", "greater than", "less than", "greater or equal", "less or
equal", "equal to", "not equal"]:
            operator = self.token
            self.move()
            if operator.value in ["greater", "less", "equal", "not"]:
                operator.value += " " + self.token.value
                self.move()
            right_node = self.expression()
            left_node = [left_node, operator, right_node]
        return left_node

    def boolean_expression(self):
        left_node = self.comp_expression()
        while self.token.value in ["and", "or"]:
            operator = self.token
```

```
            self.move()
            right_node = self.comp_expression()
            left_node = [left_node, operator, right_node]
        return left_node

    def expression(self):
        left_node = self.term()
        while self.token.value in ["+", "plus", "-", "minus"]:
            operator = self.token
            self.move()
            right_node = self.term()
            left_node = [left_node, operator, right_node]
        return left_node

    def variable(self):
        if self.token.type.startswith("VAR"):
            return self.token

    def statement(self):
        if self.token.type == "DECL":
            self.move()
            left_node = self.variable()
            self.move()
            if self.token.value in ["=", "equals"]:
                operation = self.token
                self.move()
                right_node = self.boolean_expression()
                return [left_node, operation, right_node]
        elif self.token.type in ["INT", "FLT", "OP"] or self.token.value ==
"not":
            return self.boolean_expression()
        elif self.token.value == "if":
            return [self.token, self.if_statements()]
        elif self.token.value == "while":
            return [self.token, self.while_statement()]

    def parse(self):
        return self.statement()

    def move(self):
        self.idx += 1
        if self.idx < len(self.tokens):
            self.token = self.tokens[self.idx]
```

#interpreter.py

```
from tokens import Integer, Float, Reserved

class Interpreter:
    def __init__(self, tree, base):
        self.tree = tree
        self.data = base
```

```
    def read_INT(self, value):
        return int(value)

    def read_FLT(self, value):
        return float(value)

    def read_VAR(self, id):
        variable = self.data.read(id)
        variable_type = variable.type

        return getattr(self, f"read_{variable_type}")(variable.value)

    def compute_bin(self, left, op, right):
        left_type = "VAR" if str(left.type).startswith("VAR") else
str(left.type)
        right_type = "VAR" if str(right.type).startswith("VAR") else
str(right.type)
        if op.value in ["=", "equals"]:
            left.type = f"VAR({right_type})"
            self.data.write(left, right)
            return left  # Return the left node which now holds the variable

        left = getattr(self, f"read_{left_type}")(left.value)
        right = getattr(self, f"read_{right_type}")(right.value)

        if op.value in ["+", "plus"]:
            output = left + right
        elif op.value in ["-", "minus"]:
            output = left - right
        elif op.value in ["*", "times"]:
            output = left * right
        elif op.value in ["/", "divided by"]:
            output = left / right
        elif op.value in [">", "greater than"]:
            output = 1 if left > right else 0
        elif op.value in [">=", "greater or equal"]:
            output = 1 if left >= right else 0
        elif op.value in ["<", "less than"]:
            output = 1 if left < right else 0
        elif op.value in ["<=", "less or equal"]:
            output = 1 if left <= right else 0
        elif op.value in ["?=", "equal to"]:
            output = 1 if left == right else 0
        elif op.value in ["and"]:
            output = 1 if left and right else 0
        elif op.value in ["or"]:
            output = 1 if left or right else 0

        return Integer(output) if (left_type == "INT" and right_type ==
"INT") else Float(output)

    def compute_unary(self, operator, operand):
        operand_type = "VAR" if str(operand.type).startswith("VAR") else
str(operand.type)

        operand = getattr(self, f"read_{operand_type}")(operand.value)
```

```
        if operator.value == "+":
            output = +operand
        elif operator.value == "-":
            output = -operand
        elif operator.value == "not":
            output = 1 if not operand else 0

        return Integer(output) if (operand_type == "INT") else Float(output)

    def interpret(self, tree=None):
        if tree is None:
            tree = self.tree

        if isinstance(tree, list):
            if isinstance(tree[0], Reserved):
                if tree[0].value == "if":
                    for idx, condition in enumerate(tree[1][0]):
                        evaluation = self.interpret(condition)
                        if evaluation.value == 1:
                            return self.interpret(tree[1][1][idx])

                    if len(tree[1]) == 3:
                        return self.interpret(tree[1][2])

                    else:
                        return
                elif tree[0].value == "while":
                    condition = self.interpret(tree[1][0])

                    while condition.value == 1:
                        print(self.interpret(tree[1][1]))

                        # Checking the condition
                        condition = self.interpret(tree[1][0])

                    return

        # Unary operation
        if isinstance(tree, list) and len(tree) == 2:
            expression = tree[1]
            if isinstance(expression, list):
                expression = self.interpret(expression)
            return self.compute_unary(tree[0], expression)

        # No operation
        elif not isinstance(tree, list):
            return tree

        else:

            left_node = tree[0]
            if isinstance(left_node, list):
                left_node = self.interpret(left_node)


            right_node = tree[2]
            if isinstance(right_node, list):
```

```
                right_node = self.interpret(right_node)

            operator = tree[1]
            return self.compute_bin(left_node, operator, right_node)
```

#data.py

```
class Data:
    def __init__(self):
        self.data = {}

    def __getitem__(self, key):
        return self.data[key]

    def __setitem__(self, key, value):
        self.data[key] = value

    def __repr__(self):
        return str(self.data)

    def __str__(self):
        return str(self.data)

    def __len__(self):
        return len(self.data)

    def __iter__(self):
        return iter(self.data)

    def __delitem__(self, key):
        del self.data[key]

    def read(self, id):
        return self.data[id]

    def read_all(self):
        return self.data

    def write(self, id, value):
        self.data[id.value] = value

    def clear(self):
        self.data.clear()

    def copy(self):
        return self.data.copy()

    def get(self, key):
        return self.data.get(key)

    def items(self):
        return self.data.items()
```

```
    def keys(self):
        return self.data.keys()

    def pop(self, key):
        return self.data.pop(key)

    def popitem(self):
        return self.data.popitem()

    def setdefault(self, key, default=None):
        return self.data.setdefault(key, default)

    def update(self, other):
        self.data.update(other)

    def values(self):
        return self.data.values()
```

OUTPUT:

```
Test 1: make a = 5
5
Test 2: make b = 10
10
Test 3: make c = 10 + 5
15
Test 4: make d = 10 - 5
5
Test 5: make e = 10 * 5
50
Test 6: make f = 10 / 5
2.0
Test 7: make g = 1 < 2
1
Test 8: make h = 1 and 0
0
Test 9: make i = 1 or 0
1
Test 10: make j = not 1
0
Test 11: if 2 > 1 do make k = 10
10
Test 12: while a < 5 do make a = a + 1
0
```

FUTURE DEVELOPMENT

To expand the capabilities of the Scribble interpreter, future enhancements may include:

1. **User-Defined Functions**: Allow users to define reusable blocks of code.
2. **Data Structures**: Introduce lists, dictionaries, and other structures for complex data manipulation.
3. **Enhanced Error Messages**: Provide more informative debugging information.
4. **Standard Library**: Add built-in functions for common tasks (e.g., input/output operations).
5. **IDE Integration**: Develop a graphical interface for writing and executing Scribble code with syntax highlighting and real-time feedback.

CONCLUSION

The Scribble programming language and its interpreter serve as a strong foundation for beginners learning programming. The project delivers a functional system capable of tokenizing, parsing, and executing basic programs, thereby providing an engaging learning platform. While the current implementation is tailored to beginners, it sets the stage for future advancements, potentially evolving Scribble into a more robust educational tool.