

PatternScript Compiler

Compiler Construction [CS-2002]

Taqwa Rasheed [22K-4557]

Fatimah Ansari [22K-4538]

Submitted to: Sir Syed Faisal Ali

Submission Date: 5th December

1. Introduction

1.1 Overview

PatternScript is a custom domain-specific language (DSL) designed to simplify the generation of numerical sequences and text-based visual patterns. While general-purpose languages often require verbose syntax for string manipulation and output formatting, PatternScript streamlines these tasks with specialized operators and a distinct, script-like syntax.

1.2 Key Design Features

- **The Stitch Operator (~):** A unique operator dedicated to seamless concatenation of strings and numbers, eliminating the need for explicit casting (e.g., plot "Value: " ~ 5:).
- **Distinct Syntax:** PatternScript utilizes the colon (:) as a mandatory statement terminator and note> for comments, giving it a unique visual identity distinct from C-style languages.
- **Pattern Logic:** The language supports high-level constructs like loop (for iteration) and choose (for pattern matching/switching), utilizing an arrow syntax (->) for clarity.
- **Implicit Typing:** Variables are dynamically typed, supporting Number and String primitive types with automatic inference.

2. Language Specification

2.1 Lexical Rules

The lexical analyzer identifies the following token classes:

- **Keywords:** loop, check, else, choose, default, plot, ask, in
- **Operators:** +, -, *, /, %, ~ (Stitch), ==, !=, <, >, <=, >=, -> (Arrow)
- **Separators:** {, }, (,), :, .. (Range)
- **Comments:** Lines starting with note> are treated as comments and ignored by the parser.
- **Identifiers:** Alphanumeric strings starting with a letter or underscore.
- **Literals:** Integers ([0-9]+) and Double-Quoted Strings ("[^"]*").

2.2 Grammar (BNF)

The following Context-Free Grammar defines the syntax of PatternScript.

```
<program> ::= <stmt_list>
<stmt_list> ::= <stmt>
             | <stmt_list> <stmt>

<stmt> ::= <assign_stmt>
          | <io_stmt>
          | <control_stmt>
          | <loop_stmt>

<assign_stmt> ::= IDENT "==" <expr> ":""

<io_stmt> ::= "plot" <expr> ":"
           | "ask" IDENT ":""

<loop_stmt> ::= "loop" IDENT "in" <expr> "..." <expr> "(" <stmt_list>
              ")"
              | "while" <expr> "(" <expr> ")" <stmt_list>
              | "repeat" <expr> "(" <expr> ")" <stmt_list>
              | "until" <expr> "(" <expr> ")" <stmt_list>

<control_stmt> ::= <check_stmt>
                  | <choose_stmt>

<check_stmt> ::= "check" <expr> "(" <stmt_list> ")"
                | "else" "(" <stmt_list> ")"

<choose_stmt> ::= "choose" <expr> "(" <case_list> <default_case> ")"
                  | "switch" <expr> "(" <case_list> ")"

<case_list> ::= <case_item>
                 | <case_list> <case_item>

<case_item> ::= <literal> "->" <stmt_list>

<default_case> ::= "default" "->" <stmt_list>

<expr> ::= <logic_or>
          | <term>

<logic_or> ::= <logic_and>
               | <logic_or> "||" <logic_and>

<logic_and> ::= <equality>
               | <logic_and> "&&" <equality>

<equality> ::= <relational>
              | <equality> "==" <relational>
              | <equality> "!=" <relational>

<relational> ::= <additive>
                | <additive> "<" <additive>
                | <additive> ">" <additive>
                | <additive> "<=" <additive>
                | <additive> ">=" <additive>
```

```

<additive> ::= <term>
    | <additive> "+" <term>
    | <additive> "-" <term>
    | <additive> "~" <term>

<term> ::= <factor>
    | <term> "*" <factor>
    | <term> "/" <factor>
    | <term> "%" <factor>

<factor> ::= IDENT
    | <literal>
    | "(" <expr> ")"
    | "!" <factor>      // Logical NOT
    | "-" <factor>      // Unary Minus

<literal> ::= NUMBER
    | STRING

```

2.3 Syntax Design Notes

- **Terminator:** The colon (:) acts as the statement terminator.
- **Case Separation:** The arrow (->) separates case literals from their execution blocks in ‘choose’ statements.
- **Precedence:** The grammar is stratified to ensure correct order of operations (e.g., Unary Minus > Multiplication > Addition > Logic).

3. Compiler Implementation (The 6 Phases)

3.1 Phase 1: Lexical Analysis

We implemented the Lexer using Python’s re library. A key challenge was distinguishing between the Greater Than operator (>) and the Comment start (note>). We solved this by ordering the regex rules so that note> is matched first.

- **Artifact Reference:** Please see Appendix A for the handwritten DFA for note> and loop.

3.2 Phase 2: Syntax Analysis

The parser utilizes a **Recursive Descent** strategy (Top-Down). Each non-terminal in the BNF corresponds to a Python function.

- **Error Handling:** The parser checks for missing colons (:) and unbalanced braces { }.
- **Artifact Reference:** Please see Appendix A for the handwritten Parse Trees.

3.3 Phase 3: Semantic Analysis

This phase enforces type safety and logic rules to prevent runtime errors.

- **Symbol Table:** We implemented a symbol table to track variable scope. Variables declared inside a loop block (Scope Level 1) are removed from the table upon exit, ensuring they cannot be accessed globally.
- **Type Compatibility Rules:** The semantic analyzer enforces the following strict rules:
 1. **Arithmetic (+, -, /, %):** Both operands must be of type NUMBER
 2. **Repetition (*):** Supports NUMBER * NUMBER (Math) or STRING * NUMBER (Pattern Repetition).
 3. **Stitching (~):** Accepts mixed types. Numbers are automatically coerced to Strings for concatenation.
 4. **Relational (>, <):** Comparisons are only valid between operands of the same type.
- **Artifact Reference:** Please see Appendix A for the handwritten Symbol Table.

3.4 Phase 4: Intermediate Code Generation (ICG) The compiler translates the Abstract Syntax Tree (AST) into Three-Address Code (TAC). We utilized a Quadruple structure to handle control flow via explicit labels and jumps.

Generation Logic:

- **Assignments:** $x = y + z$ converts to $t1 = y + z$ followed by $x = t1$.
- **Loops:** The high-level loop construct is broken down into initialization, a conditional jump (IF_FALSE), a label for the body (L1), and a GOTO statement.

Example Derivation (From our Compiler Output):

The screenshot shows a code editor interface with a dark theme. On the left, the 'Source Code' tab displays a Python-like script with logic for plotting names based on scores. In the center, the 'Console Output' tab shows the compilation process and the generated Three-Address Code (TAC). The TAC includes assignments like 'name = ASSIGN "Santa"', conditionals like 'IF_FALSE t1 GOTO L1', and loops with labels L1 and L2. The output concludes with 'Santa passed!' and '[Finished Successfully]'. At the bottom, the terminal window shows command-line history for running the compiler and the generated Python file.

```
note> Test Case: Logic
name = "Santa";
score = 8;

check score > 5 {
    plot name ~ " passed!";
} else {
    plot name ~ " failed!";
}

[Compiling...]
[Phase 4: Intermediate Code]
name = ASSIGN "Santa"
score = ASSIGN 8
t1 = score > 5
IF_FALSE t1 GOTO L1
t2 = name ~ " passed!"
PLOT t2
GOTO L2
L1:
t3 = name ~ " failed!"
PLOT t3
L2:
-----
Santa passed!

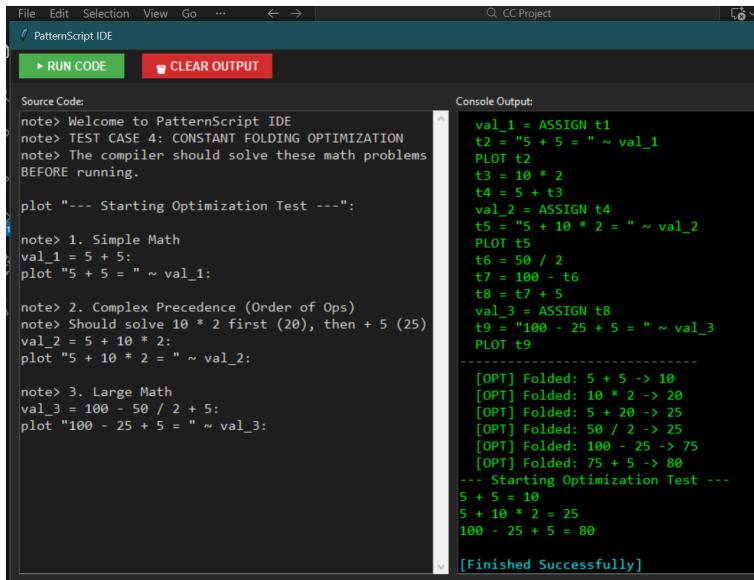
[Finished Successfully]

PS C:\Users\Taqua\OneDrive\Desktop\CC Project\src> python gui.py
```

3.5 Phase 5: Optimization

We implemented **Constant Folding**.

- **Logic:** Expressions containing only literals are computed at compile time.
- **Example:** The expression $x = 2 * 3 + 5$: is compiled directly as $x = 11$; saving runtime cycles.



The screenshot shows a software interface titled "PatternScript IDE". The top menu bar includes "File", "Edit", "Selection", "View", "Go", and "..." along with standard window controls. A toolbar below the menu contains "RUN CODE" and "CLEAR OUTPUT" buttons. The main area is divided into two panes: "Source Code" on the left and "Console Output" on the right. The "Source Code" pane contains a series of mathematical expressions and their corresponding assignments. The "Console Output" pane displays the results of these assignments and the optimization steps taken by the compiler. The output shows how simple expressions like $5 + 5$ and $10 * 2$ are folded into their numerical values, and more complex expressions like $100 - 25 + 5$ are simplified. The final message "[Finished Successfully]" indicates the test was completed successfully.

```
Source Code:
note> Welcome to PatternScript IDE
note> TEST CASE 4: CONSTANT FOLDING OPTIMIZATION
note> The compiler should solve these math problems BEFORE running.

plot "--- Starting Optimization Test ---"

note> 1. Simple Math
val_1 = 5 + 5:
plot "5 + 5 = " ~ val_1

note> 2. Complex Precedence (Order of Ops)
note> Should solve 10 * 2 first (20), then + 5 (25)
val_2 = 5 + 10 * 2:
plot "5 + 10 * 2 = " ~ val_2

note> 3. Large Math
val_3 = 100 - 50 / 2 + 5:
plot "100 - 25 + 5 = " ~ val_3

Console Output:
val_1 = ASSIGN t1
t2 = "5 + 5 = " ~ val_1
PLOT t2
t3 = 10 * 2
t4 = 5 + t3
val_2 = ASSIGN t4
t5 = "5 + 10 * 2 = " ~ val_2
PLOT t5
t6 = 50 / 2
t7 = 100 - t6
t8 = t7 + 5
val_3 = ASSIGN t8
t9 = "100 - 25 + 5 = " ~ val_3
PLOT t9
-----
[OPT] Folded: 5 + 5 -> 10
[OPT] Folded: 10 * 2 -> 20
[OPT] Folded: 5 + 20 -> 25
[OPT] Folded: 50 / 2 -> 25
[OPT] Folded: 100 - 25 -> 75
[OPT] Folded: 75 + 5 -> 80
--- Starting Optimization Test ---
5 + 5 = 10
5 + 10 * 2 = 25
100 - 25 + 5 = 80
[Finished Successfully]
```

3.6 Phase 6: Code Generation (Interpreter)

The final phase is a Python-based interpreter. We developed a **custom GUI IDE** (see screenshots) that intercepts standard print() output to display it in a console window and handles 'ask' input via popup dialogs, creating a user-friendly experience.

4. Testing & Demonstration

Test Case 1: Mathematical Logic (Fibonacci)

Demonstrates: Loops, Assignment, Math.

Input:

```
a = 0:
b = 1:
max = 50:

plot "--- Fibonacci Sequence ---":
loop i in 1..10 {
    check a > max {
        plot "Reached Limit!":
        note> This trick stops the loop by pushing iterator to end
    i = 100:
```

```

    } else {
        plot a:
        temp = a + b:
        a = b:
        b = temp:
    }
}

```

Expected Output:

```

0
1
1
2
3
5
8
13
21
34

```

The screenshot shows the PatternScript IDE interface. The left pane displays the 'Source Code' containing Python-like pseudocode for generating a Fibonacci sequence. The right pane shows the 'Console Output' where the generated sequence is printed. The bottom status bar indicates the command prompt PS C:\Users\Taqua\OneDrive\Desktop\CC Project\src> python gui.py is running multiple times.

```

Source Code:
a = 0:
b = 1:
max = 50:

plot "--- Fibonacci Sequence ---":
loop i in 1..10 {
    check a > max {
        plot "Reached Limit!":
        note> This trick stops the loop by pushing
iterator to end
        i = 100:
    }
    else {
        plot a:
        temp = a + b:
        a = b:
        b = temp:
    }
}

Console Output:
i = ASSIGN 100
GOTO L4
L3:
PLOT a
t3 = a + b
temp = ASSIGN t3
a = ASSIGN b
b = ASSIGN temp
L4:
t4 = i + 1
i = ASSIGN t4
GOTO L1
L2:
-----
--- Fibonacci Sequence ---
0
1
1
2
3
5
8
13
21
34
[Finished Successfully]

PS C:\Users\Taqua\OneDrive\Desktop\CC Project\src> python gui.py

```

Test Case 2: Pattern Generation

Demonstrates: The Repeat Operator (*) and Stitch Operator (~).

Input:

```
note> Triangle Pattern
loop i in 1..8 {
    plot "*" * i:
}

```

Expected Output:

*
* *
* * *
* * * *
* * * * *
* * * * * *

Test Case 3: Logic & Input

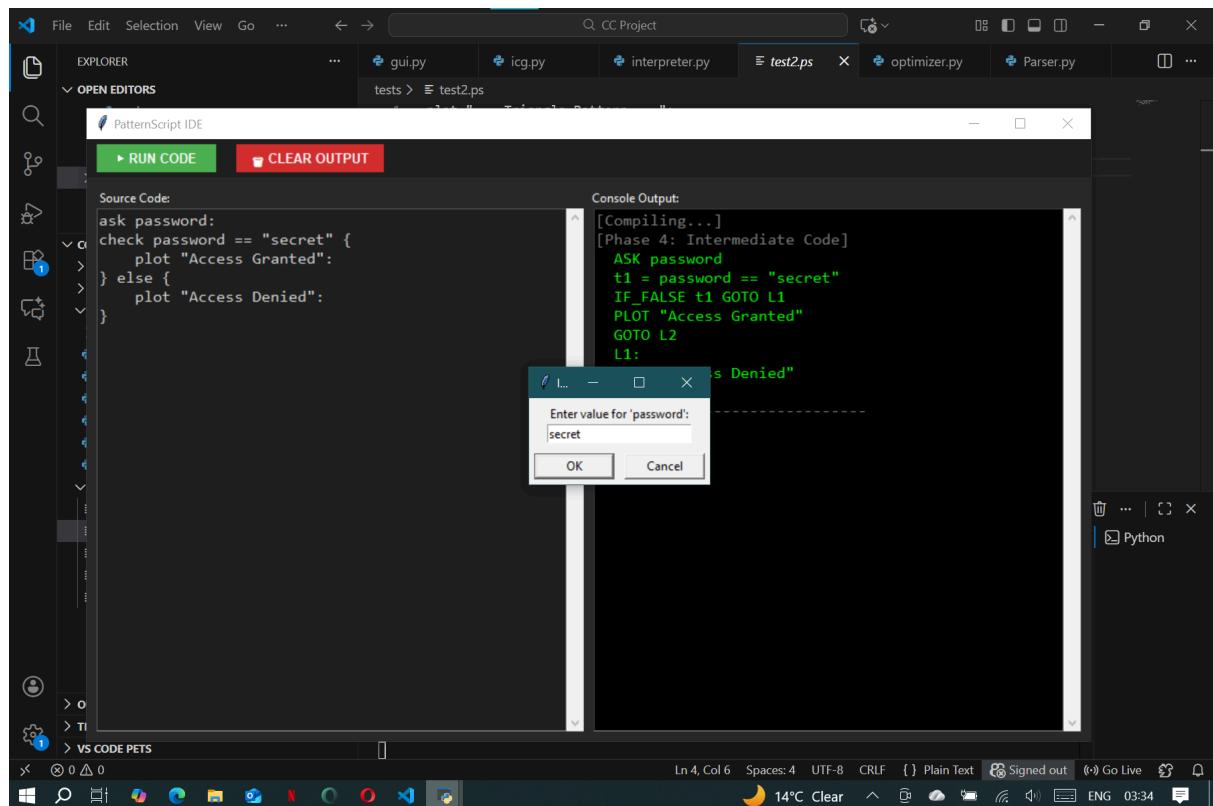
Demonstrates: ask input, check/else logic, and string comparison.

Input:

```
ask password:  
check password == "secret" {  
    plot "Access Granted":  
} else {  
    plot "Access Denied":  
}
```

Expected Output:

(Assuming user types "secret") -> Access Granted



Console Output:

```
[Compiling...]
[Phase 4: Intermediate Code]
ASK password
t1 = password == "secret"
IF_FALSE t1 GOTO L1
PLOT "Access Granted"
GOTO L2
L1:
PLOT "Access Denied"
L2:
-----
Access Granted

[Finished Successfully]
```

5. Reflection

Developing PatternScript provided deep insight into the internal workings of compilers.

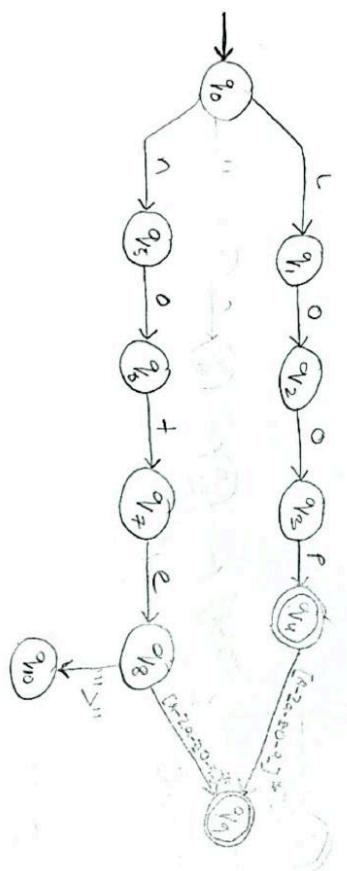
- **Key Learnings:** We learned how critical Operator Precedence is in grammar design. Initially, we struggled with expressions like $3 * 2 + 5$, but layering the grammar rules (Term vs Factor) solved this.
- **Design Choices:** We initially considered a "Grid Loop" (loop x, y) but decided against it to ensure our Intermediate Code Generation (Phase 4) remained robust and bug-free. We instead focused on unique syntax features like the Arrow (\rightarrow) and Stitch (\sim) operator.
- **Future Improvements:** Given more time, we would implement Function Declarations and Arrays to make the language fully Turing-complete.

Appendix A: Handwritten Artifacts

1. DFA Diagram:

DFA (lexical analysis)

DFA for note & loop keyword.



note> states q_1 through q_4 handles 'loop'. if the input stops here it is considered as keyword. if it continues to q_6 it will be considered identifier.

note> state q_6 checks the next character. if it is '}' then we enter comment mode. if its any other character it will be considered as identifier.

2. Parse Trees:

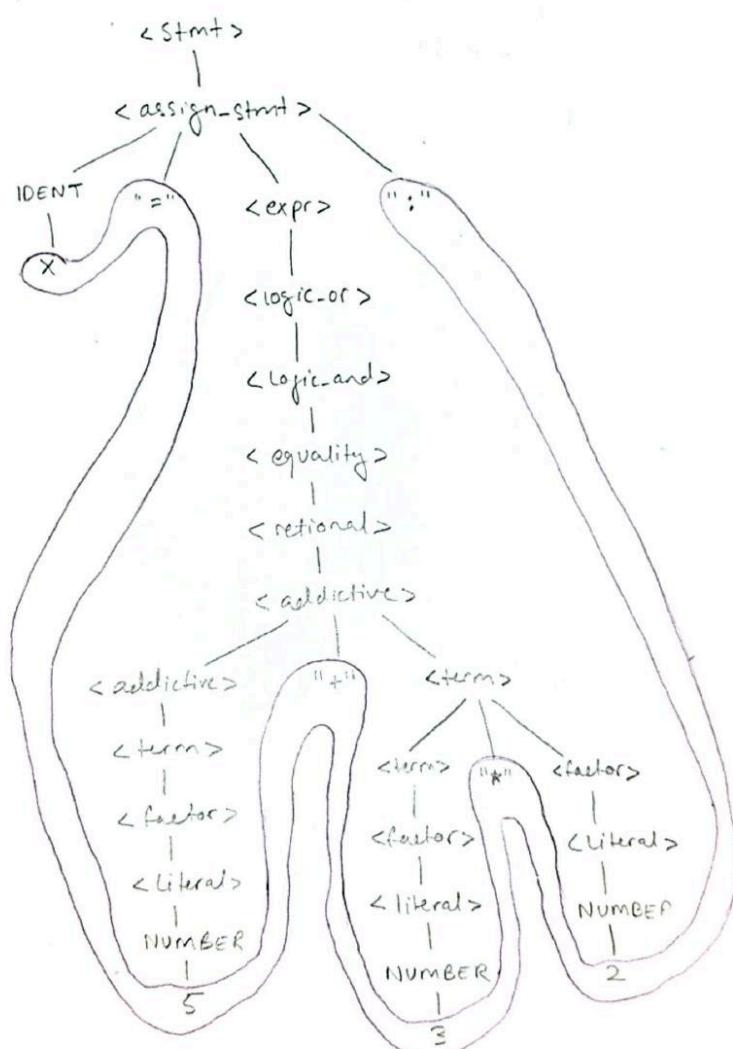
Parse tree (Syntax Analysis)

Tree # 1 : The Math & precedence Tree

Rightmost derivation

Purpose : To show that precedence is correct.

Input :- $x = 5 + 3 * 2$: ($3 * 2$ will happen first).



Tree #2: the loop structure explained

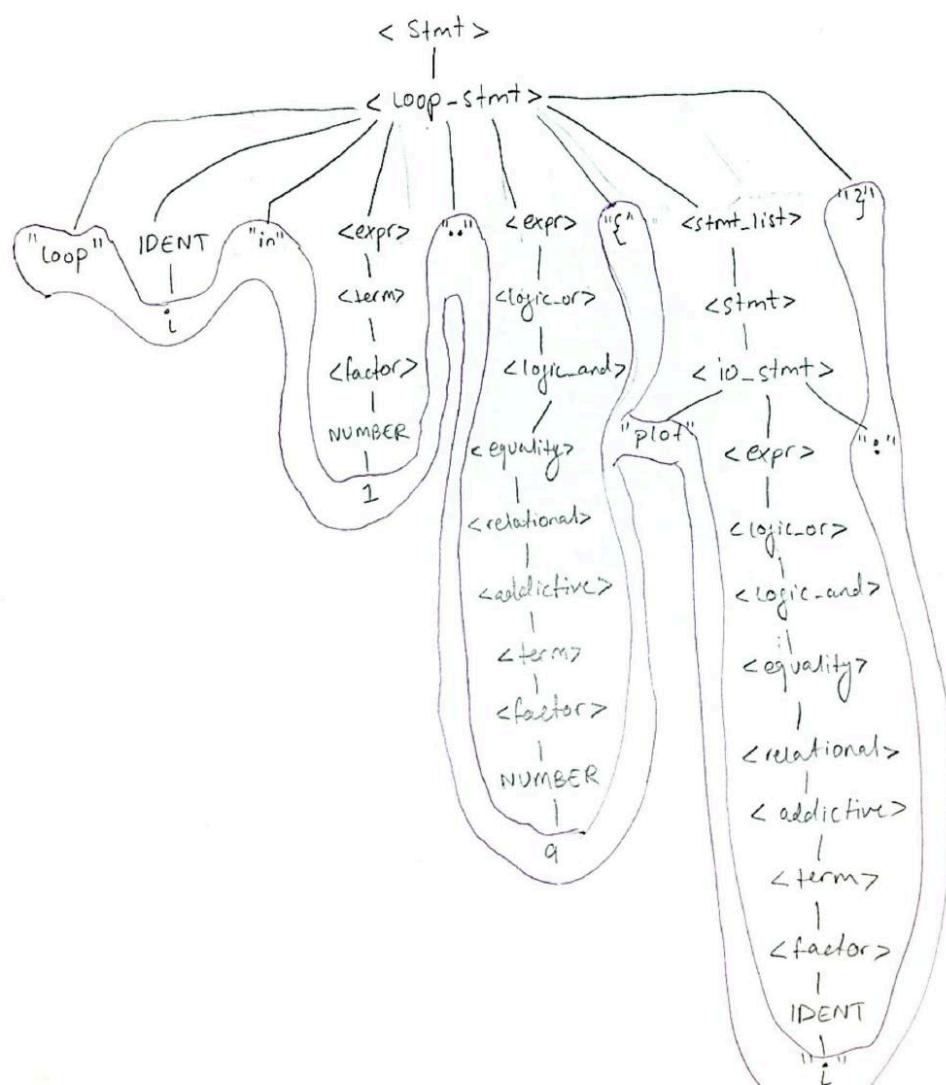
purpose : to show how the compiler parses complex statements.

input code:-

loop i in 1..9 {

plot i;

}



3. Symbol Table:

Symbol Table (Semantic Analysis)

Example code :-

1. notes> symbol table test program
2. global-x = 50;
3. message = "Result: ";
- 4.
5. notes> start of loop scope (Block 1)
6. loop i in 1..3 {
7. local-val = i * 10;
8. global-x = global-x + local-val;
9. }

Semantic Error Example

If code contained: check
message > 5 { ... }

Error: Type mismatch.
Cannot compare
STRING with NUMBER
using relational
operator

Name	Type	Scope Level	Value / Offset
global-x	NUMBER	0 (Global)	50
message	STRING	0 (Global)	"Result:"
i	NUMBER	1 (Loop Block)	1
local-val	NUMBER	1 (Loop Block)	10

Note:

* local-val is declared inside loop. When loop ends (Right brace "{}"), then the local-val row is popped/removed from table.

* global-x is in global scope (level 0). It is accessible inside the loop. The compiler resolves this by checking level 1 first, then level 0.

Appendix B: Source Code

https://github.com/TaqwaRasheed/Mini_Compiler