PatternScript Compiler

Compiler Construction [CS-4031]

Warisha Siddiqui	[22K-4186]
Nashmia Mirza	[22k-4151]
Haiqa Khan	[22K-4665]

Submitted to: Sir Zulfiqar Ali

Submission Date: 30th November

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>
"}"

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

<check_stmt> ::= "check" <expr> "{" <stmt_list> "}"
                 "else" "{" <stmt_list> "}"

<choose_stmt> ::= "choose" <expr> "{" <case_list> <default_case> "}"

<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.

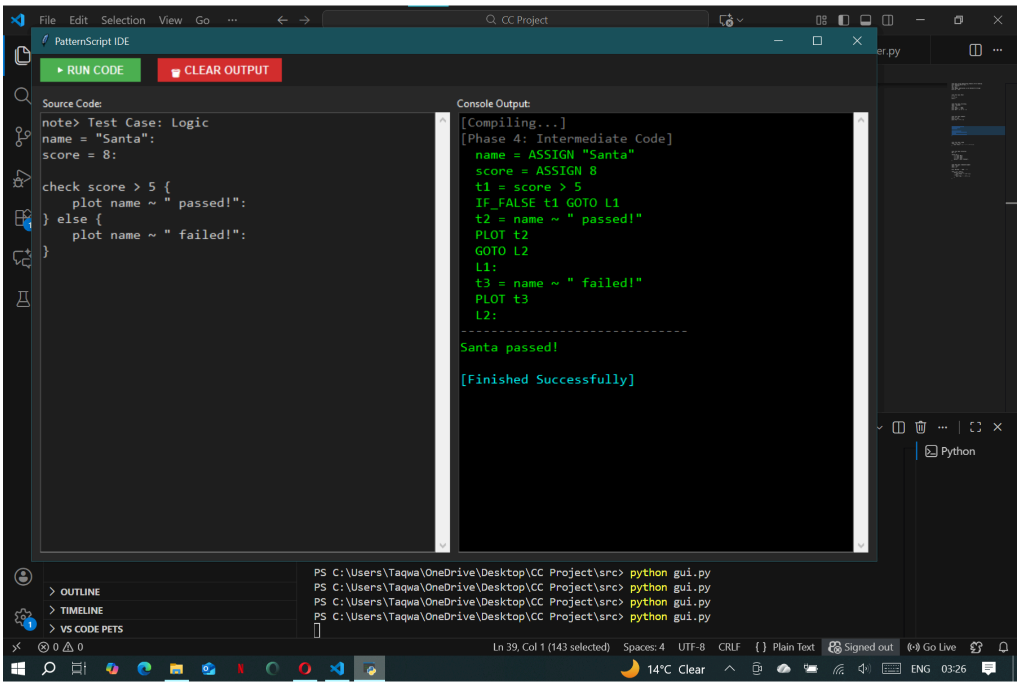
- **Symbol Table:** We implemented a symbol table to track variable scope. Variables declared inside a loop block (Scope Level 1) are removed from the table upon exit, ensuring they cannot be accessed globally.
- **Type Compatibility Rules:** The semantic analyzer enforces the following strict rules:
    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.

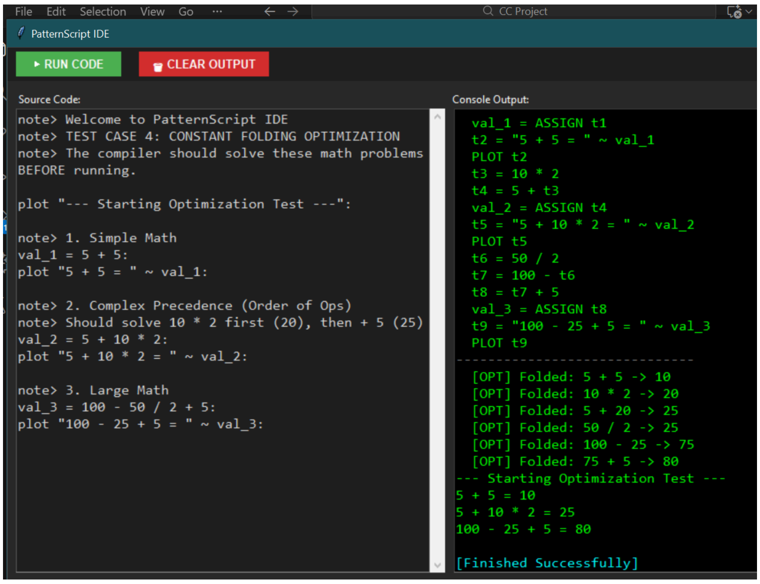
Example Derivation (From our Compiler Output):



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.



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:

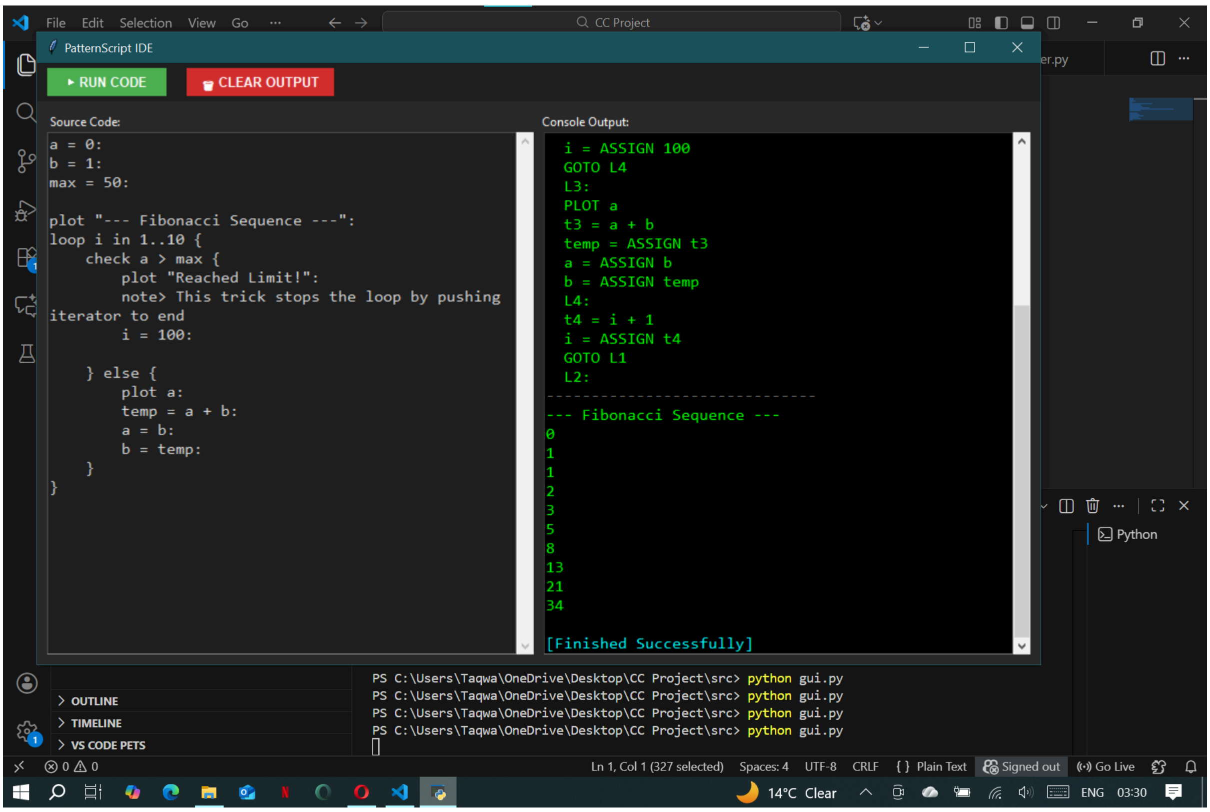
```
a = 0:
b = 1:
max = 50:

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
```



Test Case 2: Pattern Generation

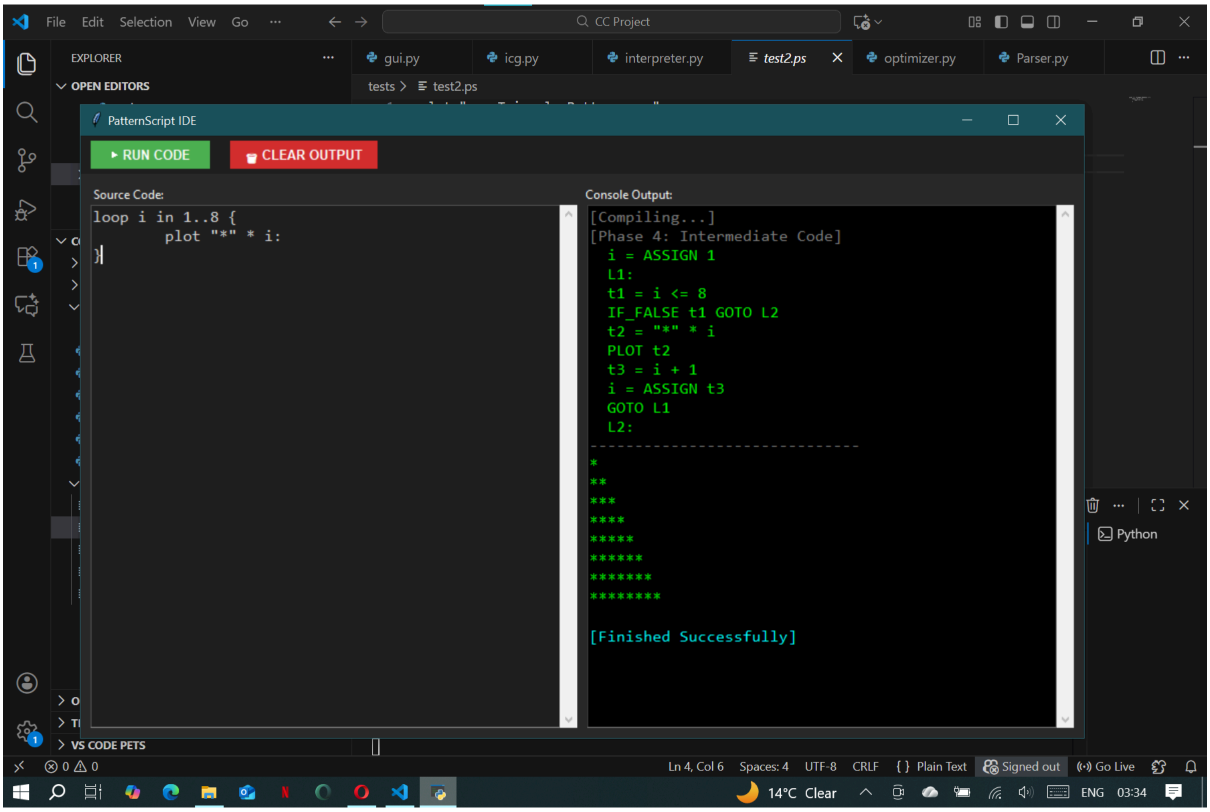
Demonstrates: The Repeat Operator (*) and Stitch Operator (~).

Input:

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

Expected Output:

```
*
**
***
****
*****
******
*******
********
```



Test Case 3: Logic & Input

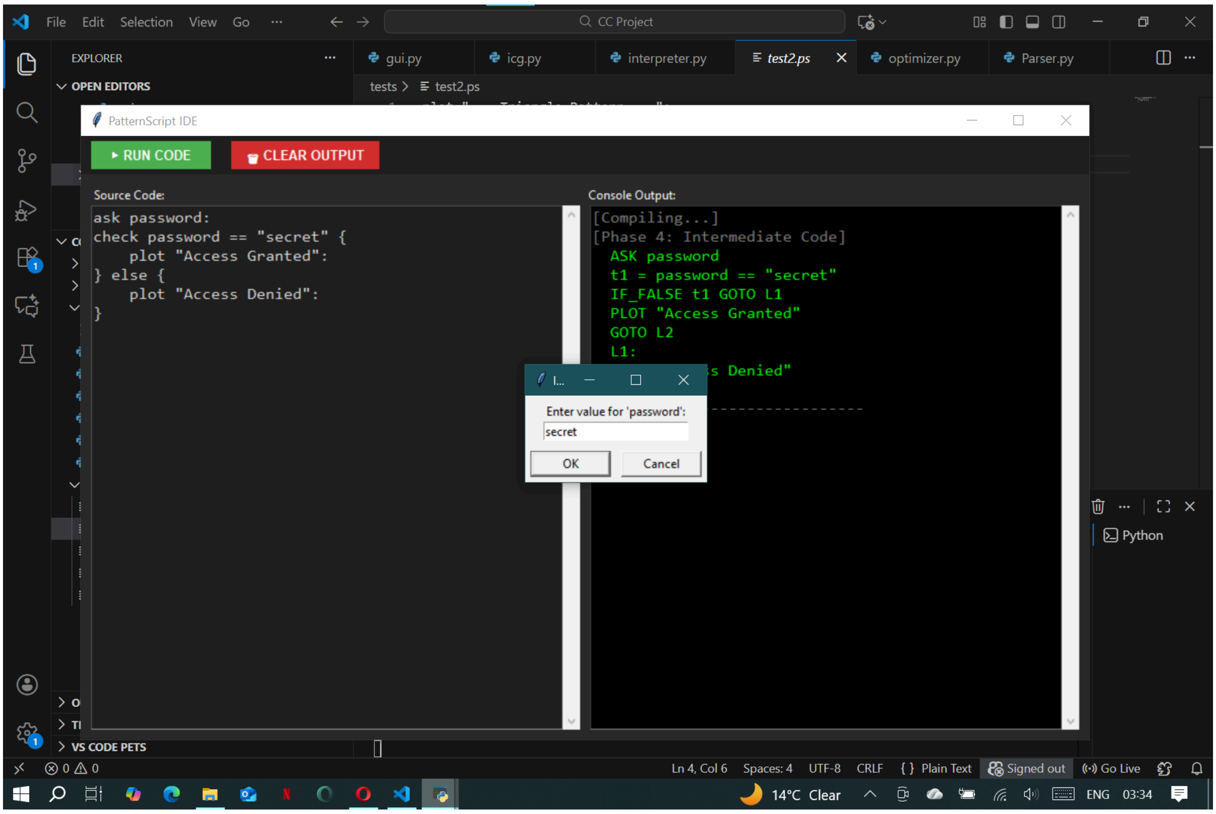
Demonstrates: ask input, check/else logic, and string comparison.

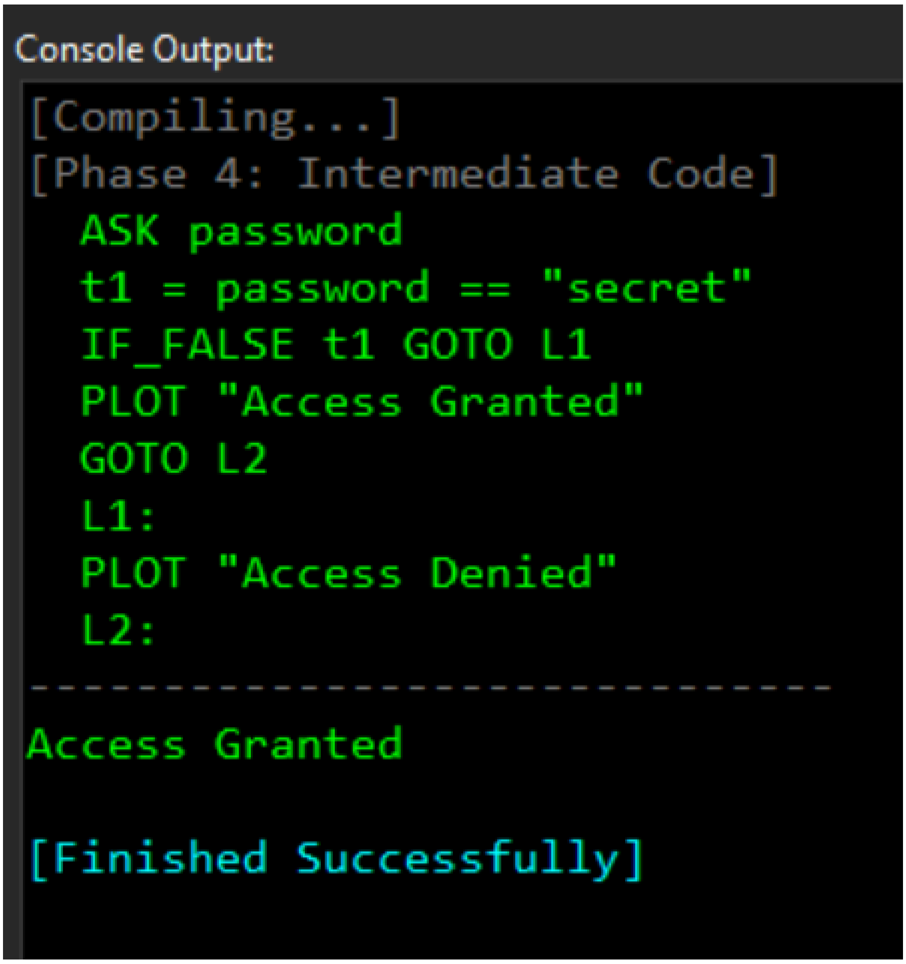
Input:

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

Expected Output:

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





5. Reflection

5.1 Project Overview & Challenges

Developing the **PatternScript** compiler was a comprehensive exercise in understanding the translation pipeline from high-level source code to executable logic. The primary challenge was not just implementing the individual phases, but ensuring seamless integration between them. Our initial design phase involved ambitious features, such as a 2D "Grid Loop," but we quickly realized that complexity in the parser often leads to fragility in the Intermediate Code Generation (ICG) phase. This taught us the importance of iterative design—starting with a robust core (the Recursive Descent Parser) and adding unique features (like the Stitch operator) only once the foundation was stable.

5.2 Key Learnings

The most significant technical hurdle was implementing correct **Operator Precedence**. In early iterations, our parser evaluated expressions strictly left-to-right, causing mathematical errors (e.g., 3 + 2 * 5 evaluating to 25 instead of 13).

- **The Solution:** We learned to stratify the grammar rules. By separating <term> (for multiplication/division) from <additive> (for addition/subtraction) and <factor> (for literals/parentheses), we forced the parser to respect the standard order of operations. This experience solidified our understanding of how Context-Free Grammars (CFGs) directly dictate the shape of the Parse Tree and, consequently, the logic of the program.

We also gained a deep appreciation for the **Symbol Table's** role in scope management. Implementing local scope for loops required us to track when to "enter" and "exit" a block, ensuring that temporary loop variables (like iterators) did not leak into the global scope.

5.3 Design Decisions & Trade-offs

- **The "Grid Loop" Pivot:** We initially aimed to implement a syntactic sugar feature loop x, y to iterate over 2D grids. However, generating the Three-Address Code (TAC) for nested jumps and label management proved highly error-prone within our timeframe. We made the engineering decision to cut this feature to prioritize the stability of the ICG phase. This allowed us to deliver a bug-free compiler rather than a feature-rich but unstable one.
- **Unique Syntax Identity:** To distinguish PatternScript from generic C-clones, we adopted specific syntax choices. Using the Arrow (->) for choosing cases prevents ambiguity with the statement terminator (:). This was a conscious HCI decision to improve code readability and reduce parsing conflicts.

5.4 Future Improvements

While the current version of PatternScript is a powerful linear scripting tool, it is not yet Turing-complete. Given more time, we would implement:

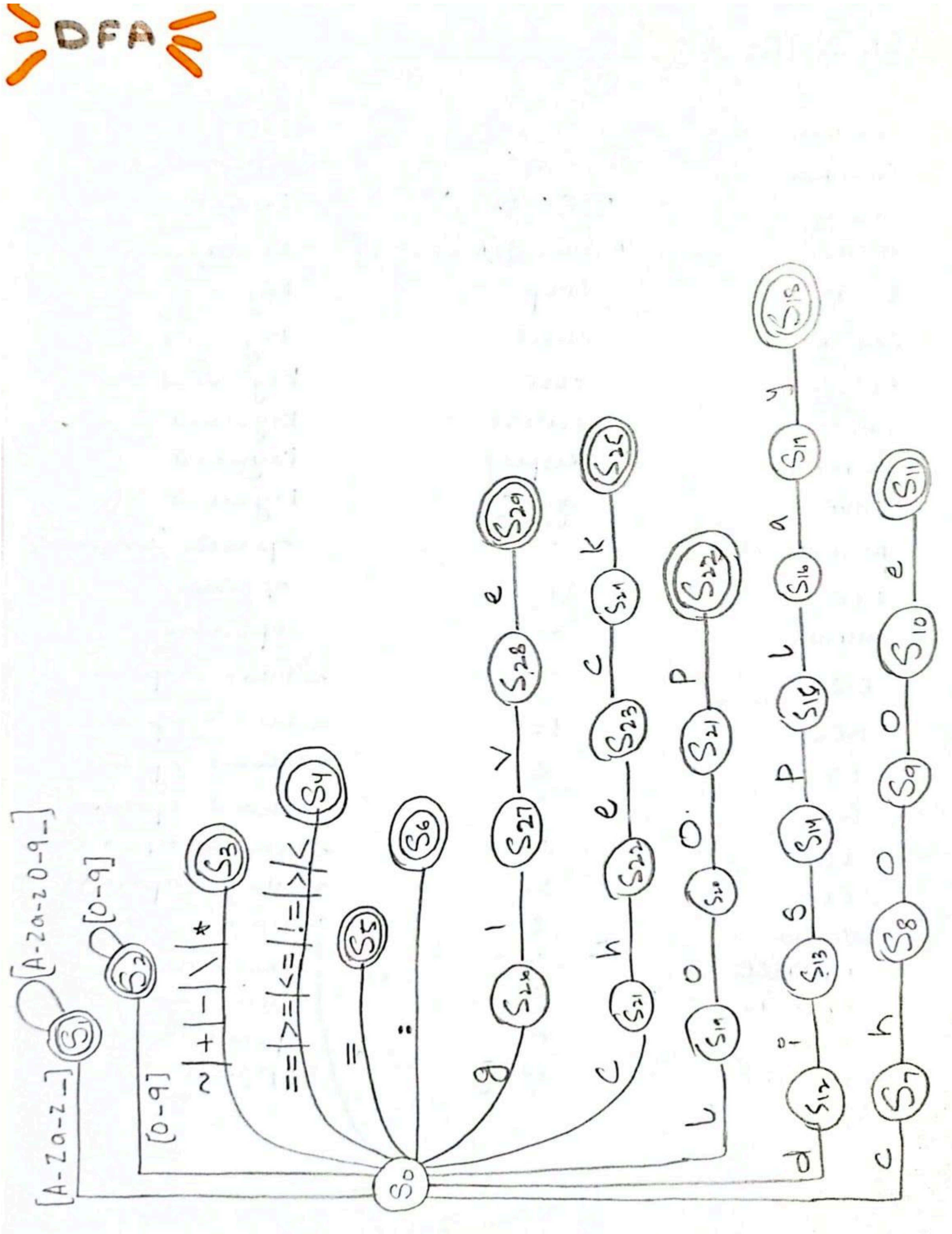
1. **Function Declarations:** Adding call stacks to the interpreter to support reusable code blocks and recursion.
2. **Array Data Structures:** Enabling the storage of lists to support more complex pattern generation algorithms.
3. **File I/O:** Allowing the language to read external data sources rather than relying solely on user input.

5.5 Conclusion

This project successfully demonstrated the end-to-end creation of a compiler. By building the Lexer, Parser, Optimizer, and Interpreter from scratch, we moved beyond theoretical knowledge to practical application. The resulting PatternScript compiler is a functional, optimized, and user-friendly tool that meets all specified requirements.

Appendix A: Handwritten Artifacts

1. **DFA Diagram:**



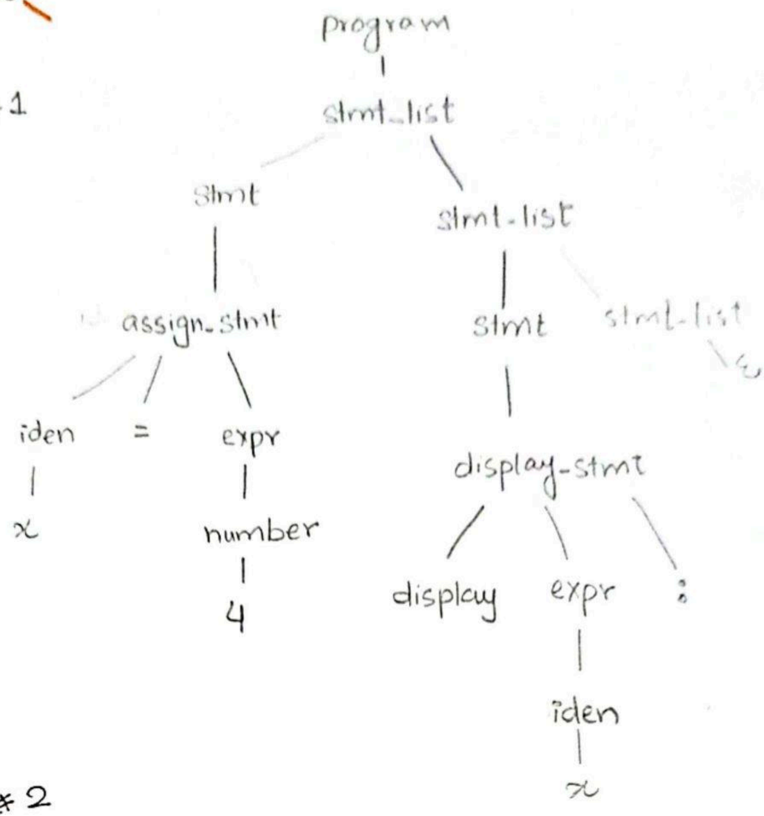
2. **Parse Trees:**

PARSE TREE

Parse Tree #1

x = 4;

display x:

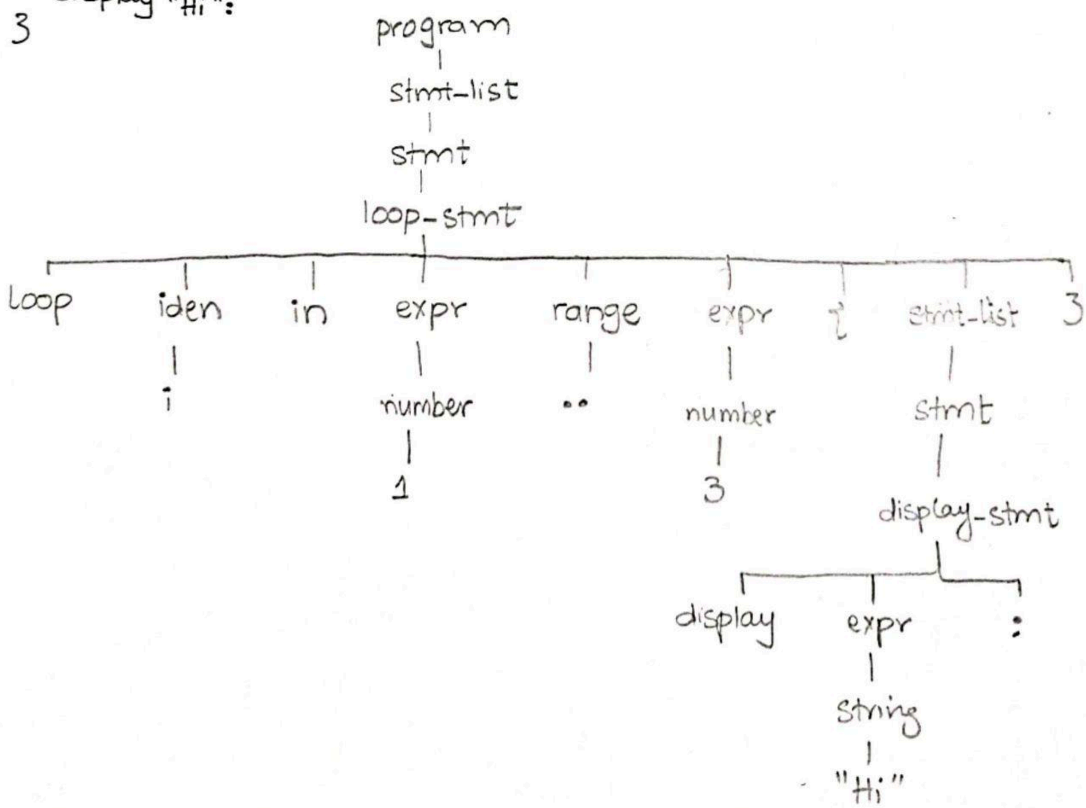


Parse Tree #2

loop i in 1..3 {

display "Hi":

}



3. **Symbol Table:**

Symbol table (Semantic Analysis)

Example code :-

```
note> Symbol table test program
global_x = 50:
message = "Result: ":

note> Start of loop Scope (Block 1)
loop i in 1..3 {
      local_val = i * 10:
      global_x = global_x + local_val:
}
```

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.

CamScanner