Semester Project: Mini-Language Compiler (PyCompiler)

Course: CS4031 - Compiler Construction
Date: Fall 2025
Group Members: [Your Names Here]

1. Language Specification

1.1 Overview

"PyCompiler" is a custom domain-specific language (DSL) designed for integer arithmetic and control flow logic. It supports variable declarations, mathematical operations, while loops, and if conditionals.

1.2 Lexical Rules (Tokens)

The language recognizes the following token patterns:

Token Name	**Pattern / Rule**	**Description**
INTEGER	[0-9]+	Sequence of digits
ID	[a-zA-Z][a-zA-Z0-9_]*	Variable names
LET	let	Keyword for declaration
PRINT	print	Keyword for output
IF	if	Keyword for conditionals
WHILE	while	Keyword for loops
ASSIGN	=	Assignment operator
OP	+, -, *, /	Arithmetic operators
REL_OP	<, >	Relational operators
LBRACE	{	Start of block
RBRACE	}	End of block

SEMI	;	Statement terminator

1.3 BNF Grammar (Syntax)

The syntax is defined by the following Backus-Naur Form (BNF):

```
<Program>      ::= <StatementList>
<StatementList>::= <Statement> | <Statement> <StatementList>
<Statement>    ::= <LetStmt> | <AssignStmt> | <PrintStmt> | <IfStmt> | <WhileStmt>

<LetStmt>      ::= "let" <ID> "=" <Expression> ";"
<AssignStmt>   ::= <ID> "=" <Expression> ";"
<PrintStmt>    ::= "print" <Expression> ";"

<IfStmt>       ::= "if" "(" <Expression> ")" <Block>
<WhileStmt>    ::= "while" "(" <Expression> ")" <Block>
<Block>        ::= "{" <StatementList> "}"

<Expression>   ::= <Term> { ("+" | "-" | "<" | ">") <Term> }
<Term>         ::= <Factor> { ("*" | "/") <Factor> }
<Factor>       ::= <INTEGER> | <ID> | "(" <Expression> ")"
```

2. Compiler Phases (Implementation Details)

Phase 1: Lexical Analysis

Implemented in the Lexer class. It scans the source string character by character. It skips whitespace and identifies keywords (let, if, etc.) versus identifiers.

- **Artifact:** List of Token objects (e.g., Token(LET, 'let'), Token(ID, 'a')).

Phase 2: Syntax Analysis

Implemented in the Parser class using a **Recursive Descent** strategy.

- The parser defines a method for each grammar rule (e.g., statement(), expr(), term()).
- It builds an **Abstract Syntax Tree (AST)** where nodes represent operations (e.g., BinOp, IfStmt, Block).

Phase 3: Semantic Analysis

Implemented in the SemanticAnalyzer class.

- **Symbol Table:** A Set data structure is used to track declared variables.
- **Scope Check:** The analyzer traverses the AST. If a Var node is encountered that is not in

the symbol_table, it raises a "Variable used before assignment" error.

Phase 4: Intermediate Code Generation (TAC)

Implemented in the TACGenerator class.

- Converts the tree-based AST into linear **Three-Address Code**.
- **Temporaries:** Generates t1, t2 for intermediate math results.
- **Labels:** Generates L1, L2 for control flow (jumps) in if and while statements.

Phase 5 & 6: Code Generation (Virtual Machine)

Implemented in the VirtualMachine class.

- Instead of generating assembly, we implemented a VM that executes the TAC instructions directly.
- The VM maintains a memory dictionary (simulating RAM) and a program counter (pc).

3. Reflection

What We Learned

Developing "PyCompiler" from scratch provided deep insight into the internal workings of programming languages. The project demystified the "black box" nature of compilers, revealing them to be a structured pipeline of data transformations.

1. The Power of Recursive Descent Parsing:
We learned how a grammar specification (BNF) directly translates into code. Writing the Parser class taught us that the structure of the code mirrors the structure of the grammar itself. For example, the way expr() calls term() perfectly captures the mathematical concept of precedence. We now understand why syntax errors occur and how the compiler tracks its position within nested structures like blocks or parentheses.
2. The Transition from Tree to Linear Code:
One of the most significant learning curves was moving from Phase 2 (AST) to Phase 4 (TAC). While the AST is excellent for representing the logical hierarchy of code (e.g., "this block belongs to this if statement"), the CPU (or VM) executes instructions linearly. We learned how to flatten complex logic into simple GOTO and conditional jumps, effectively manually compiling high-level logic into assembly-style instructions.
3. State Management in Compilation:
The importance of the Symbol Table became clear during Semantic Analysis. We realized that a compiler doesn't just read code; it must maintain a "memory" of what has happened so far (e.g., which variables are declared). This highlighted the difference between syntax correctness (valid grammar) and semantic correctness (valid logic).

Challenges Faced

1. Operator Precedence and Associativity:
Initially, our parser treated all arithmetic operators with equal weight, leading to incorrect

calculations where 2 + 3 * 5 resulted in 25 instead of 17. We overcame this by layering our parsing functions: separating expr (addition/subtraction) from term (multiplication/division) to enforce the correct order of operations.
2. Handling Control Flow Labels:
Generating Three-Address Code for while loops was complex. We struggled with managing the jump targets—specifically, ensuring that the loop body jumped back to the start condition, while a false condition jumped past the entire body. Debugging infinite loops in our VM required careful tracing of the generated L1 and L2 labels.
3. Left Recursion:
While designing the grammar, we initially encountered infinite recursion issues because our grammar was left-recursive (e.g., A -> A + B). We learned to refactor this into right-recursive rules or iterative loops (using while inside the parser) to fit the Recursive Descent model.

Future Improvements

If we were to extend this project, we would prioritize the following features:

1. **Expanded Type System:** Currently, PyCompiler only supports integers. We would add support for floating-point numbers (float) and text strings (string), which would require a more complex Symbol Table that tracks variable types, not just existence.
2. **Function Definitions:** Implementing functions (def myFunc() { ... }) would be the next major milestone. This would require implementing a Call Stack in our Virtual Machine to handle scope frames, arguments, and return values, moving beyond our current global memory model.
3. **Error Recovery:** Currently, the compiler halts at the first error. We would implement "Panic Mode" recovery to synchronize the parser state, allowing it to report multiple errors in a single run, which is standard in modern compilers.
4. **Optimizations:** We could implement basic optimizations in Phase 5, such as Constant Folding (calculating 3 + 5 during compilation) or Dead Code Elimination (removing code after a return statement), to make the generated bytecode more efficient.