****

**Lab Terminal # 04**

**Submitted by:** Isha EMan

## **Reg #:** fa21-bcs-046

**Explain the core function of your mini-compiler?**

1. **Lexical Analysis (Tokenization)**:
   * **Explanation**: Lexical analysis is the first step of compiling, where the raw source code is broken down into a stream of tokens. Tokens are the smallest units of meaningful code, such as keywords, operators, identifiers, literals, and punctuation.
   * **Process**:
     + The lexer reads the source code character by character and groups them into tokens.
     + For example, for the line int x = 10;, the lexer would produce the following tokens:
       - int (keyword)
       - x (identifier)
       - = (assignment operator)
       - 10 (integer literal)
       - ; (semicolon)
     + This step ensures that the code is divided into manageable parts for the subsequent stages.

**Code Example**:

// Lexer converts source code into tokens

var tokens = lexer.Tokenize("int x = 10;");

the code is properly tokenized, enabling the parser to interpret it correctly.

1. **Parsing (Syntax Analysis)**:
   * **Explanation**: The parser takes the stream of tokens produced by the lexer and checks whether they conform to the grammar rules of the mini-C# language. It builds an **Abstract Syntax Tree (AST)**, a hierarchical representation of the structure of the program, which reflects the program's syntax.
   * **Process**:
     + The parser uses a set of grammar rules to analyze the token stream and determine if it forms valid statements according to the mini-C# language.
     + For example, the parser would take the tokens int x = 10; and check if they conform to a valid declaration statement.
     + It then builds an AST, where each node in the tree represents a program construct (e.g., variable declaration, assignment, expression).
     + The AST provides a structured, visual representation of the program, which makes it easier for the compiler to generate the corresponding low-level code.

**File: Parser.cs**

**// From Parser.cs**

**public AstNode ParseAssignmentStatement() {**

**if (CurrentToken.Type != TokenType.IDENTIFIER) {**

**throw new Exception("Expected Identifier");**

**}**

**string varName = CurrentToken.Value;**

**NextToken();**

**if (CurrentToken.Type != TokenType.ASSIGN) {**

**throw new Exception("Expected '='");**

**}**

**NextToken();**

**AstNode expression = ParseExpression();**

**if(CurrentToken.Type != TokenType.SEMICOLON){**

**throw new Exception("Expected ';'");**

**}**

**NextToken();**

**return new AssignmentNode(varName, expression);**

**}**

**public AstNode ParseExpression() {**

**AstNode term = ParseTerm();**

**while (CurrentToken.Type == TokenType.PLUS || CurrentToken.Type == TokenType.MINUS) {**

**TokenType op = CurrentToken.Type;**

**NextToken();**

**AstNode right = ParseTerm();**

**term = new BinaryOpNode(op,term,right);**

**}**

**return term;**

**}**

* + **Purpose**: The parser ensures that the program is syntactically correct by building an AST. If there are syntax errors, the parser will flag them.

1. **Code Generation**:
   * **Explanation**: After the AST is constructed, the next step is code generation, where the compiler translates the AST into a low-level intermediate representation (IR) that can be processed further or executed.
   * **Process**:
     + The **Code Generator** traverses the AST, node by node, and generates low-level instructions based on the tree structure.
     + For example, if the AST has a node that represents an integer variable int x = 10;, the code generator would generate instructions to allocate memory for the variable x and store the value 10 in it.
     + The generator might also produce assembly-like instructions (or bytecode) to perform operations. For example, an addition operation in the AST would translate to a machine-level addition instruction.
     + The generated code is typically in a form that can be executed directly (like assembly) or further optimized for machine code generation.

**File: CodeGenerator.cs**

**Purpose**: The code generation step ensures that the logical structure of the public void GenerateCode(AstNode node)

{

switch (node)

{

case AssignmentNode assignmentNode:

// Generate code for expression on right-hand side

GenerateCode(assignmentNode.Expression);

// Generate instruction to store result to variable

\_instructions.Add($"store {assignmentNode.VarName}");

break;

case BinaryOpNode binaryOpNode:

GenerateCode(binaryOpNode.Left);

GenerateCode(binaryOpNode.Right);

\_instructions.Add($"{binaryOpNode.Operator.ToString()}");

break;

case IntLiteralNode intLiteralNode:

\_instructions.Add($"load {intLiteralNode.Value}");

break;

}

}

program is translated into actual executable operations.