Compiler Construction Lab

Session: 2021 – 2025

Submitted by:

Name: Yasir Mahmood **Registration No:** 2021-CS-124

Supervised by:

Prof. Laeeq Khan Niazi

Department of Computer Science

University of Engineering and Technology

Lahore Pakistan

Contents

Introduction:

Compilers are fundamental tools in computer science, responsible for translating high-level programming languages into machine-readable code. This project focuses on developing a compiler in C++ that processes source code from lexical analysis to machine code generation. The compiler is designed to handle features such as arithmetic operations, control structures, and function definitions. Additionally, extra functionality, such as switch-case statements, semantic checks, has been implemented to enhance its capabilities.

This report explains the functionality and evolution of the compiler, detailing each phase and the additional features added for improved performance and functionality.

Phase 1: Lexical Analysis(Tokenizer)

Objective:

The purpose of the lexical analysis phase is to scan the source code and convert it into a sequence of tokens. These tokens represent the smallest units of meaningful data, such as keywords, identifiers, operators, and literals.

Implementation:

- The Lexer class is responsible for tokenizing the source code.
- It reads the input code (from a file) and identifies tokens based on predefined rules.
- Tokens are categorized into types like KEYWORD, IDENTIFIER, NUMBER, OPERATOR, etc.

Token Type:

A **token** is a fundamental building block of the lexical analysis phase in a compiler. the enum TokenType categorizes all possible types of tokens that can appear in the source code processed by the compiler. Each TokenType represents a specific kind of token that the lexer can recognize.

```
enum TokenType
{
    T_INT,T_ID,T_NUM,T_IF,T_ELSE,T_RETURN,T_ASSIGN,T_PLUS,T_MINUS,T_MUL,
    T_DIV,T_LPAREN,T_RPAREN,T_LBRACE,T_RBRACE,T_SEMICOLON,T_GT,T_WHILE,
    T_FUNC,T_SWITCH,T_CASE,T_DEFAULT,T_BREAK,T_BOOL,T_TRUE,T_FALSE,T_STRING,
    T_COMMENT,T_COLON,T_EOF
};
```

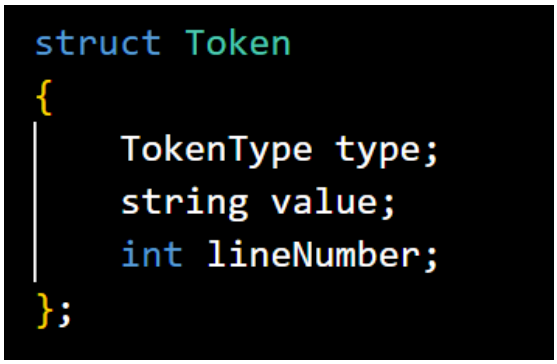
Tokens Explained

1. **T_INT**
   Represents the keyword `int` in the source code, typically used for declaring integer variables.
2. **T_ID**
   Represents an identifier, which is the name of a variable, function, or any user-defined symbol.
3. **T_NUM**
   Represents a numeric literal (e.g., `10`, `42`, `3.14`).
4. **T_IF**
   Represents the keyword `if`, used in conditional statements.
5. **T_ELSE**
   Represents the keyword `else`, which complements an `if` statement.
6. **T_RETURN**
   Represents the keyword `return`, used to exit a function and optionally return a value.
7. **T_ASSIGN**
   Represents the assignment operator `=`.
8. **T_PLUS**, **T_MINUS**, **T_MUL**, **T_DIV**
   Represent the arithmetic operators `+`, `-`, `*`, and `/`, respectively.
9. **T_LPAREN**, **T_RPAREN**
   Represent the left parenthesis `(` and right parenthesis `)`, used for grouping expressions or specifying function calls.
10. **T_LBRACE**, **T_RBRACE**
    Represent the left brace `{` and right brace `}`, used for defining blocks of code.
11. **T_SEMICOLON**
    Represents the semicolon `;`, used to terminate statements.
12. **T_GT**
    Represents the greater-than operator `>`.
13. **T_WHILE**
    Represents the keyword `while`, used for looping constructs.

14. **T_FUNC**
    Represents the keyword `func`, which might be used for defining a function in your language.
15. **T_SWITCH**, **T_CASE**, **T_DEFAULT**, **T_BREAK**
    Represent parts of the `switch-case` statement:
    - `T_SWITCH`: The `switch` keyword.
    - `T_CASE`: The `case` keyword.
    - `T_DEFAULT`: The `default` keyword.
    - `T_BREAK`: The `break` keyword, used to exit a case block.
16. **T_BOOL**, **T_TRUE**, **T_FALSE**
    Represent boolean-related constructs:
    - `T_BOOL`: The `bool` type.
    - `T_TRUE`: The `true` value.
    - `T_FALSE`: The `false` value.
17. **T_STRING**
    Represents a string literal (e.g., `"Hello, World!"`).
18. **T_COMMENT**
    Represents a comment, either single-line or multi-line.
19. **T_COLON**
    Represents the colon `:`, often used in `case` statements or dictionary-like structures.
20. **T_EOF**
    Represents the end-of-file, marking the completion of the input source code.

Struct Token:

The Token structure represents the fundamental unit of data generated during the **lexical analysis phase**. It encapsulates all the necessary information about a token for subsequent phases of the compiler.



Fields in Token:

1. **`TokenType type;`**
   Represents the category/type of the token (e.g., T_INT, T_NUM, T_IF).

2. **`string value;`**
   Holds the actual value of the token as a string (e.g., "int", "5", "x").
   - For a keyword like `int`, `value` would be `"int"`.
   - For a number like `42`, `value` would be `"42"`.
3. **`int lineNumber;`**
   Indicates the line number in the source code where the token appears. This is useful for debugging, error reporting, and diagnostics.

Role of the Token **Struct:**

- It is used to pass data between the **Lexer** and subsequent phases (e.g., Parser).
- Each Token is a single piece of information that the compiler can process systematically.

Class Lexer:

The Lexer is responsible for the **lexical analysis phase**, which is the first phase of compilation. Its job is to:

1. Read the source code character by character.
2. Recognize patterns to identify valid tokens.
3. Return a sequence (vector) of Token objects to be used by the parser.

Key Fields in Lexer:

1. **`string src;`**
   Holds the source code being analyzed.
2. **`size_t pos;`**
   Tracks the current position in the source code.
   - It helps the lexer know which character to process next.
3. **`int lineNumber;`**
   Tracks the current line number in the source code.
   - This is incremented whenever a newline character (\n) is encountered.

Key Functions in Lexer:

1. **Constructor (Lexer(const string &src)):**
   - Initializes the lexer with the source code (src).
   - Sets the initial position (pos) to 0 and the line number (lineNumber) to 1.
2. **vector<Token> tokenize()**
   This is the core function of the lexer, which:
   - Iterates through the source code character by character.
   - Identifies tokens based on the current character or sequence of characters.
   - Creates Token objects for recognized patterns and adds them to the tokens vector.

3. **Token Consumption Functions:**
   These helper functions extract specific types of tokens:
   - **consumeNumber()**
     Extracts numeric tokens by recognizing sequences of digits (e.g., 123).
   - **consumeWord()**
     Extracts words that could be identifiers (e.g., variable names) or keywords (e.g., int, if).
   - **consumeString()**
     Extracts string literals enclosed in double quotes ("string").
4. **Keyword Recognition:**
   In the tokenize() function, consumeWord() is used to identify keywords like int, if, else, etc., and assign them specific TokenTypes.
5. **Handling Special Characters:**
   The tokenize() function handles one-character tokens like +, -, {, }, ;, etc., using a switch statement.
6. **Handling Comments and Whitespace:**
   - Skips single-line comments starting with //.
   - Skips whitespace and newline characters, incrementing the lineNumber for each newline.
7. **Error Reporting:**
   If the lexer encounters an unexpected character, it outputs an error message with the character and its line number, then terminates the program.
8. **End-of-File Token:**
   Adds a special token (T_EOF) at the end of the token list to indicate the end of the source code.

Phase 2: Syntax Analysis (Parsing):

Objective:

The parser validates the syntactic structure of the code based on predefined grammar rules. It ensures the code follows proper structure and builds an Abstract Syntax Tree (AST) for further processing.

Implementation:

- The Parser class takes the tokenized input and processes it to verify syntax.
- Grammar rules for statements (e.g., variable declarations, expressions, control structures) are defined and validated.
- Errors like missing semicolons or unbalanced parentheses are identified and reported.

Parser Class:

The Parser class is responsible for:

- Verifying the syntactic correctness of the token stream.
- Generating intermediate code for various language constructs.
- Utilizing a **symbol table** for managing variables and **intermediate code generator (ICG)** for generating intermediate representations.

Constructor:

```
// Constructor
Parser(const vector<Token> &tokens, SymbolTable &symTable, IntermediateCodeGenerator &icg)
    : tokens(tokens), pos(0), symTable(symTable), icg(icg) {}
```

Inputs:

- `tokens`: The list of tokens produced by the lexer.
- `symTable`: A symbol table for managing declared variables and their types.
- `icg`: An intermediate code generator to produce low-level representations of the code.

Purpose: Initializes private members (`tokens`, `pos`, `symTable`, `icg`) for parsing operations.

Parsing Program:

- Iterates through the token stream until the **end-of-file (EOF)** token is encountered.
- Delegates parsing individual statements to the `parseStatement` method.

```
void parseProgram()
{
    while (tokens[pos].type != T_EOF)
    {
        parseStatement();
    }
}
```

Statement Parsing:

```
void parseStatement()
{
    if (tokens[pos].type == T_INT)
    {
        parseDeclaration();
    }
    else if (tokens[pos].type == T_ID)
    {
        parseAssignment();
    }
    else if (tokens[pos].type == T_IF)
    {
        parseIfStatement();
    }
    else if (tokens[pos].type == T_RETURN)
    {
        parseReturnStatement();
    }
    else if (tokens[pos].type == T_LBRACE)
    {
        parseBlock();
    }
    else if (tokens[pos].type == T_WHILE)
    {
        parseWhileStatement();
    }
```

This method handles various types of statements based on the current token:

- **Example**: If the current token is `T_INT`, it delegates to `parseDeclaration`.

- Ensures syntactic correctness by choosing the appropriate method for each statement type (e.g., `parseAssignment`, `parseIfStatement`).
- Handles errors when encountering unexpected tokens, exiting with a message.

Declaration Parsing:

```
void parseDeclaration()
{
    expect(T_INT);
    string varName = expectAndReturnValue(T_ID);
    symTable.declareVariable(varName, "int");
    expect(T_SEMICOLON);
}
```

Handles the declaration of variables:

- For Example, ensures the `int` keyword is followed by a valid identifier and a semicolon.
- Registers the variable in the symbol table with its type.

Assignment Parsing:

```
void parseAssignment()
{
    string varName = expectAndReturnValue(T_ID);
    symTable.getVariableType(varName); // Ensure t
    expect(T_ASSIGN);
    string expr = parseExpression();
    icg.addInstruction(varName + " = " + expr); //
    expect(T_SEMICOLON);
}
```

Handles assignment statements:

- Ensures the variable being assigned is declared.
- Parses the expression on the right-hand side.
- Generates intermediate code for the assignment.

Phase 3: Semantic Analysis:

Objective:

Semantic analysis ensures the program is semantically correct by checking for type consistency, variable declarations, and scope rules.

Implementation:

- The `SymbolTable` class is integrated with the parser to track declared variables, their types, and scopes.
- Type-checking mechanisms are implemented to ensure operations are valid for given types.

Member Variables:

```
private:
    map<string, string> symbolTable;
```

- This is a `std::map` that associates a variable name (`string`) with its type (`string`).
- Example: If the program contains `int x;`, the map might store `{"x", "int"}`.

Public Methods:

```
void declareVariable(const string &name, const string &type)
{
    if (symbolTable.find(name) != symbolTable.end())
    {
        throw runtime_error("Semantic error: Variable '" + name + "' is already declared.");
    }
    symbolTable[name] = type;
}

string getVariableType(const string &name)
{
    if (symbolTable.find(name) == symbolTable.end())
    {
        throw runtime_error("Semantic error: Variable '" + name + "' is not declared.");
    }
    return symbolTable[name];
}

bool isDeclared(const string &name) const
{
    return symbolTable.find(name) != symbolTable.end();
}
```

1. declareVariable:

- **Purpose:** Adds a new variable and its type to the symbol table.
- **Process:**

    1. Check if the variable is already declared using `symbolTable.find(name)`.
    2. If found, throw a semantic error (redeclaration is not allowed).
    3. Otherwise, insert the variable into the `symbolTable`.

- **Error Handling:**

    - If a variable is already declared, it throws a runtime error with an appropriate message.

2. getVariableType:

- **Purpose:** Retrieves the type of a given variable.
- **Process:**

    1. Check if the variable exists in the `symbolTable` using `find(name)`.
    2. If not found, throw a runtime error (variable is undeclared).
    3. If found, return its type.

- **Error Handling:**

- Throws a semantic error if the variable is not declared.

3. isDeclared:

- **Purpose:** Checks whether a variable is already declared.
- **Process:**
  1. Use `find(name)` to check if the variable exists in the `symbolTable`.
  2. Return `true` if it exists, `false` otherwise.
- **Use Case:**
  - Often used in situations where you want to check if a variable exists without throwing an error.

Purpose of the Symbol Table:

- The symbol table acts as a data structure that stores information about identifiers (e.g., variables, functions) in the program.
- It helps in:
  - **Declaring identifiers:** Keeping track of declared variables, ensuring no redeclaration.
  - **Type checking:** Validating the types of variables during operations.
  - **Scope management:** (Though not explicitly handled here, scopes could be integrated in a more advanced version.)

Phase 4: Intermediate Code Generation

Objective:

Intermediate code generation converts the high-level source code into a simplified representation, typically three-address code (TAC). This serves as an abstraction for generating machine code.

Implementation:

- The `IntermediateCodeGenerator` class generates TAC for statements.
- Control flow constructs like `if`, `while`, and `switch` are translated into intermediate instructions.

This phase acts as a bridge between the high-level source code and the low-level machine code (or assembly). The goal is to generate an intermediate representation (IR) that is simpler than the source code but retains enough information for optimization and translation to target code. Here's a breakdown of the class:

Member Variables:

```
vector<string> instructions;
int tempCount = 0;
```

- **vector<string> instructions**
  - This is a collection of strings, where each string represents an intermediate code instruction (e.g., `t1 = a + b`).
  - Intermediate code typically uses **three-address code (TAC)** or similar forms, which consist of simple instructions that perform one operation (e.g., assignment, arithmetic, or branching).

- **int tempCount = 0**
  - A counter to generate unique temporary variable names (e.g., `t0`, `t1`, etc.).
  - Temporary variables are used to store intermediate results during computations.

Methods:

```
string newTemp()
{
    return "t" + to_string(tempCount++);
}
```

newTemp():

- **Purpose:** Generates a new temporary variable name, such as `t0`, `t1`, and so on.
- **Use Case:** When a complex expression is broken down into smaller parts, each intermediate result is stored in a temporary variable.

```
void addInstruction(const string &instr)
{
    instructions.push_back(instr);
}
```

addInstruction(const string &instr):

- **Purpose:** Adds a new intermediate code instruction to the `instructions` list.
- **Use Case:** Each time a source code statement or expression is translated into IR, the resulting instruction is stored.

```
string getInstructionsAsString() const
{
    string result;
    for (const auto &instr : instructions)
    {
        result += instr + "\n";
    }
    return result;
}
```

getInstructionsAsString():

- **Purpose:** Returns all the intermediate code instructions as a single string, with each instruction separated by a newline.
- **Use Case:** Useful for debugging or outputting the IR in a readable format.

```
// Returns all instructions as a vector of strings
vector<string> getInstructionsAsVector() const
{
    return instructions;
}
```

getInstructionsAsVector():

- **Purpose:** Returns the `instructions` as a vector of strings, allowing further manipulation or analysis.
- **Use Case:** Useful for optimization passes or writing to external files.

```
void printInstructions()
{
    for (const auto &instr : instructions)
    {
        cout << instr << endl;
    }
}
```

printInstructions():

- **Purpose:** Prints each intermediate code instruction to the console.
- **Use Case:** Debugging during development to view the generated IR.

Phase 5: Machine Code Generation

Its primary objective is to translate the **Intermediate Representation (IR)** into **machine-level instructions** that can be executed directly by the target system. These instructions are either assembly code or low-level machine code, depending on the specific requirements.

Objective:

- **Translation of IR to Low-Level Code:** Convert high-level intermediate instructions (IR) into target-specific low-level instructions (assembly or machine code).
- **Optimization for Hardware:** Ensure the generated code is efficient for the target architecture, making use of available registers and instruction sets.
- **Output to File:** Save the machine code in a file for execution or further use.

Implementation:

- **`generateMachineCode(vector<string> irInstructions)`**
  - Translates each IR instruction into assembly-like machine instructions.
  - Handles:
    - Arithmetic operations (`ADD`, `SUB`, etc.)
    - Memory operations (`LOAD`, `STORE`)
    - Conditional branches (`CMP`, `JLE`, etc.)
    - Labels (`L1:`, etc.)
  - Stores the machine instructions in an internal list.
- **`printMachineInstructions()`**
  - Prints the generated machine code to the console for debugging or visualization.
- **`storeMachineCodeToFile(const string &filename)`**
  - Writes the machine code to a file, which can be executed or further processed.

Additional Features:

The compiler has been extended to handle the following additional features:

1. Control Statements

- **if Statements**: The compiler can now process conditional statements, generating appropriate branch and comparison instructions in machine code.
- **else Blocks**: Handles the alternate execution path by generating labels to manage control flow.
- **while Loops**: Supports iterative constructs, with the generation of labels for loop start and end, along with conditional jump instructions.

2. Comments

- Supports inline and block comments in the source code. These are ignored during intermediate and machine code generation to maintain clean and optimized code output.

3. Nested Control Flow

- Allows nesting of if-else statements and loops. Proper labels are generated to ensure correct flow of execution in complex scenarios.

4. Improved Label Management

- The extended system uses unique label generation to handle complex branching and looping scenarios efficiently, avoiding conflicts in larger programs.

5. Support for Complex Expressions

- Handles arithmetic and logical expressions within control statements like if conditions and while loops.

Conclusion

This compiler project showcases the development of a fully functional compiler that translates high-level source code into machine-readable instructions. With the inclusion of additional features like `switch-case` handling, enhanced error detection, and file-based I/O, it demonstrates an understanding of compiler design principles and practical implementation skills.