Compiler Construction

CS-471L: Compiler Construction Lab

Project Supervisor

Mr. Laeeq Khan Niazi

Developer:

Amad Irfan	2021-CS-25

University of Engineering and Technology
Lahore, Pakistan

Contents

1 Introduction

1.1 Purpose and Motivation

The primary objective of this project is to design a custom compiler, showcasing the complete process of translating high-level language code into low-level machine code. Compilers are pivotal tools in software development, enabling programmers to write efficient, portable, and maintainable applications. This project bridges the gap between theoretical knowledge of language processing and practical implementation.

Constructing a compiler provides insight into crucial aspects like code generation, optimization, and error handling. The motivation stems from the curiosity to understand how human-readable code is transformed into machine-executable instructions. By designing a compiler, this project not only reinforces concepts such as automata and grammar but also highlights optimization techniques and error-detection mechanisms implemented in modern compilers.

1.2 Objectives

The project aims to:

- Implement a compiler capable of translating high-level language into machine-readable code.
- Develop a grammar-based parser to validate syntactic structures and handle errors effectively.
- Create a module for intermediate representation (IR) generation, ensuring platform independence and enabling optimizations.
- Design a modular compiler architecture encompassing all key phases of compilation.
- Provide detailed compile-time feedback, including warnings and suggestions, to aid user understanding of the source code.

1.3 Features

This compiler includes the following features:

- **Tokenization:** The lexer converts the source code into tokens, which are the smallest units of meaning, such as keywords, identifiers, operators, and punctuation marks.
- **Comment Handling:** Supports both single-line (`//`) and multi-line (`/* ... */`) comments, which are ignored during compilation but help with documentation and debugging.
- **Whitespace Management:** Ignores unnecessary whitespace while still tracking line numbers for accurate error reporting and debugging.
- **Recognition of Literals and Identifiers:** The lexer identifies various types of literals, including numeric literals (integers and floating-point numbers), string literals, character literals, and variable names.
- **Keyword Recognition:** Recognizes common programming keywords such as `int`, `float`, `bool`, `string`, `if`, `else`, `while`, and `return`.
- **Syntax Error Reporting:** Generates meaningful error messages with line numbers, allowing the programmer to easily locate and fix syntax issues.

- **Intermediate Representation (TAC):** Converts the parsed source code into a Three-Address Code (TAC), providing a platform-independent intermediate representation of the program.
- **Assembly Code Generation:** The compiler translates TAC into basic assembly-like instructions, ready for further optimization or direct execution on the target machine.

In addition to the above, the compiler supports the following advanced features:

- **Floating-Point Number Support:** Handles `float` literals, supporting both integer and floating-point data types for mathematical operations and expressions.
- **Boolean Support:** Recognizes and processes `bool` literals, including the values `true` and `false`, enabling conditional logic and boolean operations.
- **String and Character Support:** Supports string literals (e.g., `"Hello, World!"`) and character literals (e.g., `'a'`), allowing the handling of textual data.
- **Function Support:** Allows the definition and invocation of functions, including support for parameters, return values, and function calls. The compiler handles both built-in and user-defined functions, enabling modular and reusable code.
- **Control Flow Structures:** Handles common control flow structures like `if`, `else`, `while`, and `for` loops, allowing complex decision-making and iteration in the code.
- **Error Handling and Debugging Support:** In addition to syntax errors, the compiler provides warnings and debug information related to runtime errors making debugging easier for developers.

1.4 Compiler Overview

A compiler is a software tool that converts programs written in high-level programming languages into low-level machine code. It plays a critical role in enabling software development for diverse platforms, optimizing performance, and detecting errors. This project encapsulates the complete workflow of a compiler, from lexical analysis to assembly code generation.

1.5 Phases of Compilation

The compilation process consists of the following phases:

1. **Lexical Analysis:** Reads the source code and tokenizes it into meaningful units.
2. **Syntax Analysis:** Validates the grammatical structure and generates a parse tree.
3. **Semantic Analysis:** Checks the logical consistency and type correctness of the program.
4. **Intermediate Code Generation:** Produces an IR for platform-independent optimization and translation.
5. **Optimization:** Enhances the IR for improved runtime performance.
6. **Code Generation:** Translates the optimized IR into target machine code.

1.6 Applications of Compilers

Compilers are utilized in:

- Translating programming languages (e.g., from C++ to Java bytecode).
- Developing high-performance applications with tools like GCC and LLVM.
- Programming embedded systems with constrained resources.
- Optimizing machine learning frameworks such as TensorFlow.

2 Implementation Details

2.1 Implemented Compiler Phases

- **Lexer:** Tokenization of source code.
- **Parser:** Syntax analysis to generate parse tree.
- **Symbol Table:** Stores identifiers and their attributes.
- **TAC:** Intermediate code generation (Three Address Code).
- **Assembly Generation:** Translation to machine code or assembly.

2.2 Enums and Symbol Table

In the parser implementation, **enums** are used to define token types such as keywords, operators, and literals. These enums categorize tokens during lexical analysis, allowing the parser to identify and process elements of the source code efficiently. For example, the enum values could include:

```
enum TokenType {
    KEYWORD_INT,
    KEYWORD_BOOL,
    OPERATOR_ASSIGN,
    LITERAL_INT,
    LITERAL_STRING,
    IDENTIFIER,
    // Other token types...
};
```

A critical component in managing variable information is the `SymbolTable` class. This class stores variables with their associated names, types, and scope levels. It helps prevent variable redefinition and ensures that variables are properly declared before use. The symbol table is typically implemented using an unordered map for efficient lookup.

```
class SymbolTable {
private:
    std::unordered_map<std::string, SymbolInfo> table;

public:
    void addSymbol(std::string name, std::string type, int scopeLevel) {
        table[name] = SymbolInfo{name, type, scopeLevel};
    }

    SymbolInfo getSymbol(std::string name) {
        return table.at(name);
    }
};
```

In this example, `SymbolTable` manages variable entries like `name`, `type`, and `scopeLevel`.

2.3 Three-Address Code (TAC)

Three-Address Code (TAC) is an intermediate representation used in the compiler pipeline. Each TAC instruction consists of an operator, two operands, and a result. This form of intermediate code is helpful for optimizations and code generation.

For example, the assignment expression:

```
a = b + 10;
```

might generate the following TAC instructions:

```
+ b, 10 => t1
= t1 => a
```

The `AssemblyGenerator` class translates these TAC instructions into assembly-like code with register allocation:

```
MOV R1, b; Load value of b into register R1
ADD R1, 10; Add 10 to the value in R1
MOV a, R1; Store the result back into variable a
```

In this case, intermediate TAC code is transformed into machine-level instructions using registers.

2.4 Lexer

The **Lexer** is responsible for tokenizing the input source code. It splits the source code into a sequence of tokens such as keywords, identifiers, literals, and operators while ignoring whitespace and comments. The lexer also handles syntax errors like unrecognized symbols or malformed literals.

For example, given the code:

```
int x = 10;
```

The lexer will produce the following tokens:

```
INT
IDENTIFIER
ASSIGN
INT
TERMINATOR
SEMICOLON
```

The lexer ensures that only meaningful elements of the code are passed to the parser. If an invalid symbol is encountered, it will trigger an error.

2.5 Parser

The **Parser** is responsible for analyzing the syntax of the source code. It takes the sequence of tokens produced by the lexer and checks whether they follow the rules of the language. If the code is syntactically correct, the parser generates Three-Address Code (TAC).

During parsing, the parser may need to interact with the symbol table to verify that variables have been properly declared and are in the correct scope. For example, when parsing a declaration like:

```
int a;
```

The parser checks if `a` has been previously declared. If `a` is not in the symbol table, the parser adds it, ensuring the symbol's type and scope are correctly assigned.

Example parser logic for a declaration:

```
void parseDeclaration()
    {
        string type = tokens[pos].value; // Store the type
        expect(tokens[pos].type);      // Expect type (int, float, char, string)
        string id = tokens[pos].value; // Store the identifier
        expect(T_ID);
        if (tokens[pos].type == T_ASSIGN)
        {
            expect(T_ASSIGN);
            string value = parseExpression();
            tacList.push_back(TAC("=", value, "", id));
            symbolTable.addSymbol(id, type, tokens[pos].line);
        }
        else
        {
            symbolTable.addSymbol(id, type, tokens[pos].line);
        }

        expect(T_SEMICOLON);
    }
}
```

For control structures like `if` or `while`, the parser first verifies the syntax of the condition and then ensures the statements inside the block are valid. For example, parsing:

```
if (x > 10) {
    y = 5;
}
```

The parser checks the expression `x > 10` and parses the assignment `y = 5`, making sure that both `x` and `y` are declared in the symbol table.

2.6 Assembly Generator

The `AssemblyGenerator` class is responsible for converting Three-Address Code (TAC) into assembly-like instructions. The main tasks it handles include register allocation, arithmetic operations, assignments, function calls, and output statements. It manages registers by mapping TAC variables to registers and uses a simple counter for allocating available registers.

Example

Consider the following TAC instructions:

- `x = 5`
- `y = 10`
- `z = x + y`

The `AssemblyGenerator` class processes these instructions as follows:

1. For the assignment `x = 5`, it allocates a register for `x` (e.g., `R0`) and generates the assembly instruction:

   `MOV R0, 5`

2. For `y = 10`, it allocates a register for `y` (e.g., `R1`) and generates the assembly instruction:

   `MOV R1, 10`

3. For `z = x + y`, it checks if both `x` and `y` are in registers. Since they are, it generates the instruction:

   `ADD R2, R0, R1` (for `z = x + y`)

   It then assigns the result to `R2` for `z`.

After processing these TAC instructions, the generated assembly code would look like this:

```
MOV R0, 5
MOV R1, 10
ADD R2, R0, R1
```

3 Testing and Results

3.1 Sample Input

```
int a;
String sa;
a = 5;
float f;
f = 5.4;
int b;
char cs;
cs = 'c';
b = a + 10;
bool bol;
bol = true;
bol = false;
string amd;
s = "this is amad";
print(a);
if (b <= 10) {
    return b;
} else {
    a = a + b;
}

while (a <= 20) {
    print(a);
    a = a + 1;
}

// hello
/* this is multiline comment */

for (int i; i <= 10; i = i + 1;) {
    print(i);
}

name(14, 12);

function name(string x, int y) {
    print(x);
    print(y);
    return true;
}
```

3.2 Sample Output

Three Address Code (TAC):

```


= 5,  => a
= 5.4,  => f
= c,  => cs
+ a, 10 => t3
= t3,  => b
= true,  => bol
= false,  => bol
= this is amad,  => s
print a,  =>
<= b, 10 => t9
return b,  =>
+ a, b => t11
= t11,  => a
<= a, 20 => t13
print a,  =>
+ a, 1 => t15
= t15,  => a
<= i, 10 => t17
+ i, 1 => t18
= t18,  => i
print i,  =>
arg 14,  => t21
arg 12,  => t22
call name,  => t23
param t21,  =>
param t22,  =>
print x,  =>
print y,  =>
return true,  =>
```

Symbol Table

```
------------------------------------------
Name      Type      Scope Level
------------------------------------------
x         string    36
y         int       36
```

```
i         int        28
amd       string     13
sa        String     2
f         float      4
b         int        6
a         int        1
cs        char       7
bol       bool       10
---------------------------------------------
```

Generated Assembly Code

```
MOV R0, 5
MOV R1, 5.4
MOV R3, 'c'
MOV R2, R3
MOV R5, 10
ADD R0, R5
MOV R4, R0
MOV R6, R4
MOV R7, true
MOV R7, false
MOV R9, "this is amad"
MOV R8, R9
MOV EAX, R10
CALL print
MOV EAX, R6
RET
ADD R0, R6
MOV R12, R0
MOV R0, R12
MOV EAX, R10
CALL print
MOV R15, 1
ADD R0, R15
MOV R14, R0
MOV R0, R14
MOV R18, i
ADD R18, R15
MOV R17, R18
MOV R18, R17
MOV EAX, R10
CALL print
PUSH EBP
MOV EBP, ESP
CALL t23
POP EBP
MOV ESP, EBP
MOV R22, _
MOV R23, _
MOV EAX, R10
CALL print
MOV EAX, R10
CALL print
MOV EAX, true
RET
```

4 Conclusion

n this project, a basic compiler framework has been successfully implemented, covering the essential compilation phases: lexical analysis, syntax validation, intermediate representation generation, and code generation. The modular architecture of the compiler allows for easy expansion and improvements.

This compiler framework lays the foundation for future enhancements, such as supporting more advanced language features, optimizing the code, and implementing better error handling. The design is modular, making it easy to incorporate additional phases or improvements in the future.