Compiler Construction

Submitted To:
Prof. Laeeq Khan Niazi

Submitted By:

Yasir Hassan 2021-CS-28

Department of Computer Science
University of Engineering and Technology, Lahore
Pakistan

Table of Contents

Introduction
The compiler is designed to translate a simplified programming language into x86 assembly language. This documentation evaluates the development process, explaining how functionality was incrementally added and highlighting advanced features.

Compiler Overview
The compiler consists of three main phases:
1. **Lexical Analysis (Tokenizer)**: Converts source code into tokens.
2. **Intermediate Code Generation**: Transforms tokens into Three-Address Code (TAC).
3. **Assembly Code Generation**: Converts TAC into x86 assembly language.

Phases of the Compiler:
Each phase plays a critical role in transforming high-level code into a low-level executable format:

Phase	Input	Output
Lexical Analysis	Source Code	Tokens
Intermediate Generation	Tokens	Three-Address Code (TAC)
Assembly Generation	Three-Address Code	x86 Assembly Code

Phase-by-Phase Development
1. Lexical Analysis (Tokenizer)
Purpose: To convert the source code into a sequence of tokens for further processing.
Implementation:
- Each token has a type (PLUS, IDENTIFIER, INTEGER_LITERAL, etc.), value (+, x, 42), and positional metadata (line, column).
- Handles multi-character operators like <=, ==.

Key Features:
- Supports keywords (int, if, while).
- Identifies arithmetic and logical operators.
- Recognizes identifiers and literals.

Example: Source Code:
- int x = 10;

Tokens:
- [INT] [IDENTIFIER: x] [ASSIGN: =] [INTEGER_LITERAL: 10] [SEMICOLON: ;]

2. Intermediate Code Generation
Purpose: Simplify the program into a series of Three-Address Code (TAC) instructions.
Implementation:
- TAC instructions consist of an operation, two operands, and a result.
- Handles assignments, arithmetic expressions, and conditional logic.

Key Features:
- Uses temporary variables for intermediate results (temp1, temp2).
- Supports loops and conditional branches.

Example: Source Code:
- x = a + b;

Three-Address Code:
- temp1 = a + b
- x = temp1

3. **Assembly Code Generation**

   Purpose: Translate TAC into x86 assembly language for execution.

   Implementation:
   - Converts operations like addition and multiplication into (mov), (add), (imul), etc.
   - Manages memory allocation and control flow using labels and jumps.

   Key Features:
   - Generates human-readable assembly with comments.
   - Supports logical conditions with (cmp) and (jne).

   Example: TAC:
   - temp1 = a + b
   - x = temp1

   Assembly Code:
   - mov rax, [a]
   - add rax, [b]
   - mov [temp1], rax
   - mov [x], rax

Additional Features

1. Temporary Variables
- Automatically creates and manages temporary variables for intermediate computations.
- Ensures clean and consistent output in TAC.

2. Error Handling
- Detects unknown tokens and syntax errors.
- Provides line and column information for debugging.

3. Logical and Conditional Operations
- Implements &&, ||, !, and relational operators (<, >=, ==).
- Handles complex logical expressions with precedence.

Step-by-Step Evaluation

1. Adding Identifier and Arithmetic Handling
- Introduced identifiers (x, y) and arithmetic operators (+, -, etc.).
- Example:
  - Input: x = 5 + 3;
  - Output: Tokens → TAC → Assembly Code.

2. Adding Relational and Logical Operators
- Added support for comparisons (<, >=) and logical expressions (&&, ||).
- Example:
  - Input: if (x > y)
  - Output: TAC with conditional jump labels.

3. Handling Keywords and Control Flow
- Introduced loops (`for`, `while`) and conditional branching (`if`, `else`).
- Example:
  - **Input:**
    - for (int i = 0; i < 10; i++) { x = x + i; }
  - **OutPut:**
    - TAC and corresponding labeled assembly code.

4. **Supporting String Literals and Advanced Data Types**
    - Added handling for `string` types and literals.
    - Example:
        - Input: string greeting = "Hello";
        - Output: Assembly with .data section.

5. **Introducing Functions and Complex Expressions**
    - Supported function calls, parameters, and return statements.
    - Example:
        - Input:
            - int calculate(int x, int y) { return x + y; }
        - Output: Prologue/epilogue assembly code.

Conclusions and Future Work

The compiler successfully transforms source code into assembly code through a multi-phase process. Each phase is modular, enabling flexibility and extensibility. Future enhancements include:

- **Optimization**: Minimize redundant TAC and assembly instructions.
- **Error Reporting**: Provide detailed feedback on semantic errors.
- **Advanced Constructs**: Add support for arrays, structs, and classes.

References

- Compiler Design Textbook
- Intel x86 Assembly Language Guide
- C++ Standard Documentation