Ghulam Ishaq Khan Institute of Engineering Sciences and Technology

CS424: COMPILER CONSTRUCTION

Group Members:

Wardah Tariq (2020519)
Warda Bibi (2020517)
Ehsan Illahi (2020111)

Contents

Compiler Project Report

1. Introduction

The primary objective of this project is to design, implement, and document a compiler for a C-like programming language. A compiler is a crucial tool in software development, translating human-readable code into machine-executable instructions. The report outlines the comprehensive process of creating the compiler, including the specification of the language, conversion to automata, compiler design, testing methodology, and documentation.

2. Language Specification

Grammar (BNF Notation)

The grammar serves as the foundation for the language's syntax and semantics. It delineates the allowable constructs and their permissible arrangements. The grammar provided encompasses statements, expressions, conditions, and operators, enabling the representation of common programming logic.

Following was the grammar used.

program -> stmt_list

stmt_list -> stmt stmt_list | stmt

stmt -> read (identifier)

if_stmt -> if (exp) { stmt_list }

exp -> exp + exp

The grammar utilized in our compiler project encapsulates the syntax of a C-like programming language, defining the structural rules and constructs essential for program representation and execution. With clear delineations for statements, expressions, and control flow structures, the grammar enables the creation of diverse and intricate programs. By incorporating fundamental elements such as assignment statements, conditional statements, loops, and expressions, the grammar fosters the development of robust and expressive software solutions. Its recursive nature facilitates the nesting of statements and expressions, allowing for the composition of complex program logic. Moreover, the inclusion of relational operators and conditions enriches the language with decision-making capabilities, further enhancing its versatility. Overall, the grammar serves as a solid foundation for building a compiler that can effectively process and translate C-like source code into executable instructions.

Features

The language specification defines the supported features, including basic programming constructs like assignments, conditionals, and loops. It allows for arithmetic expressions with support for integers and floating-point numbers. Additionally, it accommodates relational expressions for comparisons and permits nesting of control structures for enhanced expressiveness.

Limitations

While the language specification is robust, it does have limitations. Notably, it lacks support for advanced features such as arrays, functions, and structs. Error handling capabilities are also limited, which may necessitate further refinement in future iterations.

3. Conversion to Automata

Process and Optimization

The conversion of the language grammar to a finite automaton is a crucial step in the compilation process. This process involves mapping grammar rules to states and transitions within the automaton. To optimize the automaton, we employed techniques to minimize the number of states and transitions while preserving language semantics. This optimization ensures efficient parsing and compilation of source code.

4. Compiler Design

Lexical Analysis

The lexer serves as the initial stage of the compilation process, converting input streams of characters into tokens. Tokens represent the fundamental building blocks of the language, including keywords, identifiers, constants, operators, and punctuation symbols. The lexer ensures the syntactic correctness of the input and facilitates subsequent parsing.

Syntax Analysis

The parser constructs a parse tree or abstract syntax tree (AST) from the tokens generated by the lexer. We employed a recursive descent parsing technique based on the grammar rules to parse the input effectively. The parser verifies the structural correctness of the source code, identifying valid constructs and detecting syntax errors.

Semantic Analysis

Semantic analysis plays a crucial role in ensuring the correctness and meaningfulness of the source code. This phase performs type checking to ensure that operations involving variables and expressions are semantically valid. Additionally, semantic analysis checks for semantic correctness, such as the use of undeclared variables or incompatible operations.

Intermediate Code Generation

The intermediate code generation phase translates the validated syntax tree into an intermediate representation (IR). We adopted the three-address code representation to capture high-level operations

in a platform-independent format. This intermediate representation serves as an intermediary step before generating target code.

Code Optimization

Code optimization aims to improve the efficiency and performance of the generated code. We applied various optimization techniques, including constant folding and common subexpression elimination, to optimize the intermediate code. These optimizations reduce redundant computations and enhance the runtime efficiency of the compiled code.

Code Generation

The code generation phase translates the optimized intermediate code into target assembly or machine-like code. We targeted a simple virtual machine architecture, generating code that can be executed on the specified platform. The generated code is executable and represents the compiled form of the input source code.

5. Testing and Documentation

Testing Methodology

Comprehensive testing is essential to validate the correctness and robustness of the compiler. We conducted extensive testing using diverse code snippets covering all language features. This testing approach encompassed both valid and invalid inputs to verify correct behavior and error handling. Through rigorous testing, we ensured the reliability and correctness of the compiler across various scenarios.

Documentation

Thorough documentation is indispensable for understanding the design decisions, challenges, and solutions implemented throughout the compiler development process. The project report provides detailed documentation of each phase, including design choices, optimization strategies, and testing methodologies. Additionally, the report includes examples of input-output and testing results to illustrate the compiler's functionality effectively.

6. Conclusion

In conclusion, the development of the compiler for the C-like programming language involved meticulous planning, design, and implementation. By adhering to the language specification, optimizing the automaton conversion process, and implementing robust compiler phases, we successfully created a functional and efficient compiler. Thorough testing and comprehensive documentation ensure the reliability, correctness, and usability of the compiler for future software development endeavors.