CS424 – Compiler Construction
Compiler for a C-like Programming Language

Bilal Malik (2020102)
Abdullah Farooq (2020023)
Ammar Ahmad (2020071)

Abstract

This report details the design and implementation of a compiler for a simplified C-like programming language. The project encompasses defining the language's grammar using Context-Free Grammar (CFG) and Backus-Naur Form (BNF) notation, converting the grammar into a finite automaton using JFLAP, and implementing the various phases of the compiler, including lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, and final code generation. Each phase is meticulously designed to ensure the compiler efficiently translates high-level language constructs into low-level assembly or machine-like code. Challenges encountered during the development process, such as handling ambiguities in grammar and optimizing the intermediate code, are discussed in detail. The compiler's functionality and correctness are validated through comprehensive testing with diverse code snippets. This report provides an in-depth overview of the design decisions, implementation details, and testing methodologies, offering insights into the complexities and solutions involved in

Language Specification

The grammar of the simplified C-like programming language is defined using Context-Free Grammar (CFG) notation. The notation specifies the structure and syntax rules for valid programs written in this language.

- Program Structure
  - `program -> stmt_list`
- Statement List
  - `stmt_list -> stmt stmt_list | ε`
- Statements
  - `stmt -> assignment_stmt | if_stmt | print_stmt`
- Assignment Statement
  - `assignment_stmt -> VARIABLE = expr ;`
- If Statement
  - `if_stmt -> if ( expr ) { stmt_list } else { stmt_list } | if ( expr ) { stmt_list }`
- Print Statement
  - `print_stmt -> print ( expr ) ;`
- Expressions
  - `expr -> term expr'`
  - `expr' -> + term expr' | - term expr' | ε`
- Terms
  - `term -> factor term'`
  - `term' -> * factor term' | ε`
- Factors
  - `factor -> NUMBER | VARIABLE | ( expr )`
- Number
  - `NUMBER -> [0-9]+`
- Variable
  - `VARIABLE -> [a-zA-Z]+`

1. Supported Features:
    a. Variable Assignment: Allows assignment of numeric expressions to variables.
    b. Conditional Statements: Supports basic `if-else` constructs for conditional execution of statements.
    c. Printing: Enables printing of expressions to standard output.
    d. Arithmetic Operations: Supports addition, subtraction, and multiplication within expressions.
    e. Nested Statements: Allows nesting of statements within conditional blocks.

2. Limitations:
    a. Data Types: Only supports numeric variables (integers).
    b. Operators: Limited to basic arithmetic operators (+, -, *).
    c. Control Flow: Only basic `if-else` statements are supported; loops and other control structures are not included.
    d. Error Handling: Limited error handling capabilities, focusing primarily on syntactic correctness.
    e. Functions: No support for user-defined functions or procedures.

This grammar and feature set provides a foundational understanding of the simplified C-like language, setting the stage for the development of the compiler's various components.

Compiler Architecture

The compiler architecture comprises several interconnected components responsible for transforming source code written in the simplified C-like language into executable machine code. Each component performs specific tasks in a sequential manner, ultimately generating optimized code ready for execution.

1. Lexer (Tokenization):
   - The lexer scans the input source code character by character and converts it into a sequence of tokens.
   - Tokens represent the basic syntactic elements of the language, such as keywords, identifiers, numbers, operators, and special characters.
   - Regular expressions are typically employed to recognize and categorize different types of tokens efficiently.

2. Parser (Syntax Analysis):
   - The parser processes the token stream generated by the lexer and constructs a parse tree or abstract syntax tree (AST) based on the grammar rules of the language.
   - It verifies the syntactic correctness of the code and identifies the hierarchical structure of statements and expressions.
   - Recursive descent parsing or other parsing techniques are commonly used to navigate the grammar rules and build the parse tree.

3. Semantic Analyzer:
   - The semantic analyzer performs type checking and ensures the semantic correctness of the code.
   - It enforces language-specific rules and constraints, such as verifying variable declarations, checking compatibility of data types, and resolving identifiers.
   - Semantic analysis may involve symbol table management to track variable declarations and resolve references.

4. Intermediate Code Generator:
   - The intermediate code generator translates the AST produced by the parser into an intermediate representation (IR) of the program.
   - IR is a platform-independent and language-agnostic representation that captures the essential semantics of the source code.
   - Common intermediate representations include three-address code, static single assignment (SSA) form, and abstract stack machine code.

5. Code Optimizer:
    - The code optimizer analyzes the intermediate code and applies various optimization techniques to improve the efficiency and performance of the generated code.
    - Optimization strategies may include constant folding, dead code elimination, loop optimization, and register allocation.
    - The goal is to minimize resource usage, reduce execution time, and enhance overall program quality.
6. Code Generator:
    - The code generator translates the optimized intermediate code into target-specific assembly language or machine code.
    - It maps high-level language constructs to low-level instructions supported by the target hardware architecture.
    - Instruction selection, scheduling, and register allocation are key tasks performed by the code generator.

Implementation Details

1. Lexer Implementation:
   - Utilizes regular expressions to tokenize the input source code.
   - Defines token types and corresponding patterns for recognition.
   - Generates a token stream as output, ready for parsing.

2. Parser Implementation:
   - Implements parsing methods for each grammar rule defined in the language specification.
   - Uses recursive descent parsing or other parsing techniques to construct the parse tree.
   - Handles syntax errors gracefully by raising exceptions or providing detailed error messages.

3. Semantic Analyzer Implementation:
   - Implements type checking and semantic rules enforcement.
   - Manages symbol tables to store variable declarations and resolve identifiers.
   - Detects semantic errors such as type mismatches, undeclared variables, and incompatible operations.

4. Intermediate Code Generator Implementation:
   - Constructs an intermediate representation (IR) of the program based on the parse tree or AST.
   - Generates IR instructions corresponding to high-level language constructs.
   - Ensures the IR captures the essential semantics of the source code.

5. Code Optimizer Implementation:
   - Applies optimization passes to the intermediate code to improve performance.
   - Implements optimization algorithms and transformations tailored to specific IR representations.
   - Balances optimization effectiveness with compilation time constraints.

6. Code Generator Implementation:
   - Translates the optimized intermediate code into target assembly language or machine code.
   - Maps IR instructions to corresponding target instructions, considering the target architecture's instruction set and memory model.
   - Handles instruction selection, scheduling, and register allocation to produce efficient and correct machine code output.

Testing and Validation

Our testing strategy aims to ensure the correctness and reliability of the compiler by systematically evaluating its behaviour against a diverse set of input expressions. Test cases were selected to cover a wide range of language features and edge cases, including valid syntax constructs and common error scenarios. The criteria for evaluating the compiler's performance include its ability to accurately parse valid syntax, detect and report syntax errors, and handle invalid input gracefully.

1. Test Cases:
    a. a = 5;
    b. x = 5; print(x / 5);
    c. print(4 + 7 + 5);
    d. if (x > 2) print(x);
    e. a \ 5 (Invalid syntax)
    f. 3 * [b - 2] (Invalid syntax)
    g. x / {y + 7} (Invalid syntax)
    h. a = 5 (Invalid syntax)
    i. if (x > 2) print(x); else print(z); (Valid Syntax but not parsing accurately)

```
# Test the parser with various input expressions
test_parser("a = 5;")
test_parser("x = 5; print(x / 5);")
test_parser("print(4 + 7 + 5);")
test_parser("if (x > 2) print(x);")
```

```
test_parser("a \ 5") # Invalid syntax
test_parser("3 * [ b - 2]") # Invalid syntax
test_parser("x / {y + 7}")  # Invalid syntax
test_parser("a = 5") # Invalid syntax
```

```
# Valid Syntax but not parsing accurately
test_parser("if (x > 2) print(x); else print(z); ")
```

2. Results:
    a. a = 5;
        i. Parsed Result: [('=', 'a', 5)]
    b. x = 5; print(x / 5);
        i. Parsed Result: [('=', 'x', 5), ('print', ('/', 'x', 5))]
    c. print(4 + 7 + 5);
        i. Parsed Result: [('print', ('+', ('+', 4, 7), 5))]
    d. if (x > 2) print(x);
        i. Parsed Result: [('if', 'x', '>', 2, [('print', 'x')], [])]
    e. a \ 5 (Invalid syntax)
        i. Error: Syntax Error
    f. 3 * [b - 2] (Invalid syntax)
        i. Error: Syntax Error
    g. x / {y + 7} (Invalid syntax)
        i. Error: Syntax Error
    h. a = 5 (Invalid syntax)
        i. Error: Syntax Error
    i. if (x > 2) print(x); else print(z); (Valid Syntax but not parsing accurately)
        i. Error: Syntax Error

```
Expression: a = 5;
Parsed Result: [('=', 'a', 5)]

Expression: x = 5; print(x / 5);
Parsed Result: [('=', 'x', 5), ('print', ('/', 'x', 5))]

Expression: print(4 + 7 + 5);
Parsed Result: [('print', ('+', ('+', 4, 7), 5))]

Expression: if (x > 2) print(x);
Parsed Result: [('if', 'x', '>', 2, [('print', 'x')], [])]

Expression: a \ 5
Error: Syntax Error

Expression: 3 * [ b - 2]
Error: Syntax Error

Expression: x / {y + 7}
Error: Syntax Error

Expression: a = 5
Error: Syntax Error

Expression: if (x > 2) print(x); else print(z);
Error: Syntax Error
```

Design

Throughout the development of the compiler, several key design decisions were made to achieve the desired functionality and performance. One crucial decision was the choice of a recursive descent parsing approach for syntax analysis, as it offers simplicity and ease of implementation for parsing context-free grammars. Another important decision was the selection of an intermediate representation (IR) based on abstract syntax trees (ASTs) for representing program semantics during code generation. This choice enables efficient traversal and manipulation of program structures, facilitating optimization and code generation processes.

The development of the compiler posed several challenges, particularly in handling complex language constructs and ensuring robust error handling. One significant challenge was devising an effective strategy for parsing expressions and resolving operator precedence and associativity. Another challenge involved implementing semantic analysis to enforce type checking and ensure the correctness of program semantics. Additionally, error handling mechanisms required careful attention to provide informative error messages while maintaining parser efficiency.

Conclusion

The development of the compiler has been a challenging yet rewarding journey, culminating in a functional and efficient tool for translating high-level code into executable machine instructions. By documenting design decisions, challenges faced, and examples of input-output behavior, this documentation provides valuable insights into the compiler's development process and outcomes. Moving forward, continuous refinement and improvement will be undertaken to enhance the compiler's capabilities and ensure its suitability for a wide range of programming tasks.