Compiler Construction

Session: 2021 – 2025

Submitted by:

Khadim Hussain 2021-CS-204

Supervised by:

Sir Laeeq Khan Niazi

Department of Computer Science

University of Engineering and Technology

Lahore Pakistan

Table of Content

Multi-Phase Compiler Implementation

Introduction:

This document provides a detailed explanation of my custom compiler implementation, designed to transform high-level source code into assembly-like intermediate representation through multiple compilation phases.

Phases of Compiler:

My compiler is main comprises of 4 phases:

1. Lexical Analysis (Lexer)
2. Parsing and Semantic Analysis (Parser)
3. Intermediate Code Generation
4. Assembly Code Generation

Phase 1: Lexical Analysis (Lexer):

Lexical analyser is the first phase of the compiler that takes source code in string as input and makes it token by using its function tokenize(). The Lexer serves as the first stage of compilation, transforming raw source code into a structured sequence of tokens. Its primary objective is to break down the input text into meaningful, atomic units that can be further processed by subsequent compiler stages.

Token Type:

The TokenType enumeration defines all possible tokens recognized by the compiler.

Utility Function for TokenType:

The tokenTypetoString utility function maps TokenType values to their string representations. This enhances the readability and debugging of compiler output.

Key Features:

Some key features of lexer are:

- Support for a single primitive data type (initially integer)
- Recognize basic tokens: numbers, identifiers, arithmetic operators
- Handle single and multi line comments
- Basic error detection for unexpected characters

Phase 2: Syntax and Semantic Analysis (Parser):

Parser is the second phase of the compiler that takes a token generated by first phase of the compiler(lexer) and parses it to check its syntax and semantics. The Parser validates the syntactic structure of the token

sequence, ensuring that the code adheres to the language's grammatical rules. It builds an intermediate representation of the program while performing semantic checks.

Key Features:

Some key features of lexer are:

- Verify syntactic correctness of the token sequence
- Enforce language grammar rules
- Manage symbol table for variable tracking
- Perform type checking
- Detect and report syntactic and semantic errors

Phase 3: Intermediate Code Generation

If no errors occurred the compiler then enters into the third phase in which the expressions in the code are converted into Three Address Code (TAC) format. This phase translates the parsed code into a lower-level, machine-independent intermediate representation called Three-Address Code (TAC). It simplifies complex expressions and prepares the code for final code generation.

Key Features:

Some key features of lexer are:

- Generate three-address code
- Generate temporary variables
- Create dynamic labels for control flow
- Prepare code for subsequent translation

Phase 4: Assembly Code Generation:

The final phase translates the intermediate code into a low-level representation resembling assembly language. It maps abstract instructions to concrete machine-like instructions, preparing the code for potential machine code generation.

Key Features:

Some key features of lexer are:

- Convert intermediate instructions to pseudo-assembly
- Translate control flow and arithmetic operations
- Register allocation

My Contributions:

I added following features:

- Single and multi-line comments
- Float datatype
- Initialization and assignment
- For and while loops

Conclusion:

The compiler development approach focuses on systematic, incremental feature addition. By starting with a simple, functional implementation and progressively enhancing capabilities, developers can create a robust and extensible compiler infrastructure.