**Mini C Compiler**

¹**Aashish Gupta[2261051], ²Arpita Rajput[2261116],3Garima Aithani[2261206],4Garima Bisht[2261207]**

¹Graphic Era Hill University, Sat Tal Road, PO: Bhowali, Bhimtal, Nainital - 263136, Uttarakhand, India

2Graphic Era Hill University, Sat Tal Road, PO: Bhowali, Bhimtal, Nainital - 263136, Uttarakhand, India

3Graphic Era Hill University, Sat Tal Road, PO: Bhowali, Bhimtal, Nainital - 263136, Uttarakhand, India

4Graphic Era Hill University, Sat Tal Road, PO: Bhowali, Bhimtal, Nainital - 263136, Uttarakhand,India

Email: [ARPITARAJPUT.220122662@gehu.ac.in](mailto:ARPITARAJPUT.220122662@gehu.ac.in)

**Abstract** – Research in compiler construction has been one of the core research areas in computing. Researchers in this domain try to understand how a computer system and computer languages associate. A compiler translates code written in human-readable form (source code) to target code (machine code) that is efficient and optimized in terms of time and space without altering the meaning of the program. This paper aims to explain what a compiler is and give an overview of the stages involved in translating computer programming languages.

**Key Words:** compiler, phases of a compiler, analysis, synthesis, features of a compiler

**1. INTRODUCTION**

Assembly or high-level languages are the languages used to write a computer system program. However, a computer system understands none of these languages. Therefore, a compiler is needed to translate the high-level language. A high-level language is a language written in a human-readable form with an easy-to-read syntax. Examples of such languages are Java, C#, Delphi, PL/I, and Python. A non-computer program written in a high-level language is known as source code. A compiler takes source code as input, processes it, and produces an object code without changing the meaning of the source code. The object code is sometimes called machine code or target code.

A compiler is a computer system software that transfigures source code into an intermediate code which afterwards is transformed into target code without affecting the meaning of the source code. The result of the transformation (machine code) must be efficient and optimized in terms of time and space (memory size). The interface between a computer program and a computer system is the standard (the operating system). A compiler detects error(s) in the source code during compilation processes and time. There are three types of errors in computer programming. They are syntax, runtime, and logical errors. The only detected error during compilation processes is the syntax error. The other two types of errors occur during program execution.

The back-end and the front-end are the two parts of a compiler. The task of the back-end is to synthesize the target language. Then, the front-end analyses the source code. In a perfect compiler design, the back-end will lack any knowledge of the source code, and the front-end will also lack knowledge about the target code. A compiler operates in stages. Each stage performs a specific task. These stages are a scanner, parser, semantic analysis, intermediate code, code optimization, and code generator.

**1.1 Features of a Compiler**

a. Correctness  
b. Speed of compilation  
c. Preserve the correct meaning of the code  
d. Compile-time proportion to program size  
e. Good diagnostics for syntax errors  
f. Good error reporting and handling  
g. Work well with debugging

**1.2 Types of Compiler**

A compiler is divided into three, namely:

* Single-pass compiler
* Two-pass compiler
* Multi-pass compiler

**2. THE COMPONENTS OF A COMPILER**

Before a compiler translates source code to object code, the source code undergoes a series of steps, and these steps are called phases of a compiler. Each stage performs a single task only. A data structure called a symbol table is needed to store the output of each stage, and an error handler needs to be present to keep track of errors encountered.

The phases of a compiler consist of six phases. These phases can be grouped into two categories as follows:

**2.1 Analysis**

The source code is divided into meaning characters and creates an intermediate representation. This part is further subdivided into three as follows:

a. Lexical analysis  
b. Syntax analysis  
c. Semantic analysis

**2.2 Synthesis**

The output of the analysis is used here to produce the desired machine-oriented code. This section is subdivided into three:

a. Intermediate code generation  
b. Code optimization  
c. Code generator

**LEXICAL ANALYSIS**

Lexical analysis is the first stage in compiler construction. This stage is also called scanning. In this stage, the source code is scanned to remove any whitespace or comments. Then, it groups the source code into categories into meaningful sequences of lexical items called tokens.

A token may be composed of a single character or a sequence of characters. A token is classified as being either: Identifiers, Keywords, Operators, Separators, Literals, and Comments. For each lexeme, the scanner produces a token as output in the form: <Token-name, attribute-value>

**Example: Lexical Analysis Table**

|  |  |
| --- | --- |
| **Lexeme (Collection of Characters)** | **Tokens (Category of Lexeme)** |
| Variable | Identifier (id) |
| = | Assignment Operator |
| Value | Identifier |
| + | Addition Operator |
| 10 | Integer Constant |

**SYNTAX ANALYSIS**

This phase involves parsing the tokenized input to validate the program's grammatical structure based on C language rules. Parsers such as LL(1) and LR are commonly used for syntax analysis.

**Example:**

***Input:* int x = 10 + y;**

***Output:* Syntax tree**

**=**

**/ \**

**x +**

**/ \**

**10 y**

**SEMANTIC ANALYSIS**

Semantic analysis ensures logical correctness, such as type checking, scope resolution, and function verification. It detects errors that are not caught by syntax analysis.

**Example:**

***Input:* int x = "hello";**

***Output:* *Error: Type mismatch (expected \*\*, found \*\*).***

**INTERMEDIATE CODE GENERATION**

The compiler converts the parsed code into an intermediate representation (IR), which is a lower-level abstraction that makes further optimizations easier. Examples of IR include three-address code (TAC) and static single assignment (SSA) form.

**Example:**

***Input:* x = a + b \* c;**

***Output (Three-Address Code):***

**t1 = b \* c**

**t2 = a + t1**

**x = t2**

**CODE OPTIMIZATION**

Code optimization improves execution speed and reduces memory usage. Techniques like constant folding, loop unrolling, and dead code elimination are employed to enhance performance.

**Example:**

***Before Optimization:***

**x = 5 \* 2;**

**y = x + 0;**

***After Optimization:***

**x = 10;**

**FINAL CODE GENERATION**

The final phase involves translating the optimized intermediate representation into machine code. This includes register allocation, instruction selection, and assembly code generation.

**Example:**

***Input:* x = 10 + y;**

***Output (Assembly Code - x86):***

**MOV EAX, 10**

**ADD EAX, [y]**

**MOV [x], EAX**

**3. CONCLUSION**

This document provides an overview of compiler construction, covering the essential phases involved in translating source code into executable machine code. Each phase plays a crucial role in ensuring that the final output is correct, efficient, and optimized for execution.

**4.REFERENCES**

**1. Compiler design tutorial**

<https://www.geeksforgeeks.org/compiler-design-tutorials/>

**2.Phases of compiler**

<https://www.geeksforgeeks.org/phases-of-a-compiler/>

**3.An ultimate guide to Git and Github**

<https://www.geeksforgeeks.org/ultimate-guide-git-github/>