Introduction to Compiler Design

Table of Contents

[Generations of Programming Languages 2](#_Toc141040773)

[Bootstrapping 3](#_Toc141040774)

[Language Processing System 4](#_Toc141040775)

[Structure of a Compiler 5](#_Toc141040776)

[Compiler Construction Tools 6](#_Toc141040777)

The purpose of a **compiler** is to convert a high-level language, such as Python or C++, into a low-level language such as assembly language. This allows us to work with languages that are easier to read and write.

Compilers are a form of **translator**, which translate text from one language to another. A simple translator that only converts text sentence by sentence without considering the relations and dependencies between them is called an **interpreter**. It the translator also checks the dependencies for correctness, it becomes a compiler.

A **hybrid compiler** combines the two types of translators but first generating **intermediate code** using a compiler which is then passed through an **interpreter** at execution time.

## Generations of Programming Languages

Over time, programming languages have gotten easier to use. They can be divided into five generations:

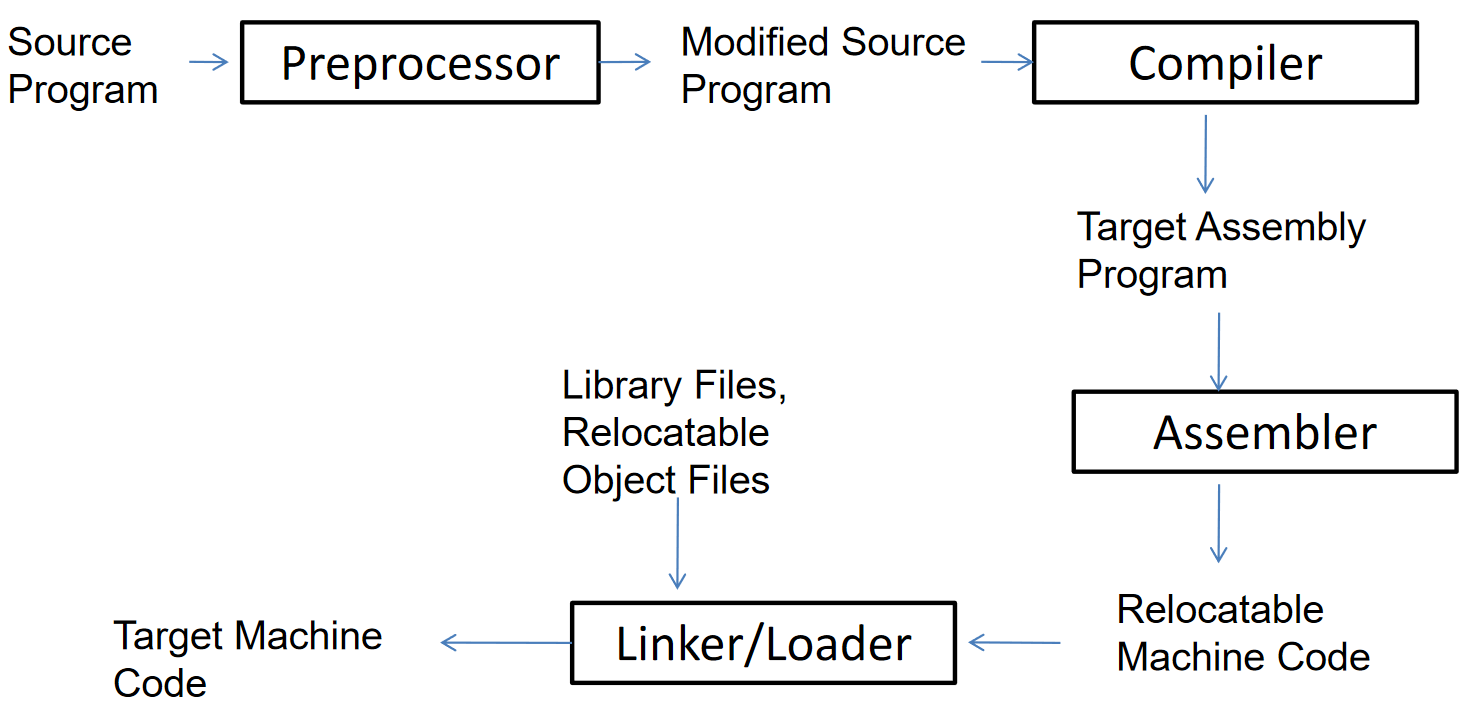
1. **First Generation** – These are Machine Languages, which are low-level languages that deal with binary numbers. These languages can be directly executed by the computer quickly and efficiently. Memory utilization is also efficient since every bit of data can be tracked.
2. **Second Generation** – These are Assembly Languages, which are also low-level languages. However, these deal with mnemonics, which are symbolic names that represent the operation code and operands for instructions. Assembly languages are easier to work with and are less error prone.
3. **Third Generation** – The third generation is where high-level languages began to appear, such as C or C++. High-level languages allow the programmer to think about the logic of the program without having to think about the internal architecture of the computer system. This is done by making the language easy to develop with and understand, which in turn makes them less error prone and require less time to write with.
4. **Fourth Generation** – The fourth generation of languages are considered very-high-level programming languages since they further reduce the time, cost and effort needed to develop a program. The implement databases to efficiently use data, and are highly portable. Notable examples include Python and SQL.
5. **Fifth Generation** – These are Artificial Intelligence languages, since they are mainly used in artificial intelligence. The program is created based on user input instead of algorithms written by a programmer. These languages can query their databases quickly and efficiently, and allow the user to communicate with the computer in a simple and easy manner. 5th generation languages are not widely used yet. Examples include Prolog and Mercury.

## Bootstrapping

**Bootstrapping** is magical process by which a program can be developed using the program itself.

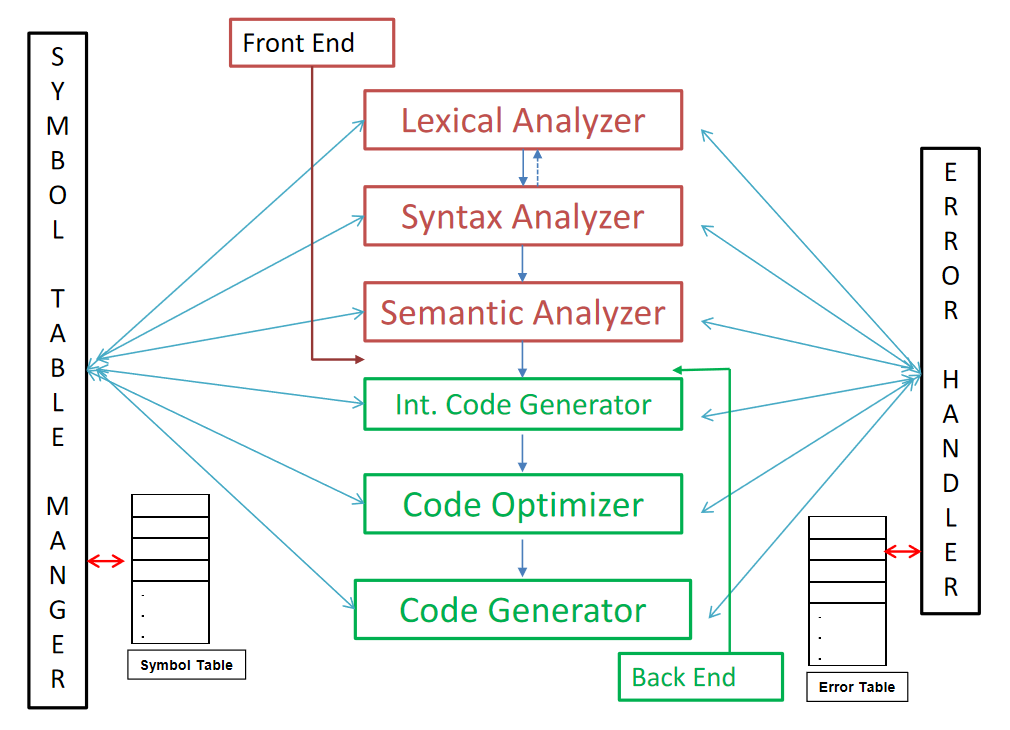
## Language Processing System

For high-level languages to be converted to machine code, a series of steps, called the **language processing system**, is required.



The source code is converted to a pure form (replacing include statements with the corresponding files and macros with their values) by a **preprocessor**, which is turned into assembly language by the **compiler**, which is turned into machine code by an **assembler** to which library files are added using a linker. The only part in this entire process we are interested in is the details of the compiler and how it translates pure source code to assembly code.

## Structure of a Compiler



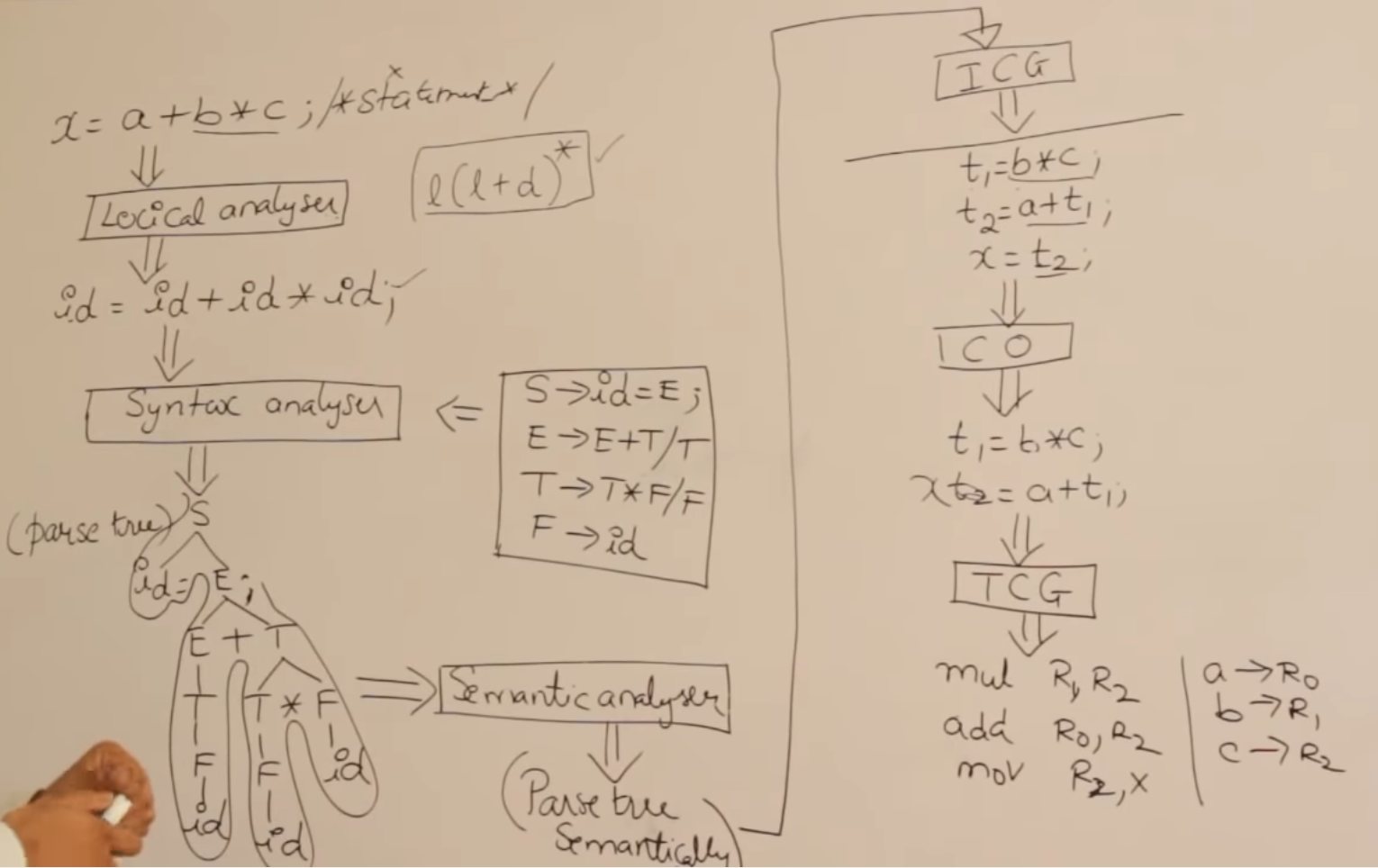
A compiler has mainly 6 parts. The first three parts are the **front end** which deal mostly with the source code itself. The last three parts are the **back end** which deal with assembly code.

For a language, each individual smallest meaning item is called a **token**. For example, the words in a sentence can be considered tokens. The **lexical analyzer** creates a stream of tokens from a stream of characters. The **syntax analyzer** checks if the order of the tokens is correct based on some predefined rules. It creates a **parse tree** from the tokens using context-free grammar, which is passed to the **semantic analyzer**. This verifies that the parse tree is meaningful.

The **intermediate code generator** generates **three-address codes** from the parse tree. This is then optimized by the **code optimizer** and finally converted to **assembly code** by the **code generator**.

In addition to these we also have the **Symbol Table Manager** and the **Error Handler**, which supports all of the 6 sections.

An example of the entire process is provided below:



## Compiler Construction Tools

There are various tools available that can help us create each of the parts of the compiler:

1. **Scanner Generators** like Lex, which can help create lexical analyzers.
2. **Parser Generators** like Yacc, which can help create syntax analyzers.
3. **Syntax-Generated Translation Engines** which generate collections of procedures that walk through the parse tree.
4. **Automatic Code Generators**
5. **Data Flow Engines**