CHAP 1 :

Introduction to Compiler

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When you execute a program which is written in HLL programming language then it executes into two parts.
- In the first part, the source program compiled and translated into the object program (low level language).
- In the second part, object program translated into the target program through the assembler.

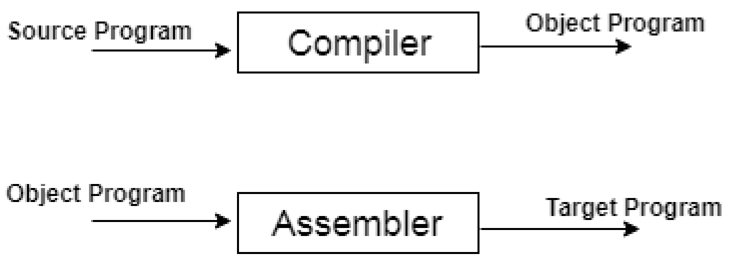
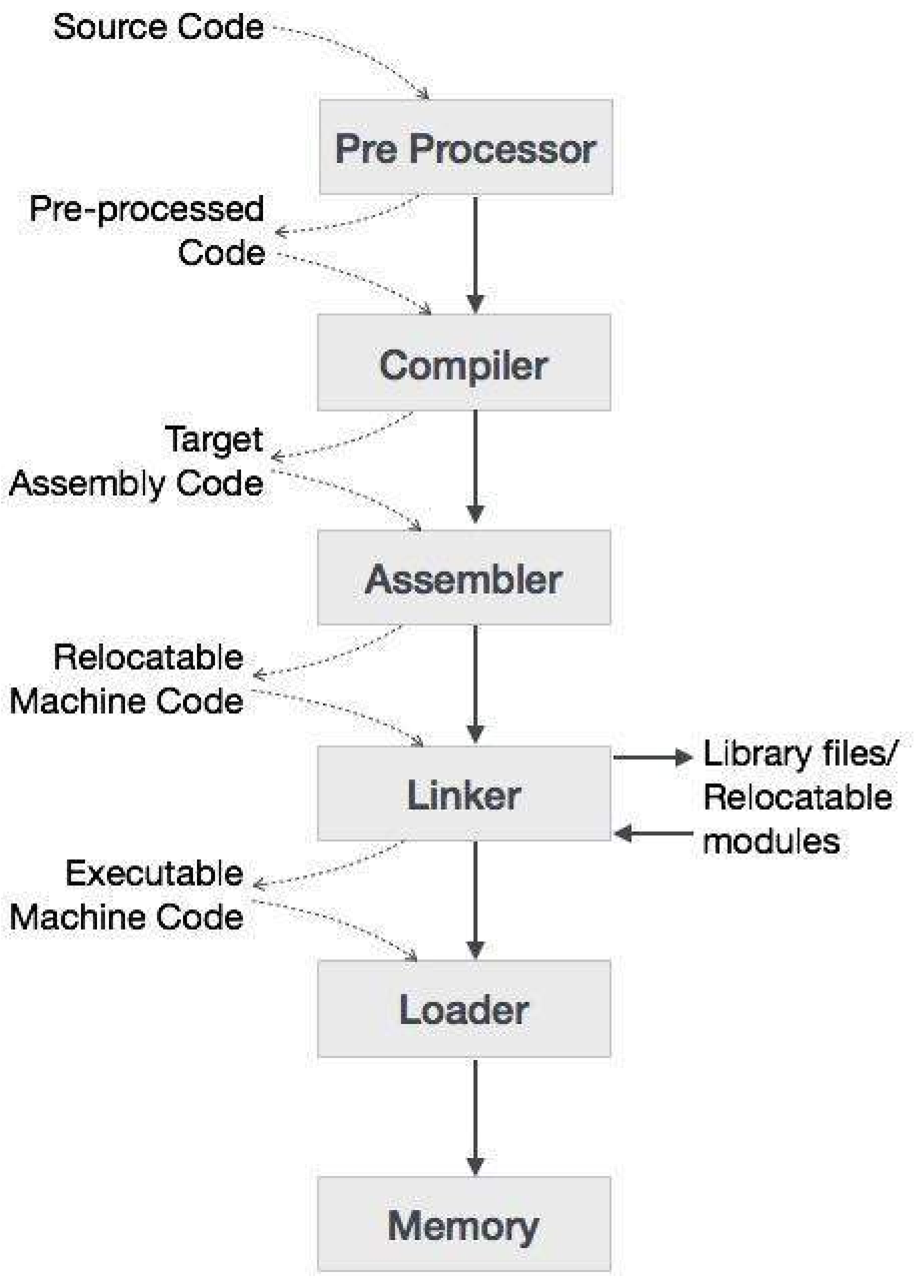


Fig: Execution process of source program in Compiler

Language Processing System

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.



The high-level language is converted into binary language in various phases. A **compiler** is a program that converts high-level language to assembly language. Similarly, an **assembler** is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using C compiler, is executed on a host machine.

- User writes a program in C language (high-level language).
- The C compiler, compiles the program and translates it to assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).
- A linker tool is used to link all the parts of the program together for execution (executable machine code).
- A loader loads all of them into memory and then the program is executed.

Before diving straight into the concepts of compilers, we should understand a few other tools that work closely with compilers.

Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

Assembler

An assembler translates assembly language programs into machine code.The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

Linker

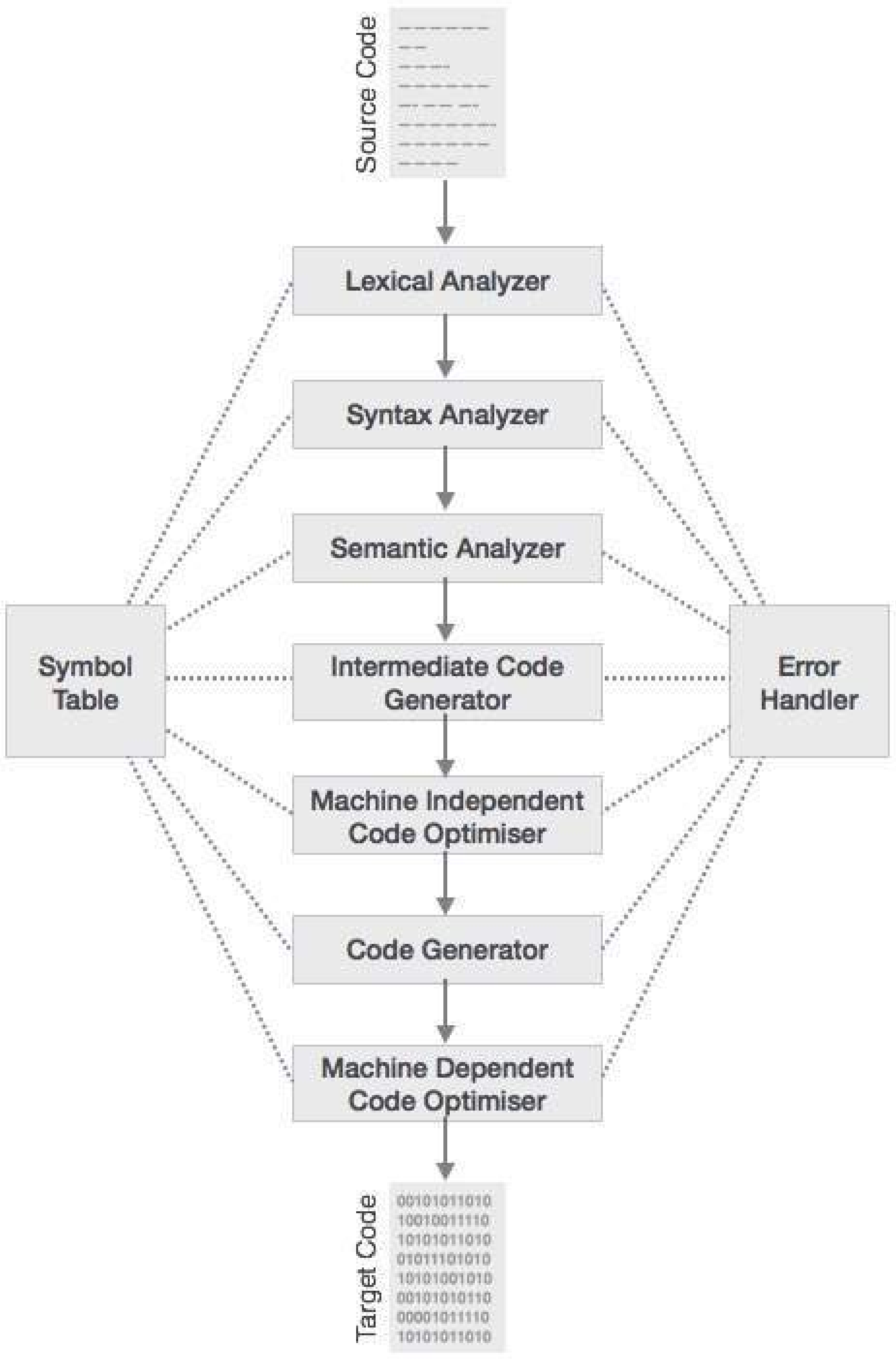
Linker is a computer program that links and merges various object files together in order to make an executable file. All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.

Loader

Loader is a part of operating system and is responsible for loading executable files into memory and execute them. It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.

Phases of Compiler

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

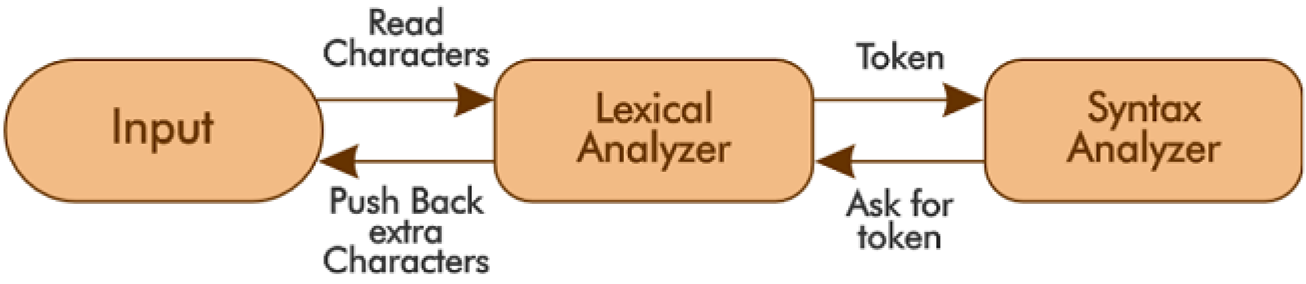


Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

A lexical analyzer is also called a "**Scanner**". Given the code's statement/ input string, it reads the statement from left to right character-wise. The input to a lexical analyzer is the pure high-level code from the preprocessor. It identifies valid lexemes from the program and returns tokens to the syntax analyzer, one after the other, corresponding to the ***getNextToken*** command from the syntax analyzer.



There are three important terms to grab:

1. **Tokens**: A Token is a pre-defined sequence of characters that cannot be broken down further. It is like an abstract symbol that represents a unit. A token can have an optional attribute value. There are different types of tokens:
    - Identifiers (user-defined)
    - Delimiters/ punctuations (;, ,, {}, etc.)
    - Operators (+, -, *, /, etc.)
    - Special symbols
    - Keywords
    - Numbers
2. **Lexemes**: A lexeme is a sequence of characters matched in the source program that matches the pattern of a token.
   For example: (,) are lexemes of type punctuation where punctuation is the token.
3. **Patterns:** A pattern is a set of rules a scanner follows to match a lexeme in the input program to identify a valid token. It is like the lexical analyzer's description of a token to validate a lexeme.
   For example, the characters in the keyword are the pattern to identify a keyword. To identify an identifier the pre-defined set of rules to create an identifier is the pattern

Token	Lexeme	Pattern
Keyword	while	w-h-i-l-e
Relop	<	<, >, >=, <=, !=, ==

Integer	7	(0 - 9)*-> Sequence of digits with at least one digit
String	"Hi"	Characters enclosed by " "
Punctuation	,	; , . ! etc.
Identifier	number	A - Z, a - z A sequence of characters and numbers initiated by a characte

Everything that a lexical analyzer has to do:

1. Stripping out comments and white spaces from the program
2. Read the input program and divide it into valid tokens
3. Find lexical errors
4. Return the Sequence of valid tokens to the syntax analyzer
5. When it finds an identifier, it has to make an entry into the symbol table.

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.