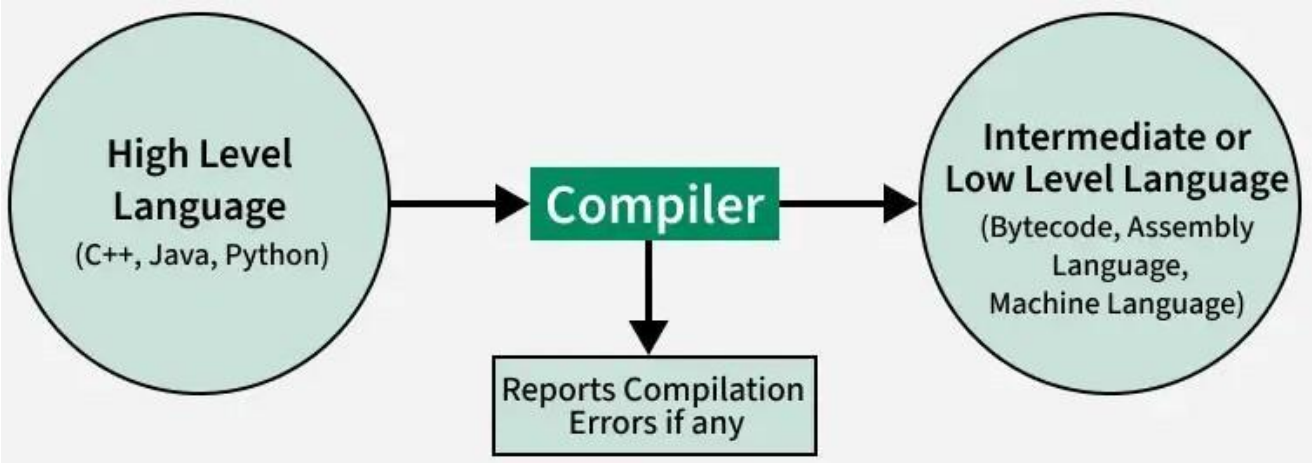
UNIT- I Introduction to Compiling

A compiler is software that translates or converts a program written in a high-level language (Source Language) into a low-level language (Machine Language or Assembly Language). Compiler design is the process of developing a compiler.



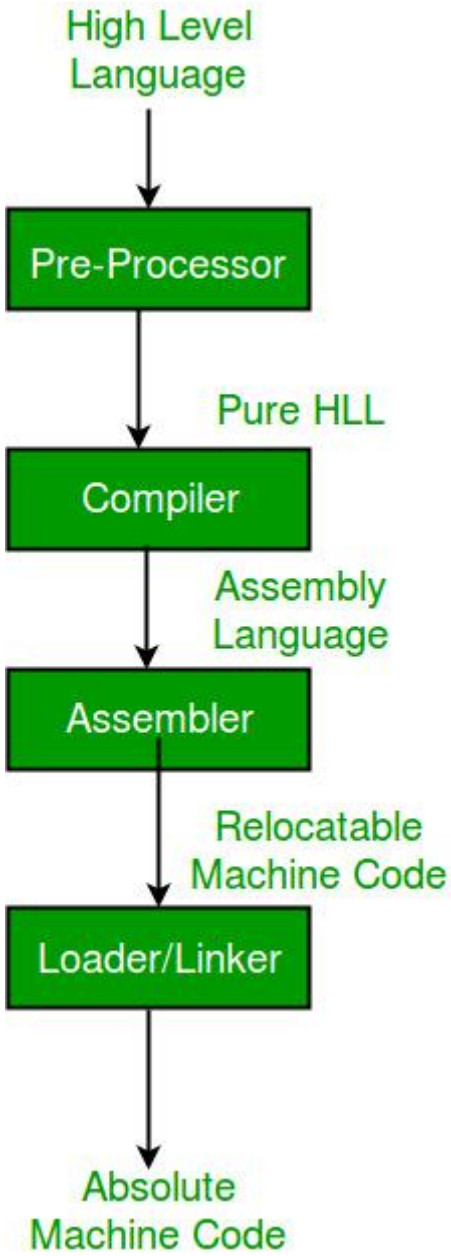
Role of Compilers

A program written in a high-level language cannot run without compilation. Each programming language has its own compiler, but the fundamental tasks performed by all compilers remain the same. Translating source code into machine code involves multiple stages, such as lexical analysis, syntax analysis, semantic analysis, code generation, and optimization.

While compilers are specialized, they differ from general translators. A translator or language processor is a tool that converts an input program written in one programming language into an equivalent program in another language.

Language Processing Systems

We know a computer is a logical assembly of Software and Hardware. The hardware knows a language, that is hard for us to grasp, consequently, we tend to write programs in a high-level language, that is much less complicated for us to comprehend and maintain in our thoughts. Now, these programs go through a series of transformations so that they can readily be used by machines. This is where language procedure systems come in handy.



High-Level Language to Machine Code

- **High-Level Language:** If a program contains pre-processor directives such as #include or #define it is called HLL. They are closer to humans but far from machines. These (#) tags are called preprocessor directives. They direct the pre-processor about what to do.
- **Pre-Processor:** The pre-processor removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion. It performs file inclusion, augmentation, macro-processing, etc. For example: Let in the source program, it is written #include “Stdio. h”. Pre-Processor replaces this file with its contents in the produced output.
- **Assembly Language:** It’s neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.

- **Assembler:** For every platform (Hardware + OS) we will have an assembler. They are not universal since for each platform we have one. The output of the assembler is called an object file. Its translates assembly language to machine code.
- **Compiler:** The compiler is an intelligent program as compared to an assembler. The compiler verifies all types of limits, ranges, errors, etc. Compiler program takes more time to run and it occupies a huge amount of memory space. The speed of the compiler is slower than other system software. It takes time because it enters through the program and then does the translation of the full program.
- **Interpreter:** An interpreter converts high-level language into low-level machine language, just like a compiler. But they are different in the way they read the input. The Compiler in one go reads the inputs, does the processing, and executes the source code whereas the interpreter does the same line by line. A compiler scans the entire program and translates it as a whole into machine code whereas an interpreter translates the program one statement at a time. Interpreted programs are usually slower concerning compiled ones.
- **Relocatable Machine Code:** It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate with the program movement.
- **Loader/Linker:** Loader/Linker converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.
    - **Linker:** The basic work of a linker is to merge object codes (that have not even been connected), produced by the compiler, assembler, standard library function, and operating system resources.
    - **Loader:** The codes generated by the compiler, assembler, and linker are generally re-located by their nature, which means to say, the starting location of these codes is not determined, which means they can be anywhere in the computer memory. Thus the basic task of loaders to find/calculate the exact address of these memory locations.

Overall, compiler design is a complex process that involves multiple stages and requires a deep understanding of both the programming language and the target platform. A well-designed compiler can greatly improve the efficiency and performance of software programs, making them more useful and valuable for users.

Phases of a Compiler

There are two major phases of compilation, which in turn have many parts. Each of them takes input from the output of the previous level and works in a coordinated way.

Analysis Phase

An intermediate representation is created from the given source code :

- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Intermediate Code Generator

Synthesis Phase

An equivalent target program is created from the intermediate representation. It has two parts :

- Code Optimizer
- Code Generator

Read more about Phases of Compiler, Here.

Compiler Construction Tools

Compiler construction tools are specialized software that help developers create compilers more efficiently. Here are the key tools:

1. **Parser Generators:** It creates syntax analyzers (parsers) based on grammatical descriptions of programming languages.
2. **Scanner Generators:** It produces lexical analyzers using regular expressions to define the tokens of a language.
3. **Syntax-Directed Translation Engines:** It generates intermediate code in three-address format from input comprising a parse tree.
4. **Automatic Code Generators:** It converts intermediate language into machine language using template matching techniques.
5. **Data-Flow Analysis Engines:** It supports code optimization by analyzing the flow of values throughout different parts of the program.
6. **Compiler Construction Toolkits:** It provides integrated routines to facilitate the construction of various compiler components.

Types of Compiler

- **Self Compiler:** When the compiler runs on the same machine and produces machine code for the same machine on which it is running then it is called as self compiler or resident compiler.
- **Cross Compiler**: The compiler may run on one machine and produce the machine codes for other computers then in that case it is called a cross-compiler. It is capable of creating code for a platform other than the one on which the compiler is running.
- **Source-to-Source Compiler:** A Source-to-Source Compiler or trans compiler or tran spiler is a compiler that translates source code written in one programming language into the source code of another programming language.
- **Single Pass Compiler:** When all the phases of the compiler are present inside a single module, it is simply called a single-pass compiler. It performs the work of converting source code to machine code.
- **Two Pass Compiler:** Two-pass compiler is a compiler in which the program is translated twice, once from the front end and the back from the back end known as Two Pass Compiler.
- **Multi-Pass Compiler:** When several intermediate codes are created in a program and a syntax tree is processed many times, it is called Multi-Pass Compiler. It breaks codes into smaller programs.
- **Just-in-Time (JIT) Compiler:** **I**t is a type of compiler that converts code into machine language during program execution, rather than before it runs. It combines the benefits of interpretation (real-time execution) and traditional compilation (faster execution).
- **Ahead-of-Time (AOT) Compiler:** It converts the entire source code into machine code before the program runs. This means the code is fully compiled during development, resulting in faster startup times and better performance at runtime.
- **Incremental Compiler:** It compiles only the parts of the code that have changed, rather than recompiling the entire program. This makes the compilation process faster and more efficient, especially during development.

Operations of Compiler

These are some operations that are done by the compiler.

- It breaks source programs into smaller parts.
- It enables the creation of symbol tables and intermediate representations.
- It helps in code compilation and error detection.
- it saves all codes and variables.
- It analyses the full program and translates it.

- Convert source code to machine code.

Advantages of Compiler Design

- **Efficiency:** Compiled programs are generally more efficient than interpreted programs because the machine code produced by the compiler is optimized for the specific hardware platform on which it will run.
- **Portability:** Once a program is compiled, the resulting machine code can be run on any computer or device that has the appropriate hardware and operating system, making it highly portable.
- **Error Checking:** Compilers perform comprehensive error checking during the compilation process, which can help catch syntax, semantic, and logical errors in the code before it is run.
- **Optimizations:** Compilers can make various optimizations to the generated machine code, such as eliminating redundant instructions or rearranging code for better performance.

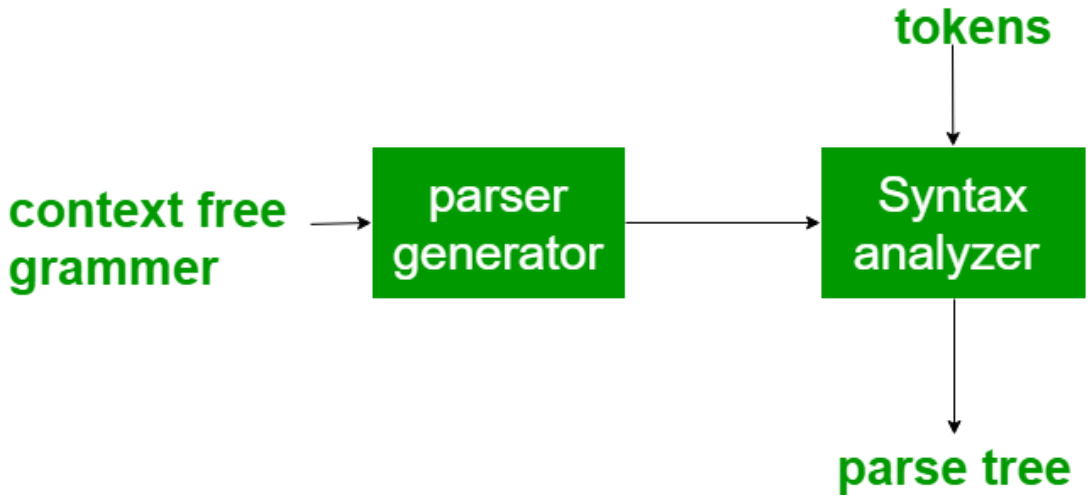
Disadvantages of Compiler Design

- **Longer Development Time:** Developing a compiler is a complex and time-consuming process that requires a deep understanding of both the programming language and the target hardware platform.
- **Debugging Difficulties:** Debugging compiled code can be more difficult than debugging interpreted code because the generated machine code may not be easy to read or understand.
- **Lack of Interactivity:** Compiled programs are typically less interactive than interpreted programs because they must be compiled before they can be run, which can slow down the development and testing process.
- **Platform-Specific Code:** If the compiler is designed to generate machine code for a specific hardware platform, the resulting code may not be portable to other platforms.

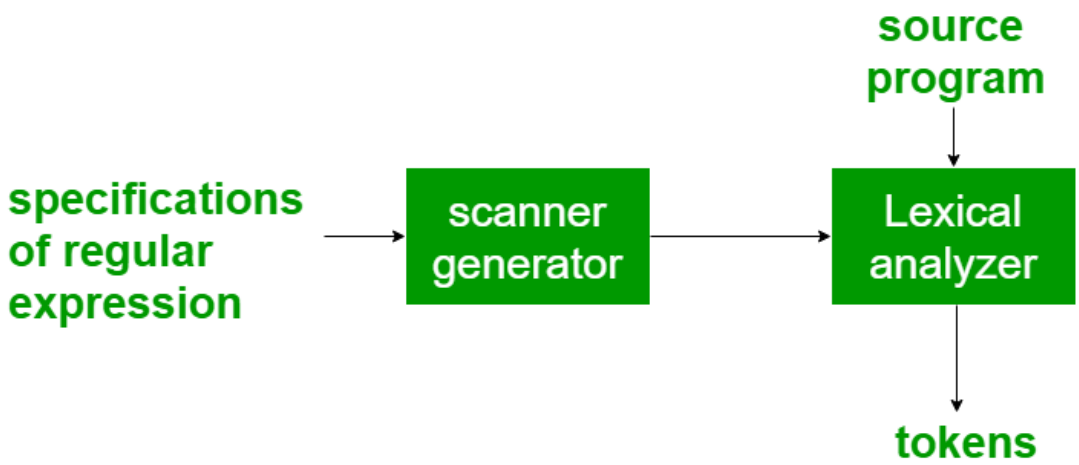
Compiler construction tools

The compiler writer can use some specialized tools that help in implementing various phases of a compiler. These tools assist in the creation of an entire compiler or its parts. Some commonly used compiler construction tools include:

1. **Parser Generator –** It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time. Example: PIC, EQM



2. **Scanner Generator –** It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language. It generates a finite automaton to recognize the regular expression. Example: Lex

3. 

4. **Syntax directed translation engines –** It generates intermediate code with three address format from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produces the intermediate code. In this, each node of the parse tree is associated with one or more translations.
5. **Automatic code generators –** It generates the machine language for a target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator. A template matching process is used. An intermediate language statement is replaced by its equivalent machine language statement using templates.
6. **Data-flow analysis engines –** It is used in code optimization.Data flow analysis is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another. Refer – data flow analysis in Compiler
7. **Compiler construction toolkits –** It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.

Features of compiler construction tools :

Lexical Analyzer Generator: This tool helps in generating the lexical analyzer or scanner of the compiler. It takes as input a set of regular expressions that define the syntax of the language being compiled and produces a program that reads the input source code and tokenizes it based on these regular expressions.
Parser Generator: This tool helps in generating the parser of the compiler. It takes as input a context-free grammar that defines the syntax of the language being compiled and produces a program that parses the input tokens and builds an abstract syntax tree.
Code Generation Tools: These tools help in generating the target code for the compiler. They take as input the abstract syntax tree produced by the parser and produce code that can be executed on the target machine.
Optimization Tools: These tools help in optimizing the generated code for efficiency and performance. They can perform various optimizations such as dead code elimination, loop optimization, and register allocation.

Debugging Tools: These tools help in debugging the compiler itself or the programs that are being compiled. They can provide debugging information such as symbol tables, call stacks, and runtime errors.
Profiling Tools: These tools help in profiling the compiler or the compiled code to identify performance bottlenecks and optimize the code accordingly.
Documentation Tools: These tools help in generating documentation for the compiler and the programming language being compiled. They can generate documentation for the syntax, semantics, and usage of the language.
Language Support: Compiler construction tools are designed to support a wide range of programming languages, including high-level languages such as C++, Java, and Python, as well as low-level languages such as assembly language.
Cross-Platform Support: Compiler construction tools may be designed to work on multiple platforms, such as Windows, Mac, and Linux.
User Interface: Some compiler construction tools come with a user interface that makes it easier for developers to work with the compiler and its associated tools.

Phases of a Compiler

- A compiler is a software tool that converts high-level programming code into machine code that a computer can understand and execute. It acts as a bridge between human-readable code and machine-level instructions, enabling efficient program execution. The process of compilation is divided into six phases:

1. **Lexical Analysis:** The first phase, where the source code is broken down into tokens such as keywords, operators, and identifiers for easier processing.
2. **Syntax Analysis or Parsing:** This phase checks if the source code follows the correct syntax rules, building a parse tree or abstract syntax tree (AST).
3. **Semantic Analysis:** It ensures the program's logic makes sense, checking for errors like type mismatches or undeclared variables.
4. **Intermediate Code Generation:** In this phase, the compiler converts the source code into an intermediate, machine-independent representation, simplifying optimization and translation.
5. **Code Optimization:** This phase improves the intermediate code to make it run more efficiently, reducing resource usage or increasing speed.
6. **Target Code Generation:** The final phase where the optimized code is translated into the target machine code or assembly language that can be executed on the computer.

The whole compilation process is divided into two parts, front-end and back-end. These six phases are divided into two main parts, front-end and back-end with the intermediate code generation phase acting as a link between them. The front end analyzes source code for syntax and semantics, generating intermediate code, while ensuring correctness. The back end optimizes this intermediate code and converts it into efficient machine code for execution. The front end is mostly machine-independent, while the back end is machine-dependent.

The compilation process is an essential part of transforming high-level source code into machine-readable code. A compiler performs this transformation through several phases, each with a specific role in making the code efficient and correct. Broadly, the compilation process can be divided into two main parts:

1. **Analysis Phase:** The analysis phase breaks the source program into its basic components and creates an intermediate representation of the program. It is sometimes referred to as front end.
2. **Synthesis Phase:** The synthesis phase creates the final target program from the intermediate representation. It is sometimes referred to as back end.

Phases of a Compiler

The compiler consists of two main parts: the front-end and the back-end. The front-end includes the lexical analyzer, syntax analyzer, semantic analyzer, and intermediate code generator. The back-end takes over from there, handling optimization, code generation, and assembly.

1. Lexical Analysis

Lexical analysis is the first phase of a compiler, responsible for converting the raw source code into a sequence of tokens. A token is the smallest unit of meaningful data in a programming language. Lexical analysis involves scanning the source code, recognizing patterns, and categorizing groups of characters into distinct tokens.

The lexical analyzer scans the source code character by character, grouping these characters into meaningful units (tokens) based on the language's syntax rules. These tokens can represent keywords, identifiers, constants, operators, or punctuation marks. By converting the source code into tokens, lexical analysis simplifies the process of understanding and processing the code in later stages of compilation.

2. Syntax Analysis

Syntax analysis, also known as parsing, is the second phase of a compiler where the structure of the source code is checked. This phase ensures that the code follows the correct grammatical rules of the programming language.

The role of syntax analysis is to verify that the sequence of tokens produced by the lexical analyzer is arranged in a valid way according to the language's syntax. It checks whether the code adheres to the language's rules, such as correct use of operators, keywords, and parentheses. If the source code is not structured correctly, the syntax analyzer will generate errors.

To represent the structure of the source code, syntax analysis uses parse trees or syntax trees.

- **Parse Tree:** A parse tree is a tree-like structure that represents the syntactic structure of the source code. It shows how the tokens relate to each other according to the grammar rules. Each branch in the tree represents a production rule of the language, and the leaves represent the tokens.
- **Syntax Tree:** A syntax tree is a more abstract version of the parse tree. It represents the hierarchical structure of the source code but with less detail, focusing on the essential syntactic structure. It helps in understanding how different parts of the code relate to each other.

3. Semantic Analysis

Semantic analysis is the phase of the compiler that ensures the source code makes sense logically. It goes beyond the syntax of the code and checks whether the program has any semantic errors, such as type mismatches or undeclared variables.

Semantic analysis checks the meaning of the program by validating that the operations performed in the code are logically correct. This phase ensures that the source code follows the rules of the programming language in terms of its logic and data usage.
Some key checks performed during semantic analysis include:

- Type Checking: The compiler ensures that operations are performed on compatible data types. For example, trying to add a string and an integer would be flagged as an error because they are incompatible types.
- Variable Declaration: It checks whether variables are declared before they are used. For example, using a variable that has not been defined earlier in the code would result in a semantic error.

Type Checking:

- *a is int and b is float. Adding them (a + b) results in float, which cannot be assigned to int a.*
- ***Error:*** *Type mismatch: cannot assign float to int.*

4. Intermediate Code Generation

Intermediate code is a form of code that lies between the high-level source code and the final machine code. It is not specific to any particular machine, making it portable and easier to optimize. Intermediate code acts as a bridge, simplifying the process of converting source code into executable code.
The use of intermediate code plays a crucial role in optimizing the program before it is turned into machine code.

- **Platform Independence:** Since the intermediate code is not tied to any specific hardware, it can be easily optimized for different platforms without needing to recompile the entire source code. This makes the process more efficient for cross-platform development.
- **Simplifying Optimization:** Intermediate code simplifies the optimization process by providing a clearer, more structured view of the program. This makes it easier to apply optimization techniques such as:
    - **Dead Code Elimination:** Removing parts of the code that don't affect the program's output.
    - **Loop Optimization:** Improving loops to make them run faster or consume less memory.
    - **Common Subexpression Elimination:** Reusing previously calculated values to avoid redundant calculations.
- **Easier Translation:** Intermediate code is often closer to machine code, but not specific to any one machine, making it easier to convert into the target machine code. This step is typically handled in the back end of the compiler, allowing for smoother and more efficient code generation.

5. Code Optimization

Code Optimization is the process of improving the intermediate or target code to make the program run faster, use less memory, or be more efficient, without altering its functionality. It involves techniques like removing unnecessary computations, reducing redundancy, and reorganizing code to achieve better performance. Optimization is classified broadly into two types:

- Machine-Independent
- Machine-Dependent

Common Techniques:

- **Constant Folding**: Precomputing constant expressions.

- **Dead Code Elimination**: Removing unreachable or unused code.
- **Loop Optimization**: Improving loop performance through invariant code motion or unrolling.
- **Strength Reduction**: Replacing expensive operations with simpler ones.

Example:

Code Before Optimization	Code After Optimization

for (int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
	x = y + z ;
	for (int j = 0 ; j < n ; j ++)
	{
	a[j] = 6 x j;
	}

6. Code Generation

Code Generation is the final phase of a compiler, where the intermediate representation of the source program (e.g., three-address code or abstract syntax tree) is translated into machine code or assembly code. This machine code is specific to the target platform and can be executed directly by the hardware.

The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. The source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

Example:

Three Address Code	Assembly Code

t1 = c * d
t2 = b + t1
a = t2
	LOAD R1, c ; Load the value of ‘c’ into register R1
	LOAD R2, d ; Load the value of ‘d’ into register R2
	MUL R1, R2 ; R1 = c * d, store result in R1
	LOAD R3, b ; Load the value of ‘b’ into register R3
	ADD R3, R1 ; R3 = b + (c * d), store result in R3
	STORE a, R3 ; Store the final result in variable ‘a’