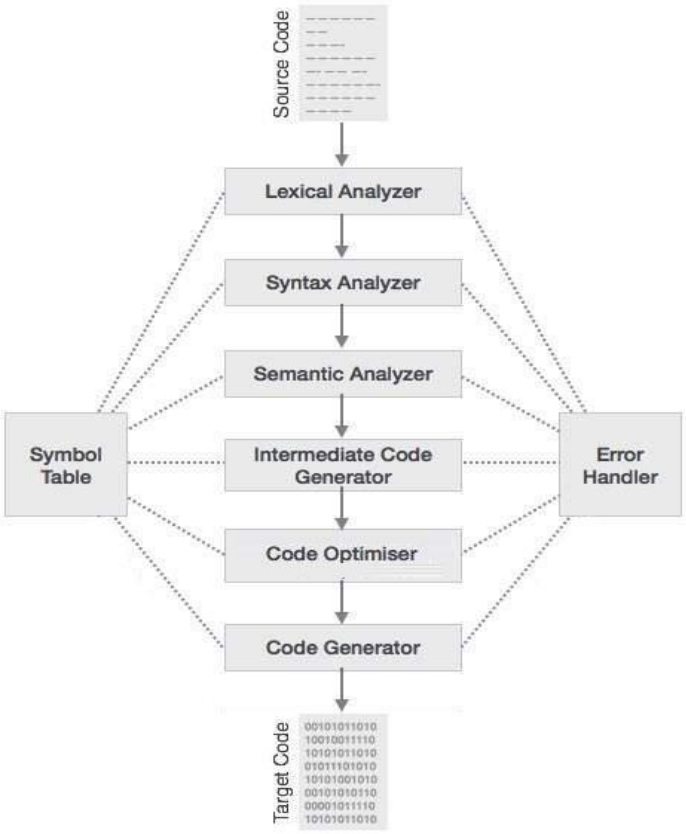
Phases Of Compiler

Compiler operates in various phases each phase transforms the source program from one representation to another. Every phase takes inputs from its previous stage and feeds its output to the next phase of the compiler.

There are 6 phases in a compiler. Each of this phase help in converting the high-level language into the machine code. The phases of a compiler are:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generator
5. Code optimizer
6. Code generator



1. Lexical Analysis

Lexical analysis is 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.

- 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.

Example: int x = 10;

The lexical analyzer would break this line into the following tokens:

- **int** - Keyword token (data type)
- **x -** Identifier token (variable name)
- **= -** Operator token (assignment operator)
- **10** - Numeric literal token (integer value)
- **; -** Punctuation token (semicolon, used to terminate statements)

2. Syntax Analysis or Parsing

This phase ensures that the code follows the correct grammatical rules of the programming language.

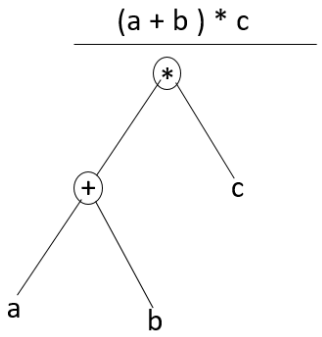
- Verifies that the sequence of tokens produced by the lexical analyzer is arranged in a valid way according to the language's syntax.
- It checks whether the code adheres to the language's rules, such as correct use of operators, keywords, and parentheses.
- If the source code is not structured correctly,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

For Example: Consider parse tree for the following example.

- (a + b) * c



3. Semantic Analysis

Semantic analysis is the phase of the compiler that ensures the source code makes sense logically.

- It checks whether the program has any semantic errors, such as type mismatches or undeclared variables.
- 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.

Example:

int a = 5;

float b = 3.5;

a = a + b;

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.

***Example**: a = b + c * d;*

*t1 = c * d*

t2 = b + t1

a = t2

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.

- Involves techniques like removing unnecessary computations, reducing redundancy, and reorganizing code to achieve better performance.
- Optimization is classified broadly into two types Machine-Independent & 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.

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

Three Address Code	Assembly Code
	ADD R3, R1 ; R3 = b + (c * d), store result in R3 STORE a, R3 ; Store the final result in variable 'a'

Symbol Table - It is a data structure being used and maintained by the compiler, consisting of all the identifier's names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

Front end and Back-end Compiler:

The phases of a compiler can be grouped as:

- **Front end:** Front end consists of all those phases which are dependent on the input (source language) and independent on the target machine (target language).
- Front end of a compiler consists of the phases,
  - Lexical analysis.
  - Syntax analysis.
  - Semantic analysis.
  - Intermediate code generation.

- **Back end:** Back end consists of all those phases of the compiler that are dependent on the target machine and independent on the source language.
- Back end of a compiler contains,
  - Code optimization.
  - Code generation.

Phase and Pass:

Phase:

- Phase is a logical division of compilation process source program into some other form.

Pass:

- Pass means reading the source program / code once.

Difference:

The main difference between **phases** and **passes** of **compiler** is that **phases** are the steps in the compilation process while **passes** are the number of times the **compiler** traverses through the source code.

Types of Compilers:

Following are the types of compilers:

- Single Pass Compiler
- Two Pass Compiler
- Multipass Compiler

Single Pass Compiler

- In single pass Compiler, source code directly transforms into machine code.
- **For example**, Pascal language.

Two Pass Compiler:

- In two pass Compiler, source code is not directly transforms into machine code, it is divided into two sections.
  - **Front end:** It translate source code into Intermediate Representation (IR).
  - **Back end:** It maps IR onto the target machine
- The Two-pass compiler method also simplifies the retargeting process.

Multi Pass Cor

- The Multipass compiler processes the source code or syntax tree of a program several times.

- It divided a large program into multiple small programs and process them. It develops multiple intermediate codes. All of these Multipass take the output of the previous phase as an input. Therefore, it requires less memory. It is also known as '**Wide Compiler**'.