Name: Moustafa Hassan Sadek

ID: 200037705

Phases of 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:

Lexical Analysis: The first phase, where the source code is broken down into tokens such as keywords, operators, and identifiers for easier processing.

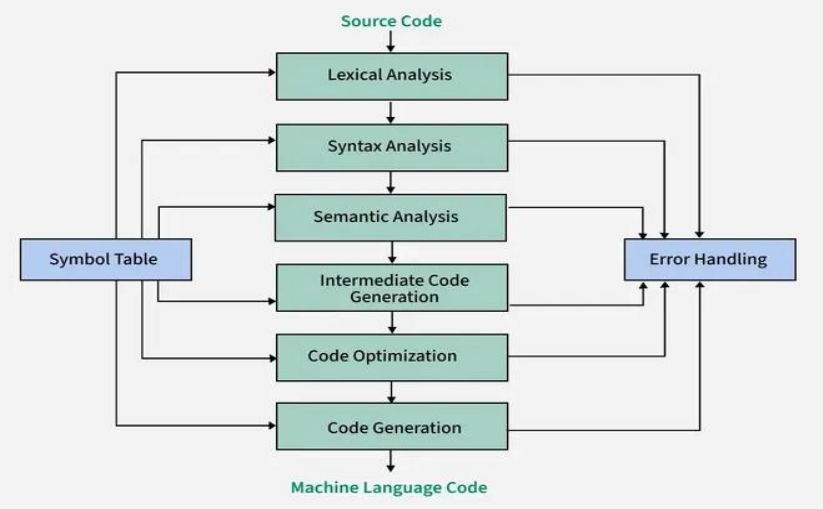
Syntax Analysis or Parsing: This phase checks if the source code follows the correct syntax rules, building a parse tree or abstract syntax tree (AST).

Semantic Analysis: It ensures the program's logic makes sense, checking for errors like type mismatches or undeclared variables.

Intermediate Code Generation: In this phase, the compiler converts the source code into an intermediate, machine-independent representation, simplifying optimization and translation.

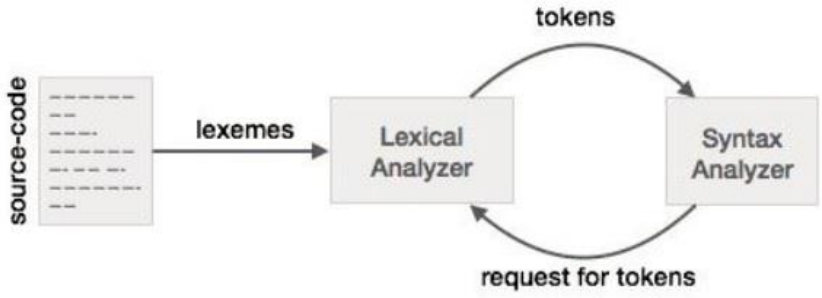
Code Optimization: This phase improves the intermediate code to make it run more efficiently, reducing resource usage or increasing speed.

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.



Lexical analysis **is the first phase of a compiler. It takes modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.**

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



I only used c++ in this project and here is my Implementation of a Lexical Analyzer:

```
#include <iostream>

#include <regex>

#include <map>

#include <vector>

using namespace std;

map<string, string> Make_Regex_Map() { return
{ {R"(\b(int|float|char|bool|if|else|while|for|return|main|cin|cout|using|namespace|std
)\b)", "Keyword"}, {R"(\b(include|define)\b)", "Preprocessor"},
{R"(\b(iostream|stdio|string)\b)", "Library"}, {R"([a-zA-Z_][a-zA-Z0-9_]*)", "Identifier"},
{R"(\b\d+\b)", "Number"}, {R"(+|-|*|/|=|==|<<|>>|<|>)", "Operator"}, {R"({|}|(|)|;|,)",
"Symbol"} }; }

vector<pair<string, string>> Analyze_Code(const string& code) { map<string, string>
patterns = Make_Regex_Map(); vector<pair<string, string>> tokens;

for (const auto& [pattern, type] : patterns) {
   regex re(pattern);
   auto begin = sregex_iterator(code.begin(), code.end(), re);
   auto end = sregex_iterator();

   for (auto it = begin; it != end; ++it) {
      tokens.push_back({ it->str(), type });
   }
}

return tokens;


}

int main() { string code; cout << "Enter your code:\n> "; getline(cin, code);

auto tokens = Analyze_Code(code);

cout << "\nLexeme\t\tToken\n-----------------------------\n";
```

```
for (const auto& [lexeme, token] : tokens) {
   cout << lexeme << "\t\t" << token << endl;
}

return 0;


}
```

Code explanation:

this code simply defines all the token types using regular expressions and then scan the input line using regex patterns looking for a match and for each match it stores the lexeme and its token in a list

References:

Geeksforgeeks, stanford, coursera, concept of programming book.