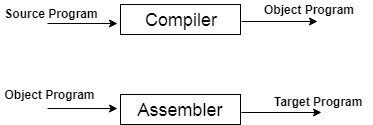
UNIT 1

Introduction to Compiler

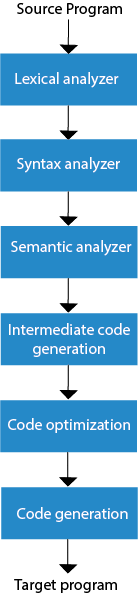
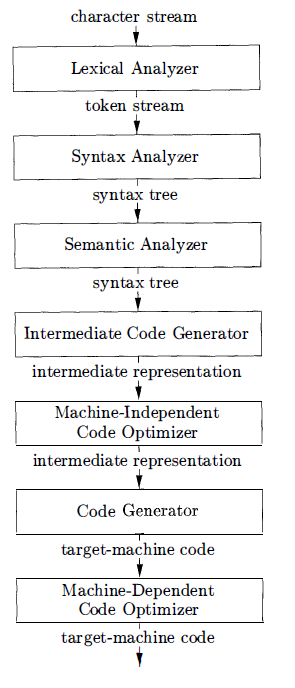
* A compiler is a translator that converts the high-level language into the machine language.
* High-level language is written by a developer and machine language can be understood by the processor.
* Compiler is used to show errors to the programmer.
* The main purpose of compiler is to change the code 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.



# Compiler Phases

The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

There are the various phases of compiler:



**Fig: phases of compiler**

### **Lexical Analysis:**

Lexical analyzer phase is the first phase of compilation process. It takes source code as input. It reads the source program one character at a time and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens.

### **Syntax Analysis**

Syntax analysis is the second phase of compilation process. It takes tokens as input and generates a parse tree as output. In syntax analysis phase, the parser checks that the expression made by the tokens is syntactically correct or not.

### **Semantic Analysis**

Semantic analysis is the third phase of compilation process. It checks whether the parse tree follows the rules of language. Semantic analyzer keeps track of identifiers, their types and expressions. The output of semantic analysis phase is the annotated tree syntax.

### **Intermediate Code Generation**

In the intermediate code generation, compiler generates the source code into the intermediate code. Intermediate code is generated between the high-level language and the machine language. The intermediate code should be generated in such a way that you can easily translate it into the target machine code.

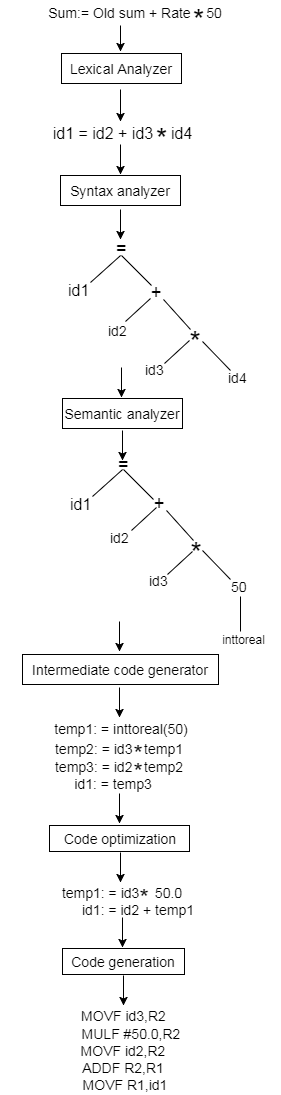
### **Code Optimization**

Code optimization is an optional phase. It is used to improve the intermediate code so that the output of the program could run faster and take less space. It removes the unnecessary lines of the code and arranges the sequence of statements in order to speed up the program execution.

### **Code Generation**

Code generation is the final stage of the compilation process. It takes the optimized intermediate code as input and maps it to the target machine language. Code generator translates the intermediate code into the machine code of the specified computer.

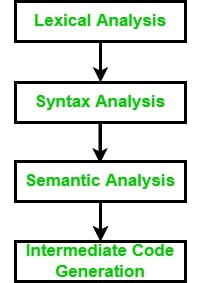
**Example:**



### Phases Grouping:

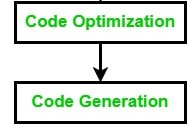
Generally, phases are divided into two parts:

**1. Front End phases:**The front end consists of those phases or parts of phases that are source language-dependent and target machine, independents. These generally consist of [lexical analysis](https://www.geeksforgeeks.org/introduction-of-lexical-analysis/), [semantic analysis](https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/), [syntactic analysis](https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/), [symbol table creation](https://www.geeksforgeeks.org/symbol-table-compiler/), and [intermediate code generation](https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/). A little part of code optimization can also be included in the front-end part. The front-end part also includes the error handling that goes along with each of the phases.



*Front End Phases*

**2. Back End** **phases:**The portions of compilers that depend on the target machine and do not depend on the source language are included in the back end. In the back end, code generation and necessary features of code optimization phases, along with error handling and symbol table operations are also included.



*Back End Phases*

### Passes in Compiler:

A pass is a component where parts of one or more phases of the compiler are combined when a compiler is implemented. A pass reads or scans the instructions of the source program or the output produced by the previous pass, which makes necessary transformation specified by its phases.

There are generally two types of passes

1. One-pass
2. Two-pass

### Grouping

Several phases are grouped togetherto a pass so that it can read the input file and write an output file.

1. One-Pass – In One-pass all the phases are grouped into one phase. The six phases are included here in one pass.
2. Two-Pass – In Two-pass the phases are divided into two parts i.e. Analysis or Front End part of the compiler and the synthesis part or back end part of the compiler.

## **Multi-pass Compiler**

* Multi pass compiler is used to process the source code of a program several times.
* In the first pass, compiler can read the source program, scan it, extract the tokens and store the result in an output file.
* In the second pass, compiler can read the output file produced by first pass, build the syntactic tree and perform the syntactical analysis. The output of this phase is a file that contains the syntactical tree.
* In the third pass, compiler can read the output file produced by second pass and check that the tree follows the rules of language or not. The output of semantic analysis phase is the annotated tree syntax.
* This pass is going on, until the target output is produced.

## **One-pass Compiler**

* One-pass compiler is used to traverse the program only once. The one-pass compiler passes only once through the parts of each compilation unit. It translates each part into its final machine code.
* In the one pass compiler, when the line source is processed, it is scanned and the token is extracted.
* Then the syntax of each line is analyzed and the tree structure is build. After the semantic part, the code is generated.
* The same process is repeated for each line of code until the entire program is compiled.

# Compiler construction tools

1. Parser generators that automatically produce syntax analyzers from a

grammatical description of a programming language.

2. Scanner generators that produce lexical analyzers fr om a regular-expression

description of the tokens of a language.

3. Syntax-directed translation engines that produce collections of routines

for walking a parse tree and generating intermediate code.

4. Code-generator generators that produce a code generator from a collection

of rules for translating each operation of the intermediate language into

the machine language for a target machine.

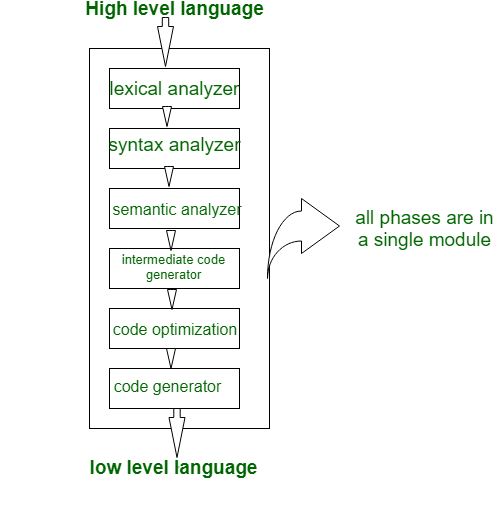
5. Data-flow analysis engines that facilitate the gathering of information

about how values are transmitted from one part of a program to each

other part. Data-flow analysis is a key part of code optimization.

6. Compiler- construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

**1. Single Pass Compiler:** If we combine or group all the phases of compiler design in a **single** module known as a single pass compiler.



 In the above diagram, there are all 6 phases are grouped in a single module, some points of the single pass compiler are as:

1. A one-pass/single-pass compiler is a type of compiler that passes through the part of each compilation unit exactly once.
2. Single pass compiler is faster and smaller than the multi-pass compiler.
3. A disadvantage of a single-pass compiler is that it is less efficient in comparison with the multipass compiler.
4. A single pass compiler is one that processes the input *exactly once*, so going directly from lexical analysis to code generator, and then going back for the next read.

**What is Lexical Analysis?**

Lexical analysis is the starting phase of the compiler. It gathers modified source code that is written in the form of sentences from the language preprocessor. The lexical analyzer is responsible for breaking these syntaxes into a series of tokens, by removing whitespace in the source code. If the lexical analyzer gets any invalid token, it generates an error. The stream of character is read by it and it seeks the legal tokens, and then the data is passed to the syntax analyzer, when it is asked for

### The Architecture of Lexical AnalyzerRoles and Responsibility of Lexical Analyzer

* The lexical analyzer is responsible for removing the white spaces and comments from the source program.
* It corresponds to the error messages with the source program.
* It helps to identify the tokens.
* The input characters are read by the lexical analyzer from the source code.

### Lexical Analysis

# What is Input Buffering in Compiler Design?

Lexical Analysis has to access secondary memory each time to identify tokens. It is time-consuming and costly. So, the input strings are stored into a buffer and then scanned by Lexical Analysis.

Lexical Analysis scans input string from left to right one character at a time to identify tokens. It uses two pointers to scan tokens −

* **Begin Pointer (bptr)** − It points to the beginning of the string to be read.
* **Look Ahead Pointer (lptr)** − It moves ahead to search for the end of the token.

## **Specification of Token in Compiler Design**

In compiler design, there are three specifications of token-

1. String
2. Language
3. Regular Expressions

## **Strings**

Strings are a finite set of symbols or characters. These symbols can be a digit or an alphabet. There is also an empty string which is denoted by **ε**.

## **Language**

A language can be defined as a finite set of strings over some symbols or alphabets.

## **Regular Expression**

Regular expressions are strings of characters that define a searching pattern with the help of which we can form a language, and each regular expression represents a language.

A regular expression**r**can denote a language **L(r)** which can be built recursively over the smaller regular expression by following some rules.

**Regular expressions,**

• Regular expressions are a notation to represent lexeme patterns for a token.

• They are used to represent the language for lexical analyzer.

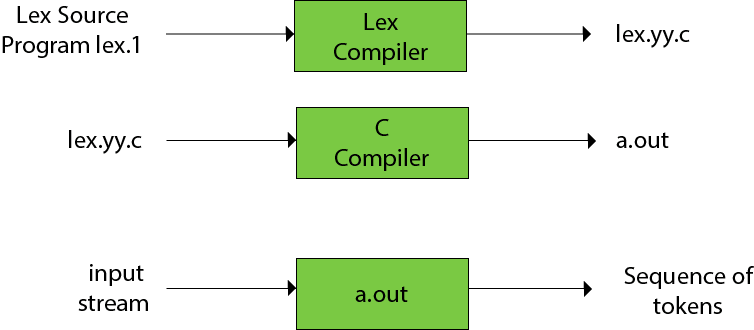
• They assist in finding the type of token that accounts for a particular lexeme.

**Lexical Analyzer generator (Lex)**

* Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
* The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
* It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

### **The function of Lex is as follows:**

* Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
* Finally C compiler runs the lex.yy.c program and produces an object program a.out.
* a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



## **Lex file format**

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:

1. { definitions }
2. %%
3. { rules }
4. %%
5. { user subroutines }