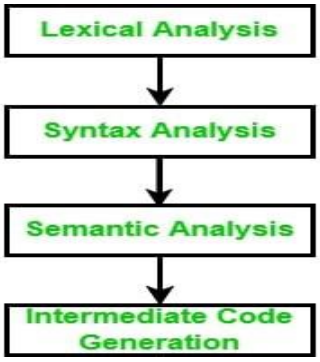
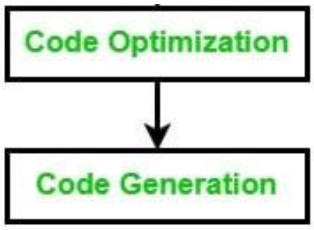
GROUPING OF THE PHASES

Compiler can be grouped into front and back ends:

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, semantic analysis, syntactic analysis, symbol table creation, and intermediate code generation. 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.



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.



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

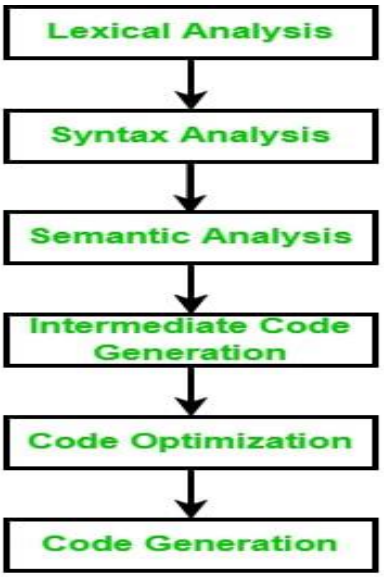
1. One-pass
2. Two-pass

Grouping

Several phases are grouped together to 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.

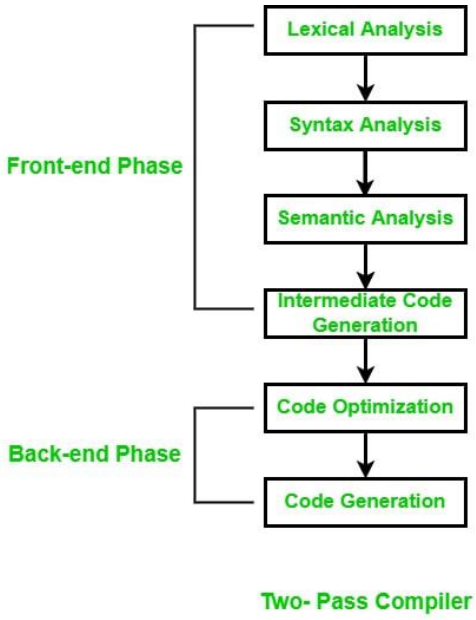
Purpose of One Pass Compiler



One- Pass Compiler

A one-pass compiler generates a structure of machine instructions as it looks like a stream of instructions and then sums up with machine address for these guidelines to a rundown of directions to be backpatched once the machine address for it is generated. It is used to pass the program for one time. Whenever the line source is handled, it is checked and the token is removed.

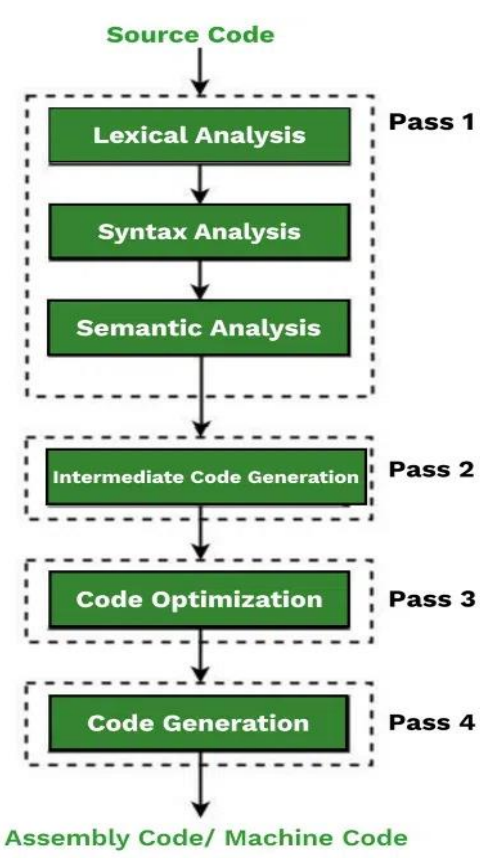
Purpose of Two-Pass Compiler



A two-pass compiler utilizes its first pass to go into its symbol table a rundown of identifiers along with the memory areas to which these identifiers relate. Then, at that point, a second pass replaces mnemonic operation codes by their machine language equivalent and replaced uses of identifiers by their machine address. In the second pass, the compiler can read the result document delivered by the first pass, assemble the syntactic tree and deliver the syntactical examination. The result of this stage is a record that contains the syntactical tree.

Multi-Pass Compiler

In Multi-Pass Compiler, the program code goes via multiple passes during the process of compilation. It also generated intermediate optimized code for the completion of each step. Later, it translates the source program into 1 or more intermediate steps between program code and machine code. Multi-Pass compiler makes the program code perform parsing, analyzing, code generation, etc processes multiple times. Here, each pass takes the result of the previous pass as the input and then generates the intermediate code or output.



Passes

From the above diagram, we can see that in the first pass, the compiler analyses the source/program code, generates the token, and saves the result for the next step. In Pass 2, the compiler read the result created by Pass 1, builds the parse tree, and performs the syntactical analysis. Output is also generated here for the next step.

In Pass 3, the compiler reads the output of Pass 2 and checks the grammar using the tree. Output is created and given as the input to Pass 4. This passes continues till the final output is been generated.