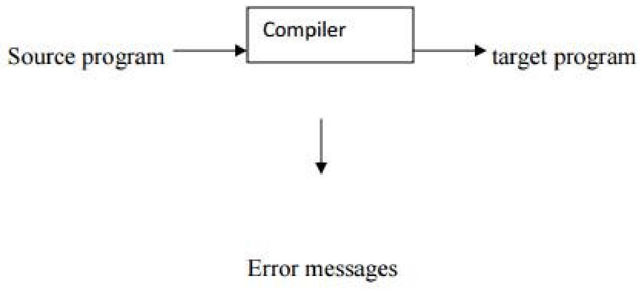
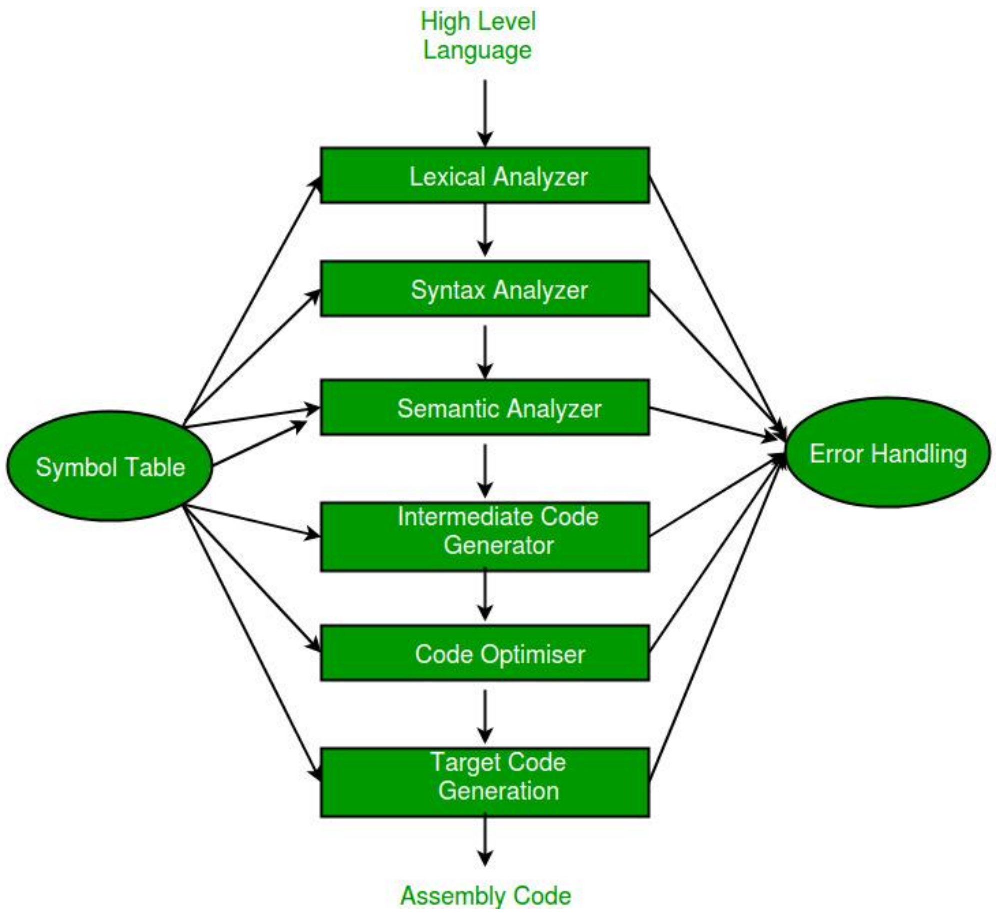
## Define Compilers. Explain phases of a compiler

- A compiler is a software program that converts the high-level source code written in a programming language into low-level machine code that can be executed by the computer hardware.
- The process of converting the source code into machine code involves several phases or stages, which are collectively known as the phases of a compiler.



The typical phases of a compiler are :



**1. Analysis Phase:**

• In the Analysis Phase, the language processor takes the source program as input and performs a comprehensive analysis of it.

• This phase involves several subtasks, including lexical analysis, syntax analysis,

and semantic analysis.

• **Lexical Analysis:** This is the initial step where the source code is broken down into tokens or lexemes. Lexemes are the smallest units of meaning in a programming language, such as keywords, identifiers, and literals. Lexical analysis helps identify and categorize these tokens.

• **Syntax Analysis:** Once the tokens are identified, the language processor checks whether they follow the syntax rules of the programming language. This involves constructing a parse tree or syntax tree to represent the grammatical structure of the source code. If any syntax errors are found, they are reported.

• **Semantic Analysis:** Beyond syntax, this phase verifies the meaning of the program. It checks for semantic errors, such as type mismatches or undeclared variables. Additionally, it may perform optimizations and symbol table management.

Consider the following example:

**percent_profit = (profit * 100) / cost_price;**

• Lexical units identifies =, * and / operators, 100 as constant, and the remaining strings as identifiers.

• Syntax analysis identifies the statement as an assignment statement with percent_profit as the left hand side and (profit * 100) / cost_price as the expression on the right hand side.

• Semantic analysis determines the meaning of the statement to be the assignment of profit X 100 / cost_price to percent_profit.

**2. Synthesis Phase:**

• After successfully analyzing the source program, the language processor proceeds to the Synthesis Phase.

• In this phase, the processor generates the target code (e.g., machine code or intermediate code) based on the analysis performed in the previous phase.

- This phase includes tasks like code generation and code optimization.

• **Intermediate Code Generation:** Here, the language processor translates the high-level

source code into low-level target code. This can involve choosing appropriate machine instructions or generating intermediate code that will be further translated or executed.

• **Code Optimization**: Depending on the compiler or interpreter, this optional step involves improving the efficiency of the target code. It may include optimizations like dead code elimination, loop unrolling, or register allocation.

• **Code Generation:** The final phase of a compiler is code generation. This phase takes the optimized intermediate code and generates the actual machine code that can be executed by the target hardware.

## Explain different following tools for which compiler technology is used to create:

**Structure editors :**

- A structure editor takes as input a sequence of commands to build a source program.
- The structure editor not only performs the text creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
- structure editors operate at a higher level of abstraction, understanding the syntax and semantics of the programming language being used.
- Thus, the structure editor can perform additional tasks that are useful in the preparation of programs.

**Pretty Printers :**

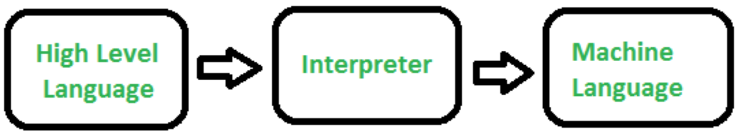
- A pretty printer is a software tool used to format source code in a visually appealing and structured manner.
- Its primary purpose is to improve code readability by presenting it in a clear and organized layout.
- Pretty printers analyze the syntax and structure of the program code and then output it with consistent indentation, spacing, and formatting conventions.
- The goal is to make the code easier to understand and maintain by highlighting its structure and logical flow.

**Static Checkers :**

- A static checker, also known as a static analysis tool or static code analyzer, is a software tool used in software development to analyze source code without executing it.
- The primary goal of a static checker is to identify potential bugs, security vulnerabilities, coding errors, and adherence to coding standards by examining the code's structure, syntax, and semantics.
- They identify security vulnerabilities in the code, such as injection attacks, insecure cryptographic implementations.
- Unlike dynamic analysis tools that require executing the program, static checkers operate solely on the source code level.

## Interpreters:

- The software by which the conversion of the high-level instructions is performed line-by-line to machine-level language, other than compiler and assembler, is known as INTERPRETER**.**
- The interpreter in the compiler checks the source code line-by-line and if an error is found on any line, it stops the execution until the error is resolved.
- Error correction is quite easy for the interpreter as the interpreter provides a line-by-line error. But the program takes more time to complete the execution successfully.



## Silicon Compilers :

- A silicon compiler is a type of electronic design automation (EDA) software tool used in the field of integrated circuit (IC) design.
- Silicon compilers are a specialized subset within the field of compiler construction, focusing specifically on the synthesis and optimization of digital circuits for implementation on silicon chips.
- Silicon compilers perform various transformations and optimizations on the high-level hardware descriptions to improve the resulting circuit's performance.
- Silicon compilers may offer customization and extensibility features.

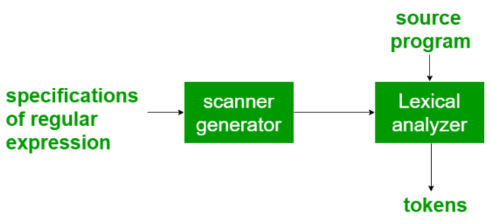
## Explain different compiler construction tools.

- A compiler is a software program that converts the high-level source code written in a programming language into low-level machine code that can be executed by the computer hardware.
- Compiler Construction Tools are specialized tools that help in the implementation of various phases of a compiler.

Some of the commonly used compiler constructions tools are:-

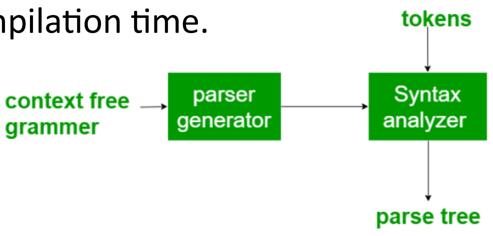
### Scanner Generator :

- 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.
- One well-known example of a scanner generator is Lex, which is commonly used in conjunction with the yacc parser generator.
- Lex reads a set of regular expression rules from an input file and generates a lexical analyzer in C or other programming languages.
- The generated lexical analyzer can be integrated into a compiler or other software tools to tokenize input streams according to the specified lexical rules.



### Parser Generator :

- A parser generator is a software tool used in compiler construction to generate syntax analyzers, also known as parsers.
- It generates parsers from descriptions of grammars that define the syntax of a programming language or specify its context-free grammar.
- Parser generators automate the process of building parsers, which are essential components of compilers responsible for analyzing the syntactic structure of source code.
- It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.

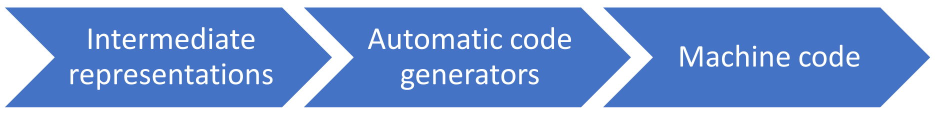


**Syntax directed translation engines :**

- Syntax-directed translation engines are components of compilers that generate intermediate code.
- 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.

**Automatic code generators :**

- Automatic code generators generates the machine language for a target machine.
- Automatic code generators are components of compilers responsible for translating intermediate code, typically in the form of intermediate representations (IR), into machine code for a specific target machine architecture.
- These code generators automate the process of converting high-level intermediate code into low-level machine instructions by applying a set of rules and templates.



**Data-flow analysis engines :**

- Data-flow analysis engines are integral components of compilers used for code optimization.
- These engines analyze the flow of data throughout a program to gather information about how values propagate from one part of the program to another.
- This analysis helps compilers make informed decisions about optimizations that can improve program performance, reduce resource usage, and enhance code quality.

**Compiler construction toolkits :**

- Compiler construction toolkits are software packages or libraries that provide developers with a comprehensive set of tools, routines, and utilities.
- It facilitate the building and development of compiler components or entire compiler systems**.**
- These toolkits are designed to streamline the process of compiler construction by offering reusable building blocks, algorithms, and data structures commonly used in various phases of compiler design and implementation.

## Explain following cousins of compiler

**Linker :**

- Linker is a program in a system which helps to link object modules of a program into a single object file.It performs the process of linking.
- Linkers are also called as link editors. Linking is a process of collecting and maintaining piece of code and data into a single file.
- In the object file, linker searches and append all libraries needed for execution of file.
- It regulates the memory space that will hold the code from each module. It also merges two or more separate object programs and establishes link among them.
- Linking is performed at both compile time, when the source code is translated into machine code and load time, when the program is loaded into memory by the loader.
- Generally, linkers are of two types :
  **1.** Linkage Editor
  **2.** Dynamic Linker

**Loader :**

- It is special program that takes input of executable files from linker, loads it to main memory, and prepares this code for execution by computer**.**
- Loader allocates memory space to program. Even it settles down symbolic reference between objects.
- It is in charge of loading programs and libraries in operating system.
- The embedded computer systems don't have loaders. In them, code is executed through ROM.
- There are following various loading schemes:
  1. Absolute Loaders
  2. Relocating Loaders
  3. Direct Linking Loaders
  4. Bootstrap Loader

**The loader performs several tasks, including:**

- Loading: The loader loads the executable file into memory and allocates memory for the program.

- Relocation: The loader adjusts the program's memory addresses to reflect its location in memory.
- Symbol resolution: The loader resolves any unresolved external symbols that are required by the program.
- Dynamic linking: The loader can dynamically link libraries into the program at runtime to provide additional functionality.

**Assembler :**

- An assembler is a software tool used in the compilation process to translate assembly language programs into machine code or object code.
- Assembly language is a low-level programming language that uses mnemonic instructions to represent machine instructions directly.
- Assemblers are essential in the process of converting human-readable assembly language code into binary machine code that can be executed by a computer's CPU.
- A programmer uses a sequence of these assembler instructions to write the source code. The assembler program then takes each statement in the source program and generates a corresponding bit stream or pattern called the *object code*.
- The assembler identifies the symbolic names associated with each instruction and allocates memory to each instruction.
- It also maintains a program counter or location counter (LC) to keep track of the memory addresses of every instruction.
- There are two main types of assemblers - **Single-pass assembler and Multipass assembler.**

**Single-pass assembler :**

- A single-pass or one-pass assembler translates the entire assembly language program into its equivalent machine language program in one go.

**Multipass assembler :**

- In a multipass assembler, the assembler must scan the assembly language program multiple times before it can be translated to its equivalent machine language avatar
- In the first pass, the assembler generates a symbol code where it records and processes the pseudo instructions in assembly language. In the second pass, it generates the equivalent machine code.

## What are different types of compiler? Explain.

- A compiler is a software program that converts the high-level source code written in a programming language into low-level machine code that can be executed by the computer hardware.
- A Compilation Error occurs when the compiler fails to compile or convert the source code due to errors in the source code or in the compiler itself.
- There are many different types of Compilers. A few of them are mentioned below :

**Traditional Compiler :**

- Traditional Compilers simply convert a high-level language program code into its corresponding machine code.
- It Optimizes code for execution time and memory space efficiency.
- Examples of traditional compilers include compilers for languages like C, C++, and Pascal.

**Incremental Compiler :**

- Incremental compilers are designed to recompile only the modified portions of the source code, rather than recompiling the entire program.
- They analyze dependencies between source files to determine which parts of the code need to be recompiled based on changes.
- Incremental compilation reduces compilation time and accelerates development iterations, particularly in large codebases.

**JIT Compiler :**

- Just In Time Compilers or JIT Compilers are run-time compilers that help form executable code (machine code) from intermediate code (byte code).
- These compilers perform specific optimizations while compiling a series of bytecodes.
- They also implement type-based verification, which makes the machine code more authentic and optimized.

**Source to Source Compiler :**

- Source-to-source compilers are software tools designed to translate source code written in one programming language into executable code in another language, allowing developers to work with multiple languages seamlessly.

- It is a transcompiler. For example, a source-to-source compiler converts a C program source code to another source code like C++, Java, etc.
- It is majorly used for converting the old code of one programming language to its newer versions.

**Cross Compilers :**

- A cross-compiler creates executable code for a platform other than the one on which it is running.
- They enable developers to write and test code on one system while generating executable code for deployment on another system with different hardware or operating system requirements.
- Cross-compilers are commonly used in embedded systems development, cross-platform software development, and scenarios where the target platform lacks native development tools.
- Examples of cross-compilers include MinGW for compiling Windows executables on Linux.

**Single-pass Compiler :**

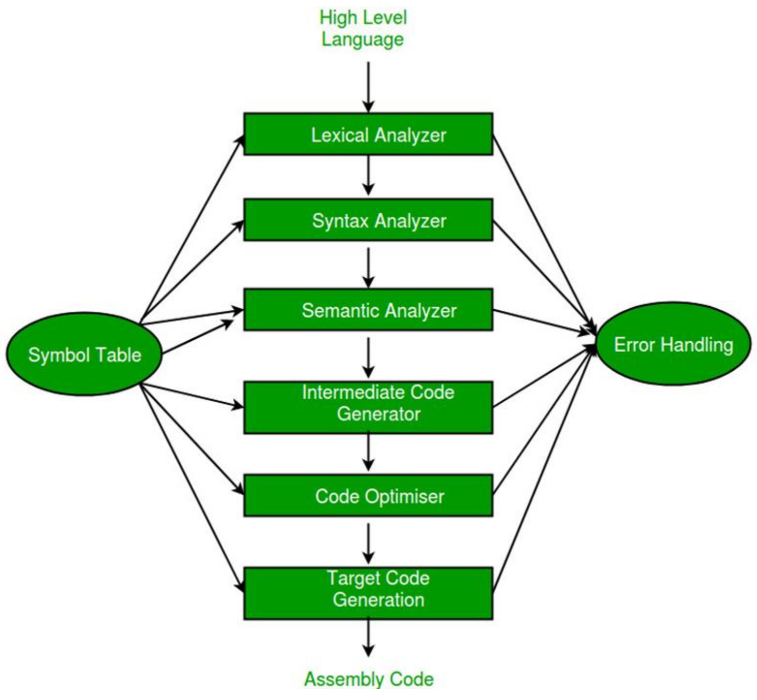
- A single pass compiler is characterized by having all the phases of the compiler contained within a single module, performing the conversion of source code to machine code in a single pass.
- It traverses the source code linearly, processing it from start to finish in one go without revisiting any code segments.
- Due to its sequential nature, a single pass compiler is generally more memory efficient and faster than multi-pass compilers, but it may lack advanced optimizations that require global analysis.

**Two Pass Compiler:**

- A two-pass compiler translates the program twice, once from the front end and then from the back end, hence known as a two-pass compiler.
- In the first pass, lexical and syntactic analysis are performed to create an intermediate representation of the program.
- The second pass involves semantic analysis, optimization, and code generation based on the intermediate representation produced in the first pass.

## What is pass in Compilers? What is meant by analysis & synthesis phase of a compiler?

- A pass refers to the traversal of a compiler take place through the entire program, then it is called a pass.
- Each pass typically performs a specific task or set of tasks on the input program, such as lexical analysis, syntax analysis, semantic analysis, optimization, or code generation.



**1. Analysis Phase:**

• In the Analysis Phase, the language processor takes the source program as

input and performs a comprehensive analysis of it.

• This phase involves several subtasks, including lexical analysis, syntax analysis,

and semantic analysis.

• **Lexical Analysis:** This is the initial step where the source code is broken down

into tokens or lexemes. Lexemes are the smallest units of meaning in a

programming language, such as keywords, identifiers, and literals. Lexical

analysis helps identify and categorize these tokens.

• **Syntax Analysis:** Once the tokens are identified, the language processor checks whether they follow the syntax rules of the programming language. This involves constructing a parse tree or syntax tree to represent the grammatical structure of the source code. If any syntax errors are found, they are reported.

• **Semantic Analysis:** Beyond syntax, this phase verifies the meaning of the program. It checks for semantic errors, such as type mismatches or undeclared variables. Additionally, it may perform optimizations and symbol table management.

Consider the following example:

**percent_profit = (profit * 100) / cost_price;**

• Lexical units identifies =, * and / operators, 100 as constant, and the remaining strings as identifiers.

• Syntax analysis identifies the statement as an assignment statement with percent_profit as the left hand side and (profit * 100) / cost_price as the expression on the right hand side.

• Semantic analysis determines the meaning of the statement to be the assignment of profit X 100 / cost_price to percent_profit.

**2. Synthesis Phase:**

• After successfully analyzing the source program, the language processor proceeds to the Synthesis Phase.

• In this phase, the processor generates the target code (e.g., machine code or intermediate code) based on the analysis performed in the previous phase.

• This phase includes tasks like code generation and code optimization.

• **Intermediate Code Generation:** Here, the language processor translates the high-level

source code into low-level target code. This can involve choosing appropriate machine instructions or generating intermediate code that will be further translated or executed.

• **Code Optimization**: Depending on the compiler or interpreter, this optional step involves improving the efficiency of the target code. It may include optimizations like dead code elimination, loop unrolling, or register allocation.

• **Code Generation:** The final phase of a compiler is code generation. This phase takes the optimized intermediate code and generates the actual machine code that can be executed by the target hardware.

## Explain translation of a statement using 6 phases of compiler

The translation of the high-level code to machine code is a complex process carried out by compilers in stages. The statement percent_profit = (profit * 100) / cost_price; will be translated through the following six phases of a typical compiler:

1. **Lexical Analysis**: This is the first phase where the compiler scans the code and breaks it down into tokens, which are the basic elements like keywords, identifiers, constants, operators, and separators. In the given statement, lexical analyzers will identify percent_profit, profit, and cost_price as identifiers, =, *, and / as operators, and 100 as a numeric constant.
2. **Syntax Analysis**: Once tokens are identified, the next phase is to arrange them in a structure that the compiler understands, using the grammar rules of the source language. This phase analyses the tokens and constructs a parse tree. It will recognize the given statement as an assignment statement where percent_profit is the left-hand side, and the right-hand side is an expression (profit * 100) / cost_price.
3. **Semantic Analysis**: In this phase, the compiler checks for semantic consistency. It ensures that operations are performed on compatible types, variables are declared before use, etc. The given statement is analyzed to ensure that the operation makes sense; i.e., the profit and cost_price should be of a type that supports multiplication and division by a constant (the number 100), and the result should be assignable to percent_profit.
4. **Intermediate Code Generation**: After semantic analysis, the compiler will convert the source code into an intermediate code that is independent of machine specifics. This abstract representation is a bridge between the source code and the machine code. For example, the statement might be translated into an intermediate code such as text

**t1 = profit * 100; t2 = t1 / cost_price; percent_profit = t2;**

where t1 and t2 are temporary variables used to hold intermediate values.

5. **Optimization**: This optional phase aims to improve the intermediate code so that it runs more efficiently (faster, using fewer resources) on the target machine. Optimization can be done on various levels, including the high-level (source code), the intermediate code, or the low-level (target machine

code). For instance, if profit is known to be always non-negative, the division might be optimized using shifts and adds instead of direct division.

6. **Code Generation**: This is the final phase where the compiler translates the (optimized) intermediate code into machine code, the binary instructions that the computer's CPU can execute directly. The machine code will include specific instructions to load the values of profit and cost_price into registers, multiply, divide, and store the result in the memory location associated with percent_profit.

7. **Code Optimization and Linking**: Although not traditionally enumerated as a separate phase, modern compilers often include additional optimization after generating the initial machine code, along with the linking of different code modules into a single executable. This can involve rearranging instructions for better pipeline usage on the CPU or resolving symbolic addresses to their actual memory locations.

Each phase takes input from its previous stage, processes it, and passes it on to the next stage. Any errors encountered along the way can cause compilation to fail, providing feedback for debugging.

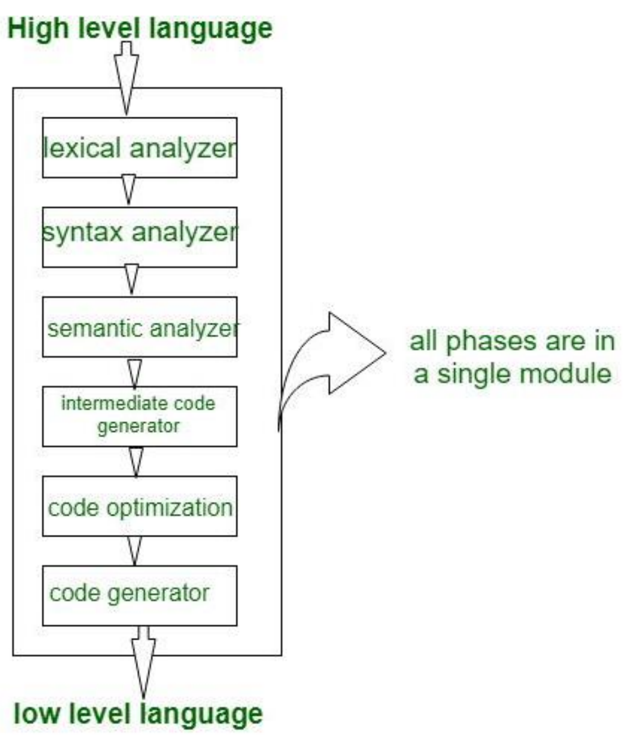
## What is difference between pass 1 & pass 2 compilers?

| Feature | One Pass Compiler | Two Pass Compiler |
|---|---|---|
| Translation Process | Performs translation in a single pass through the source code | Performs translation in two passes through the source code |
| Scanning | Scans the entire file only once during translation | Requires two passes to scan the entire source file |
| Intermediate Code Generation | Generates intermediate code | Does not generate intermediate code |
| Speed | Generally faster than a two-pass compiler | Generally slower than a one-pass compiler |
| Loader Requirement | No loader required as no object program is written | Requires a loader as object code is generated |
| Assembler Directives Processing | Processes assembler directives during the single pass | Processes assembler directives in the second pass |
| Data Structures | Utilizes symbol tables, literal tables, pool tables, and tables of incomplete data | Utilizes symbol tables and literal tables |
| Conversion Process | Performs the entire conversion of assembly code to machine code in one continuous operation | In the first pass, processes assembly code and stores values in the opcode table and symbol table; in the second pass, generates machine code using these tables |

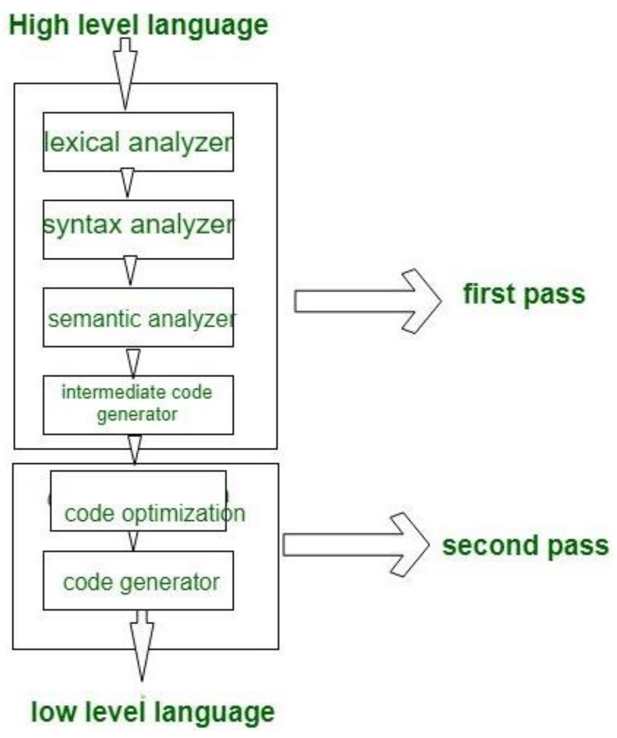
| Feature | One Pass Compiler | Two Pass Compiler |
|---|---|---|
| Examples | Compilers for languages like C and Pascal | Compilers for languages like Modula-2 |

FACETILDSS

**One pass**



**Two-Pass compiler**


High level language
lexical analyzer
syntax analyzer
semantic analyzer
intermediate code generator
first pass
code optimization
code generator
second pass
low level language