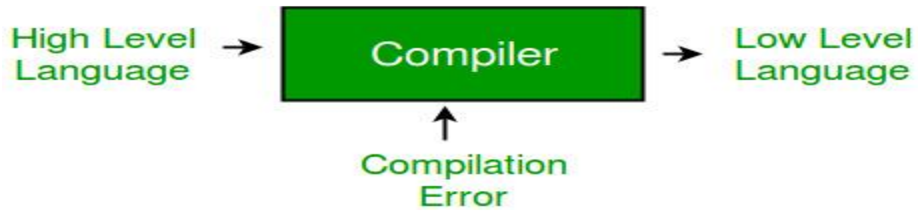
Unit-1

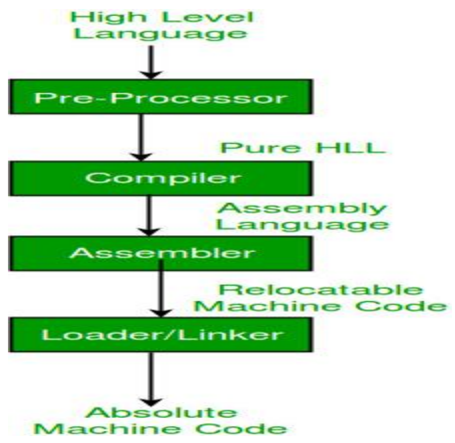
Compiler Design | Introduction of Compiler design

Compiler is a software which converts a program written in high level language (Source Language) to low level language (Object/Target/Machine Language).



- **Cross Compiler** that runs on a machine 'A' and produces a code for another machine 'B'. It is capable of creating code for a platform other than the one on which the compiler is running.
- **Source-to-source Compiler** or transcompiler or transpiler is a compiler that translates source code written in one programming language into source code of another programming language.

Language processing systems (using Compiler) –
We know a computer is a logical assembly of Software and Hardware. The hardware knows a language, that is hard for us to grasp, consequently we tend to write programs in high-level language, that is much less complicated for us to comprehend and maintain in thoughts. Now these programs go through a series of transformation so that they can readily be used machines. This is where language procedure systems come handy.

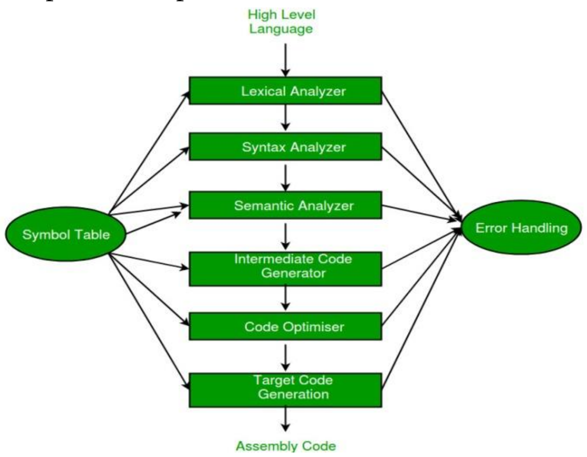


- **High Level Language** – If a program contains #define or #include directives such as #include or #define it is called HLL. They are closer to humans but far from machines. These (#) tags are called pre-processor directives. They direct the pre-processor about what to do.

- **Pre-Processor –** The pre-processor removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion. It performs file inclusion, augmentation, macro-processing etc.
- **Assembly Language –** Its neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.
- **Assembler –** For every platform (Hardware + OS) we will have a assembler. They are not universal since for each platform we have one. The output of assembler is called object file. Its translates assembly language to machine code.
- **Interpreter –** An interpreter converts high level language into low level machine language, just like a compiler. But they are different in the way they read the input. The Compiler in one go reads the inputs, does the processing and executes the source code whereas the interpreter does the same line by line. Compiler scans the entire program and translates it as a whole into machine code whereas an interpreter translates the program one statement at a time. Interpreted programs are usually slower with respect to compiled ones.
- **Relocatable Machine Code –** It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate for the program movement.
- **Loader/Linker –** It converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.

Phases of a Compiler –

There are two major phases of compilation, which in turn have many parts. Each of them take input from the output of the previous level and work in a coordinated way.



Analysis Phase – An intermediate representation is created from the give source code :

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator

Lexical analyzer divides the program into "tokens", Syntax analyzer recognizes "sentences" in the program using syntax of language and Semantic analyzer checks static semantics of each construct. Intermediate Code Generator generates "abstract" code.
Synthesis Phase – Equivalent target program is created from the intermediate representation. It has two parts :

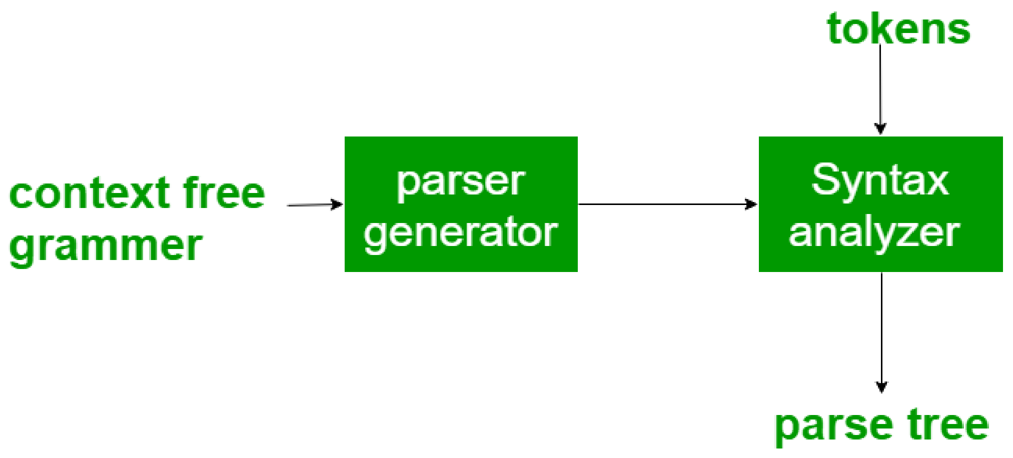
5. Code Optimizer
6. Code Generator

Code Optimizer optimizes the abstract code, and final Code Generator translates abstract intermediate code into specific machine instructions.

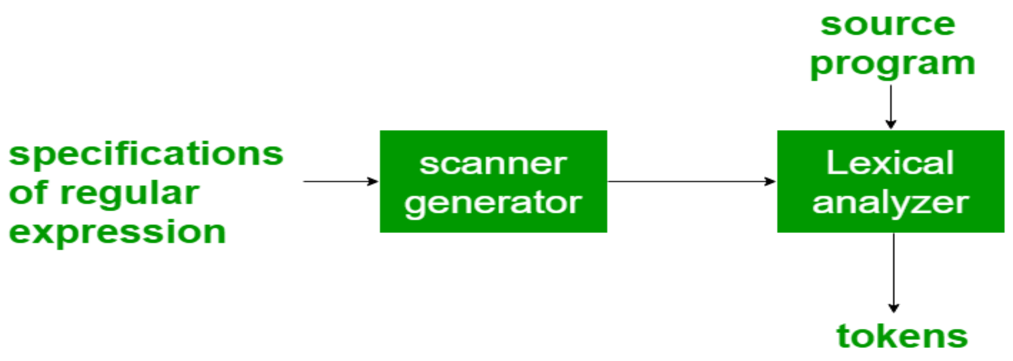
Compiler construction tools

The compiler writer can use some specialized tools that help in implementing various phases of a compiler. These tools assist in the creation of an entire compiler or its parts. Some commonly used compiler construction tools include:

1. **Parser Generator** –
   It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.
   Example:PIC, EQM



2. **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 automation to recognize the regular expression.
   Example: Lex

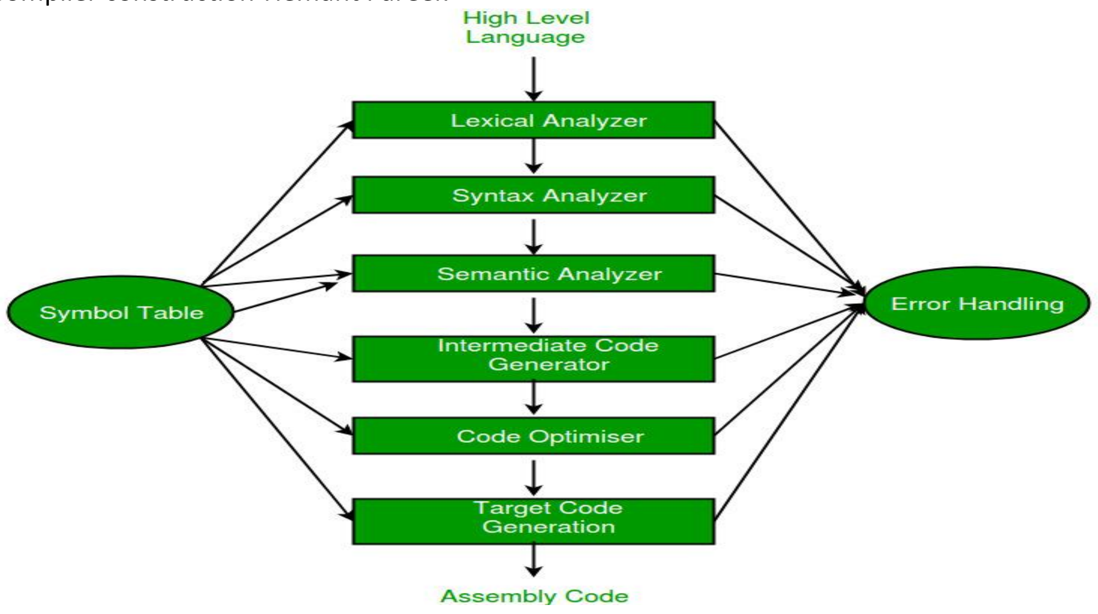


3. **Syntax directed translation engines** – It generates intermediate code with three address format from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produces the intermediate code. In this, each node of the parse tree is associated with one or more translations.
4. **Automatic code generators** – It generates the machine language for a target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator. Template matching process is used. An intermediate language statement is replaced by its equivalent machine language statement using templates.
5. **Data-flow analysis engines** – It is used in code optimization.Data flow analysis is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another. Refer – data flow analysis in Compiler
6. **Compiler construction toolkits** – It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.

Compiler Design | Phases of a Compiler

Prerequisite – Introduction of Compiler design

We basically have two phases of compilers, namely Analysis phase and Synthesis phase. Analysis phase creates an intermediate representation from the given source code. Synthesis phase creates an equivalent target program from the intermediate representation.



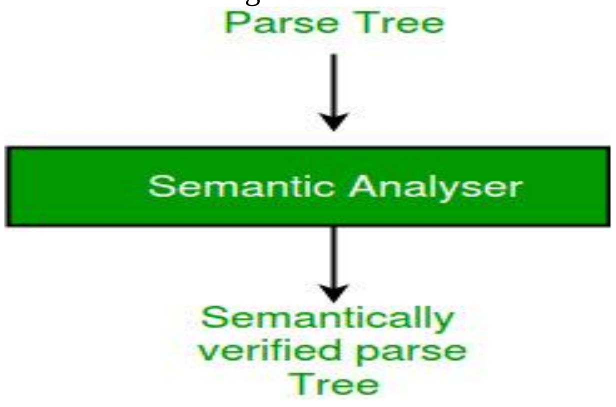
Symbol Table – It is a data structure being used and maintained by the compiler, consists all the identifier's name along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

The compiler has two modules namely front end and back end. Front-end constitutes of the Lexical analyzer, semantic analyzer, syntax analyzer and intermediate code generator. And the rest are assembled to form the back end.

1. **Lexical Analyzer** – It reads the program and converts it into tokens. It converts a stream of lexemes into a stream of tokens. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes white-spaces and comments.
2. **Syntax Analyzer** – It is sometimes called as parser. It constructs the parse tree. It takes all the tokens one by one and uses Context Free Grammar to construct the parse tree.

   Why Grammar ?
   The rules of programming can be entirely represented in some few productions. Using these productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.

   Syntax error can be detected at this level if the input is not in accordance with the grammar.



3. **Semantic Analyzer** – It verifies the parse tree, whether it's meaningful or not. It furthermore produces a verified parse tree.It also does type checking, Label checking and Flow control checking.
4. **Intermediate Code Generator** – It generates intermediate code, that is a form which can be readily executed by machine We have many popular intermediate codes. Example – Three address code etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.

   Till intermediate code, it is same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don't need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

5. **Code Optimizer** – It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimisation can be categorized into two types: machine dependent and machine independent.
6. **Target Code Generator** – The main purpose of Target Code generator is to write a code that the machine can understand and also register allocation, instruction selection etc. The output is dependent on the type of assembler. This is the final stage of compilation.

Symbol Table in Compiler

Prerequisite – Phases of a Compiler

Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variable i.e. it stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

- It is built in lexical and syntax analysis phases.
- The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code.
- It is used by compiler to achieve compile time efficiency.
- It is used by various phases of compiler as follows :-
  1. **Lexical Analysis:** Creates new table entries in the table, example like entries about token.
  2. **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
  3. **Semantic Analysis:** Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.
  4. **Intermediate Code generation:** Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
  5. **Code Optimization:** Uses information present in symbol table for machine dependent optimization.
  6. **Target Code generation:** Generates code by using address information of identifier present in the table.

Symbol Table entries – Each entry in symbol table is associated with attributes that support compiler in different phases.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Information used by compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

Operations of Symbol table – The basic operations defined on a symbol table include:

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

Implementation of Symbol table –
Following are commonly used data structure for implementing symbol table :-

1. **List** –
   - In this method, an array is used to store names and associated information.
   - A pointer **"available"** is maintained at end of all stored records and new names are added in the order as they arrive
   - To search for a name we start from beginning of list till available pointer and if not found we get an error **"use of undeclared name"**
   - While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. **"Multiple defined name"**
   - Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
   - Advantage is that it takes minimum amount of space.
2. **Linked List** –
   - This implementation is using linked list. A link field is added to each record.

- Searching of names is done in order pointed by link of link field.
- A pointer **"First"** is maintained to point to first record of symbol table.
- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

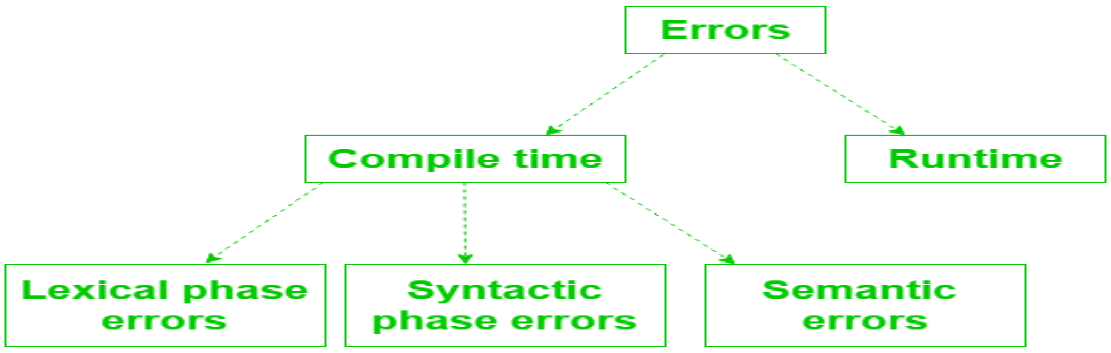
3. **Hash Table** –
    - In hashing scheme two tables are maintained – a hash table and symbol table and is the most commonly used method to implement symbol tables..
    - A hash table is an array with index range: 0 to tablesize – 1.These entries are pointer pointing to names of symbol table.
    - To search for a name we use hash function that will result in any integer between 0 to tablesize – 1.
    - Insertion and lookup can be made very fast – O(1).
    - Advantage is quick search is possible and disadvantage is that hashing is complicated to implement.
4. **Binary Search Tree** –
    - Another approach to implement symbol table is to use binary search tree i.e. we add two link fields i.e. left and right child.
    - All names are created as child of root node that always follow the property of binary search tree.
    - Insertion and lookup are O($\log_2$ n) on average.

Error detection and Recovery in Compiler

In this phase of compilation, all possible errors made by the user are detected and reported to the user in form of error messages. This process of locating errors and reporting it to user is called **Error Handling process**.
Functions of Error handler

- Detection
- Reporting
- Recovery

Classification of Errors



Compile time errors are of three types:-

Lexical phase errors

These errors are detected during the lexical analysis phase. Typical lexical errors are

- Exceeding length of identifier or numeric constants.
- Appearance of illegal characters
- Unmatched string

Error recovery:
Panic Mode Recovery

- In this method, successive characters from the input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as; or }
- Advantage is that it is easy to implement and guarantees not to go to infinite loop
- Disadvantage is that a considerable amount of input is skipped without checking it for additional errors

Syntactic phase errors

These errors are detected during syntax analysis phase. Typical syntax errors are

- Errors in structure
- Missing operator
- Misspelled keywords
- Unbalanced parenthesis

```
Example : swicth(ch)
                {
                   .......
                   .......
                }
```

The keyword **switch** is incorrectly written as swicth. Hence, **“Unidentified keyword/identifier”** error occurs.

Error recovery:

1. **Panic Mode Recovery**
    - In this method, successive characters from input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are deli-meters such as ; or }
    - Advantage is that its easy to implement and guarantees not to go to infinite loop
    - Disadvantage is that a considerable amount of input is skipped without checking it for additional errors
2. **Statement Mode recovery**
    - In this method, when a parser encounters an error, it performs necessary correction on remaining input so that the rest of input statement allow the parser to parse ahead.

- The correction can be deletion of extra semicolons, replacing comma by semicolon or inserting missing semicolon.
- While performing correction, atmost care should be taken for not going in infinite loop.
- Disadvantage is that it finds difficult to handle situations where actual error occured before point of detection.

3. **Error production**
   - If user has knowledge of common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.
   - If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
   - Disadvantage is that its difficult to maintain.
4. **Global Correction**
   - The parser examines the whole program and tries to find out the closest match for it which is error free.
   - The closest match program has less number of insertions, deletions and changes of tokens to recover from erroneous input.
   - Due to high time and space complexity, this method is not implemented practically.

Semantic errors

These errors are detected during semantic analysis phase. Typical semantic errors are

- Incompatible type of operands
- Undeclared variables
- Not matching of actual arguments with formal one

```
Example : int a[10], b;
                .......
                .......
                a = b;
```

It generates a semantic error because of an incompatible type of a and b.

Error recovery

- If error **“Undeclared Identifier”** is encountered then, to recover from this a symbol table entry for corresponding identifier is made.
- If data types of two operands are incompatible then, automatic type conversion is done by the compiler.

Error Handling in Compiler Design

The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recover strategy and implement them to handle error. During this whole process processing time of program should not be slow. An **Error** is the blank entries in the symbol table.

Types or Sources of Error – There are two types of error: run-time and compile-time error:

1. A **run-time error** is an error which takes place during the execution of a program, and usually happens because of adverse system parameters or invalid input data. The lack of sufficient memory to run an application or a memory conflict with another program and logical error are example of this. Logic errors, occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.
2. **Compile-time errors** rises at compile time, before execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is the example of this.

Classification of Compile-time error –

1. **Lexical** : This includes misspellings of identifiers, keywords or operators
2. **Syntactical** : missing semicolon or unbalanced parenthesis
3. **Semantical** : incompatible value assignment or type mismatches between operator and operand
4. **Logical** : code not reachable, infinite loop.

Finding error or reporting an error – Viable-prefix is the property of a parser which allows early detection of syntax errors.

- **Goal:** detection of an error as soon as possible without further consuming unnecessary input
- **How:** detect an error as soon as the prefix of the input does not match a prefix of any string in the
  language.
- **Example:** for(**;**), this will report an error as for have two semicolons inside braces.

Error Recovery –
The basic requirement for the compiler is to simply stop and issue a message, and cease compilation. There are some common recovery methods that are follows.

1. **Panic mode recovery:** This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops while recovering error. The parser discards the input symbol one at a time until one of the designated (like end, semicolon) set of synchronizing tokens (are typically the statement or expression terminators) is found. This is adequate when the presence of multiple errors in same statement is rare. Example: Consider the erroneous expression- (1 + + 2) + 3. Panic-mode recovery: Skip ahead to next integer and then continue. Bison: use the special terminal **error** to describe how much input to skip.

```
E->int|E+E|(E)|error int|(error)
```

2. **Phase level recovery:** Perform local correction on the input to repair the error. But error correction is difficult in this strategy.
3. **Error productions:** Some common errors are known to the compiler designers that may occur in the code. Augmented grammars can also be used, as productions that generate erroneous constructs when these errors are encountered. Example: write 5x instead of 5*x
4. **Global correction:** Its aim is to make as few changes as possible while converting an incorrect input string to a valid string. This strategy is costly to implement.

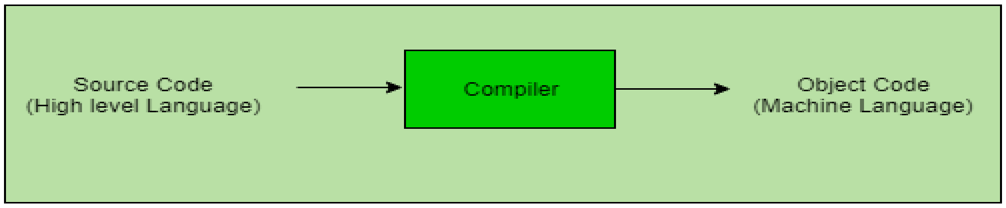
Language Processors: Assembler, Compiler and Interpreter

Language Processors –
Assembly language is machine dependent yet mnemonics that are being used to represent instructions in it are not directly understandable by machine and high Level language is machine independent. A computer understands instructions in machine code, i.e. in the form of 0s and 1s. It is a tedious task to write a computer program directly in machine code. The programs are written mostly in high level languages like Java, C++, Python etc. and are called **source code**. These source code cannot be executed directly by the computer and must be converted into machine language to be executed. Hence, a special translator system software is used to translate the program written in high-level language into machine code is called **Language Processor** and the program after translated into machine code (object program / object code).

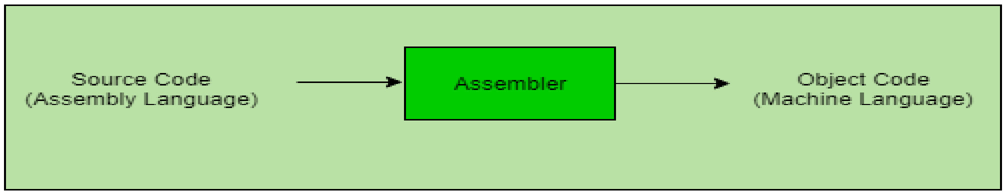
The language processors can be any of the following three types:

1. **Compiler** –
   The language processor that reads the complete source program written in high level language as a whole in one go and translates it into an equivalent program in machine language is called as a Compiler.
   Example: C, C++, C#, Java

   In a compiler, the source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again.>



2. **Assembler** –
   The Assembler is used to translate the program written in Assembly language into machine code. The source program is a input of assembler that contains assembly language instructions. The output generated by assembler is the object code or machine code understandable by the computer.



3. **Interpreter –**
   The translation of single statement of source program into machine code is done by language processor and executes it immediately before moving on to the next line is called an interpreter. If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message. The interpreter moves on to the next line for execution only after removal of the error. An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code.
   Example: Perl, Python and Matlab.

Difference between Compiler and Interpreter –

Compiler	**Interpreter**
A compiler is a program which coverts the entire source code of a programming language into executable machine code for a CPU.	interpreter takes a source program and runs it line by line, translating each line as it comes to it.
Compiler takes large amount of time to analyze the entire source code but the overall execution time of the program is comparatively faster.	Interpreter takes less amount of time to analyze the source code but the overall execution time of the program is slower.
Compiler generates the error message only after scanning the whole program, so debugging is comparatively hard as the error can be present any where in the program.	Its Debugging is easier as it continues translating the program until the error is met
Generates intermediate object code.	No intermediate object code is generated.
Examples: C, C++, Java	Examples: Python, Perl

Generation of Programming Languages

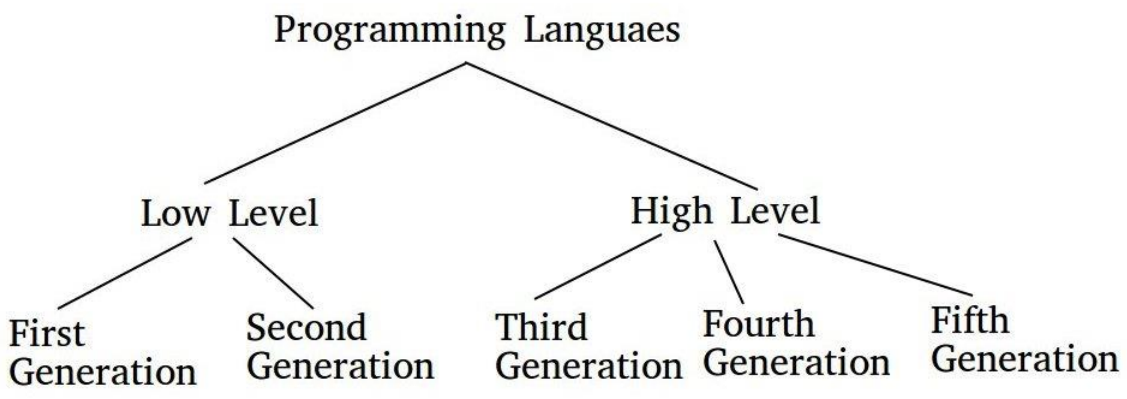
There are five generation of Programming languages.They are:
First Generation Languages :
These are low-level languages like machine language.
Second Generation Languages :
These are low-level assembly languages used in kernels and hardware drives.
Third Generation Languages :
These are high-level languages like C, C++, Java, Visual Basic and JavaScript.
Fourth Generation Languages :
These are languages that consist of statements that are similar to statements in the human language. These are used mainly in database programming and scripting. Example of these

languages include Perl, Python, Ruby, SQL, MatLab(MatrixLaboratory).

Fifth Generation Languages :

These are the programming languages that have visual tools to develop a program. Examples of fifth generation language include Mercury, OPS5, and Prolog.

The first two generations are called low level languages. The next three generations are called high level languages.


Programming Languaes
Low Level
High Level
First Generation
Second Generation
Third Generation
Fourth Generation
Fifth Generation