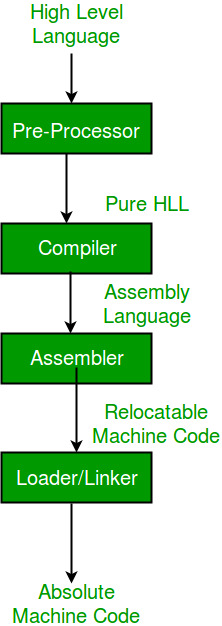
| S.No. | Compiler | Interpreter |
| --- | --- | --- |
| 1. | The compiler scans the whole program in one go. | Translates the program one statement at a time. |
| 2. | As it scans the code in one go, the errors (if any) are shown at the end together. | Considering it scans code one line at a time, errors are shown line by line. |
| 3. | The main advantage of compilers is its execution time. | Due to interpreters being slow in executing the object code, it is preferred less. |

|  |  |  |
| --- | --- | --- |
| Eg. | C, C++, C#, etc are programming languages that are compiler-based. | Python, Ruby, Perl, SNOBOL, MATLAB, etc are programming languages that are interpreter-based. |

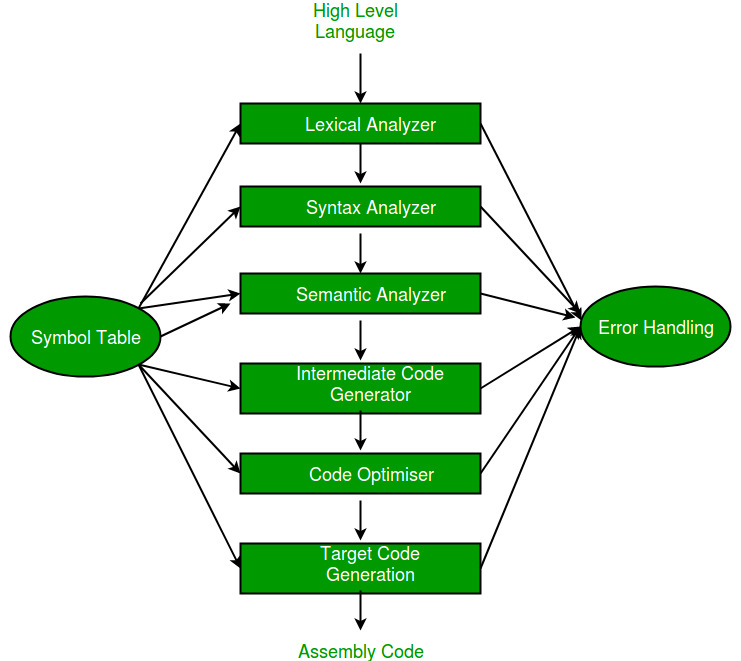
The**compiler**is software that converts a program written in a high-level language (Source Language) to a low-level language (Object/Target/Machine Language/0’s, 1’s)



**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

**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 with 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).



**Symbol Table –** It is a data structure being used and maintained by the compiler, consisting of all the identifier’s names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

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

1. [**Lexical Analyzer**](https://www.geeksforgeeks.org/compiler-lexical-analysis/)**–**   
   It is also called a scanner. It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language. It reads the characters from the source program and groups them into lexemes (sequence of characters that “go together”). Each lexeme corresponds to a token. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors (e.g., erroneous characters), comments, and white space.

[**Syntax Analyzer**](https://www.geeksforgeeks.org/compiler-design-introduction-to-syntax-analysis/)**–** It is sometimes called a parser. It constructs the parse tree. It takes all the tokens one by one and uses Context-Free Grammar to construct the parse tree.

* **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.
* [**Intermediate Code Generator**](https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/)**–** It generates intermediate code, which is a form that can be readily executed by a machine We have many popular intermediate codes. Example – Three address codes etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.

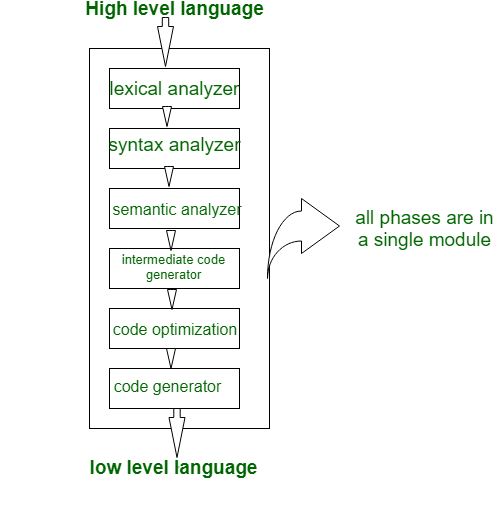
Till intermediate code, it is the same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don’t need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

[**Code Optimizer**](https://www.geeksforgeeks.org/compiler-design-code-optimization/)**–** It transforms the code so that it consumes fewer resources and produces more speed.

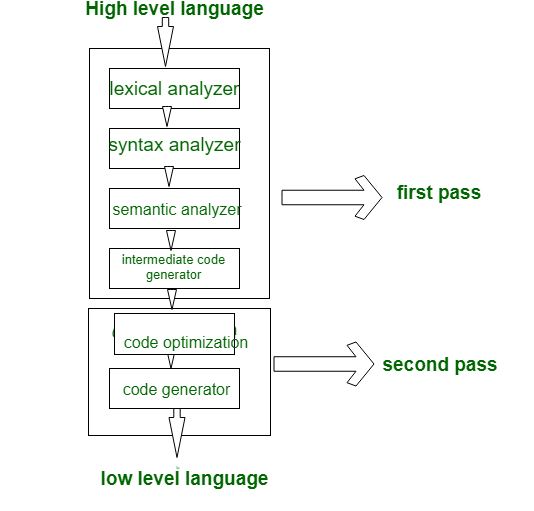
**Target Code Generator –** The main purpose of the Target Code generator is to write a code that the machine can understand. The output is dependent on the type of assembler. This is the final stage of compilation.

A **Compiler pass** refers to the traversal of a compiler through the entire program. Compiler pass are two types: Single Pass Compiler, and Two Pass Compiler *or* Multi Pass Compiler.

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



**2. Two Pass compiler *or* Multi Pass compiler:**  
A Two pass/multi-pass Compiler is a type of compiler that processes the *source code* or abstract syntax tree of a program multiple times. In multipass Compiler we divide phases in two pass as:



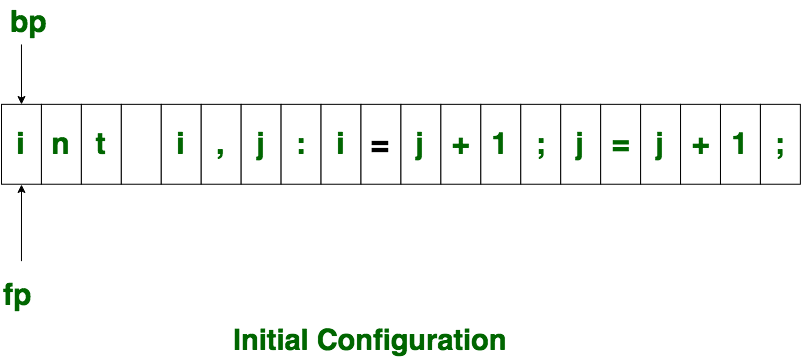
1. **First Pass:** is refers as
   * **(a).** Front end
   * **(b).** Analytic part
   * **(c).** Platform independent
   * In first pass the included phases are as Lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generator are work as front end and analytic part means all phases analyze the High level language and convert them **three address code**

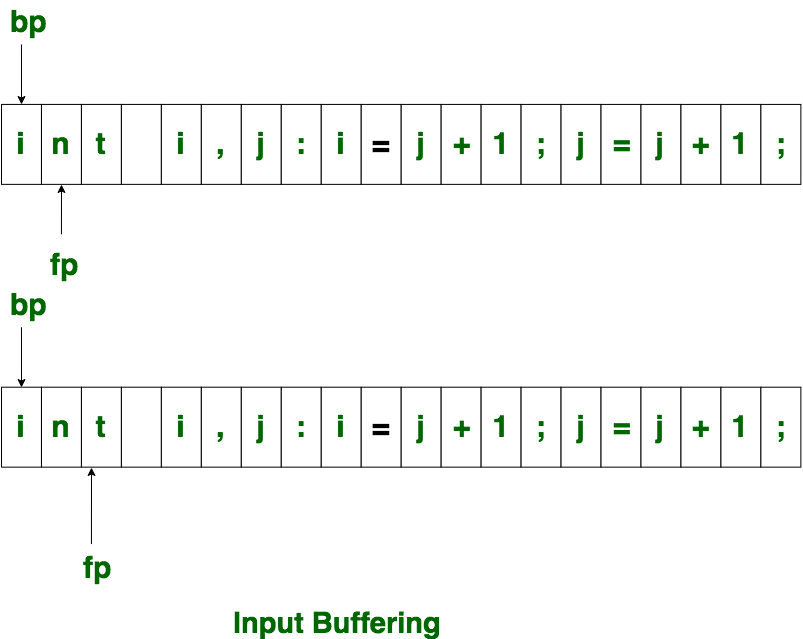
**Second Pass:** is refers as

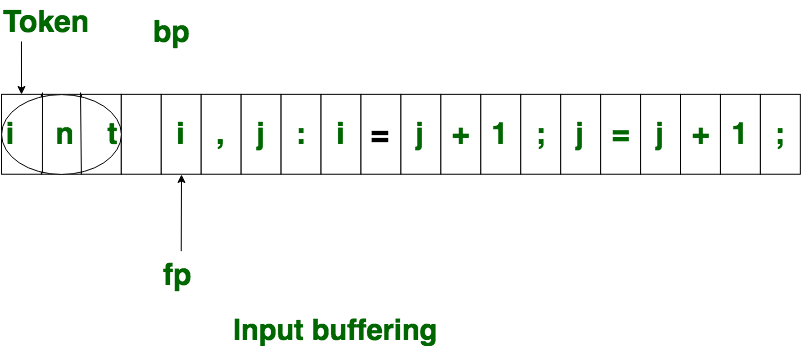
* **(a).** Back end
* **(b).** Synthesis Part / Optimiser Part
* **(c).** Platform Dependent
* In second Pass the included phases are as Code optimization and Code generator are work as back end and the synthesis part refers to taking input as three address code and convert them into Low level language/assembly language

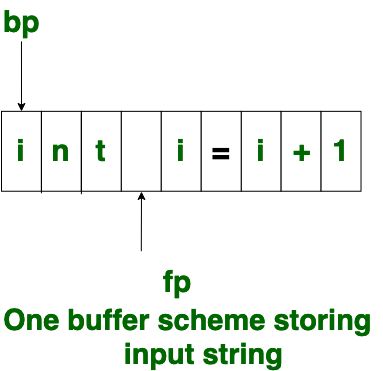
**INPUT BUFFERING**

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(**bp**) and forward ptr(**fp**).



Initially both the pointers point to the first character of the input string as shown below 

The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In above example as soon as ptr (fp) encounters a blank space the lexeme “int” is identified. The fp will be moved ahead at white space, when fp encounters white space, it ignore and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token. The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. hence buffering technique is used.A block of data is first read into a buffer, and then second by lexical analyzer. there are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme. These are explained as following below. 

1. **One Buffer Scheme:** In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme. 
2. **Two Buffer Scheme:** To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. the first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled.

to identify, the boundary of first buffer end of buffer character should be placed at the end first buffer. Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. when fp encounters first **eof**, then one can recognize end of first buffer and hence filling up second buffer is started. in the same way when second **eof** is obtained then it indicates of second buffer. This **eof** character introduced at the end is calling **Sentinel** which is used to identify the end of buffer.

**Types of Parser:**

The parser is mainly classified into two categories, i.e. Top-down Parser, and Bottom-up Parser. These are explained below:

**Top-Down Parser:**

The [top-down parser](https://www.geeksforgeeks.org/classification-of-top-down-parsers/) is the parser that **generates parse tree for the given input string**with the help of production rules by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.   
Further Top-down parser is classified into 2 types: A recursive descent parser, and Non-recursive descent parser.

1. [**Recursive descent parser**](https://www.geeksforgeeks.org/compiler-design-recursive-descent-parser/)is also known as the Brute force parser or the backtracking parser. It basically generates the parse tree by using brute force and backtracking.
2. **Non-recursive descent parser** is also known as LL(1) parser or predictive parser or without backtracking parser. It uses a parsing table to generate the parse tree instead of backtracking.

Panic mode recovery: Discard tokens one at a time until a predefined synchronizing token is found. Synchronizing tokens are usually delimiters, like ; and }, whose role in the source language is clear and unambiguous.

Phrase-level recovery: On detecting an error, the parser performs local correction on the remaining input so that parsing can be continued. For example – replace a comma by a semicolon, delete a semicolon, insert a semicolon, etc. This may lead to an infinite loop.

**Runtime Environment**

We can represent the activation of functions during the execution of a program by a tree, called an activation tree.

Each node represents one activation of one function.

The root represents the activation of the main function.

The children nodes represent the functions called by the function, in the given order.

Function calls and returns are handled using the control stack.

Each live activation of a function has an activation record, also called frame.

The activation record of the main function is at the bottom of the control stack.

Actual Parameters, returned values, control link, local data, temporaries etc.