Intro:

- Human understandable and Machine understandable languages are different.
- Humans communicate with machines, by giving instructions and data as the input to a machine.
- These instructions are written in a human understandable language, known as programming language / high-level language.
- The instructions written in a high-level programming language, can be executed by a machine by first translating it to a machine understandable language, known as binary / Assembly / low-level language.
- Hence the program which translates the instructions written in high-level programming language to low-level language, is known as a Compiler.

(or)

- A Compiler is a program that converts a source program to a target program.
- Here the source program is the human written instructions in a high-level programming language, and the target program is the machine level code in binary or assembly language.

- The target program is written in a file which can be opened and directly executed on a machine, known as an executable file.
- Every programming language has its own compiler.

Phases of a Compiler:

- A Compiler goes through (6) phases, in-order to generate a file with the executable code on it.

1. Lexical Analyzer: (Scanner)

- The program/script written in high level language is converted to string format and provided as input to the respective compiler.
- The string/program is first given to the lexical analyzer as the input.
- Lexical Analyzer (also known as Lexer/Scanner) converts the strings into tokens, which are sequences of characters with a valid pattern.
- A Token can be considered as a class of Lexemes.
- A Lexeme can be considered as an instance of Token.
- A Pattern is considered as a rule (or) set of rules which is used to validate a token.
- Hence, the input to the lexical analyzer is program string and the output is tokens.

Ex:

1. Token: Identifier
   Pattern: A sequence of letters and digits, starting with a letter.
   Lexemes: variableName, sum, x
2. Token: Integer Literals
   Pattern: A sequence of digits.
   Lexemes: 123, 456, 7890
3. Token: Keyword
   Pattern: Specific reserved words in a programming language.
   Lexemes: if, else, while, return

4. Token: Operator (RELOP (relational operators) + Others)
   Pattern: Specific symbols or combinations of symbols.
   Lexemes: +, -, *, /, <, >, <=, >=, ==, !=
5. Token: String Literals
   Pattern: A sequence of characters enclosed in double quotes.
   Lexemes: "hello", "world", "test123"

- The way to pass the generated tokens to the next phase/component (parser) is by storing them into the Symbol Table first.
- Symbol Table is a Data Structure used in compilers and interpreters (particularly during the lexical and semantic analysis phases) to store information about identifiers (symbols) from the source code, such as variables, operators, function names, objects, etc.
- So instead of passing the tokens directly, Lexical Analyzer passes the references to those tokens in the symbol table to the parser and the format is:

< id, Pointer to symbol table for token >

2. Syntax Analyzer: (Parser)

- As the name suggests, It takes the tokens provided by lexical analyzer from the symbol table and verifies whether the tokens are in correct syntax or not, according to the Grammar of the programming language.
- As the output, the Syntax Analyzer generates a parse tree. That is, the syntax is checked by generating a parse tree for each token.
- A parse tree is a hierarchical structure of a lexeme built by referencing the grammar rules. (production rules)
- If a parse tree for a token is fully constructed, it means that the syntax is correct (token satisfies the grammar), else the Syntax analyzer informs about the syntax error to error handler, with line number and position at which the syntax is mismatched.
- The parsing methods are of two types, based on the direction of parsing:

a. Top-Down Parsing
b. Bottom-Up Parsing

3. Semantic Analyzer:

- The syntactically verified parse tree generated by the parser is given to the semantic analyzer as the input.
- The Semantic analyzer does *Type Checking*, *Scope Resolution*, *Name Binding*, and *Symbol Table Binding*.
- Type Checking: Ensures that operations are performed on compatible types.
  Ex: To perform multiplication operations on two numbers, it has to be ensured that the data types of two numbers are the same. In this case the semantic analyzer performs implicit type conversion
- Scope Resolution: Ensures that variables and functions are declared before they are used. Maintains Scope information to handle local and global variables correctly.
- Name Binding: Associates variable and function references with their declarations.
- Symbol Table Binding: Extends the symbol table created during lexical analysis with more detailed information, including type information and scope details.

4. Intermediate Code Generation:

- Intermediate code generation translates high-level source code into a platform-independent intermediate representation.
- The primary purpose of this phase is to translate the high-level source code into an intermediate representation (IR) that is easier for subsequent phases of the compiler to analyze and optimize.
- This intermediate code is typically more abstract than machine code but lower-level than the source language.
- Intermediate Code can be in three forms:

(a). Postfix
(b). Triples (three address code)
(c). Quadruples

- The intermediate code should have the following properties:
  - ➔ Each instruction can have only one arithmetic operator and one assignment operation.
  - ➔ For **Three Address Code**, each instruction must have less than **three** operands and so on.

Ex:

Input Source Code =>

a = b + c * d;

Intermediate Code Representation of Input Source Code =>

t1 = c * d
t2 = b + t1
a = t2

5. Code Optimizer:

- A code optimizer is a phase in the compiler design process that aims to improve the intermediate code generated during the intermediate code generation phase.
- The goal is to enhance performance and efficiency by reducing the number of instructions, improving resource utilization, and minimizing execution time and memory usage
- Key objectives of code optimization:

(a). Speed Up Execution: Reducing the number of instructions and the execution time.
(b). Reduce Memory Usage: Minimizing the amount of memory required by the program.
(c). Improve Resource Utilization: Making better use of CPU registers and cache.
(d). Power Efficiency: Reducing the power consumption, which is crucial for mobile and embedded systems.

- Commonly used code optimization techniques are:

1. Constant Folding:
   - Compute constant expressions at compile time.
   - Example: 3 + 5 is replaced with 8.
2. Strength Reduction:
   - Replace expensive operations with equivalent, less expensive ones.
   - Example: Replace x * 2 with x + x.
3. Dead Code Elimination:
   - Remove code that does not affect the program's output.
   - Example: Remove a variable assignment that is never used later.
4. Common Subexpression Elimination:
   - Identify and eliminate expressions that are computed multiple times.
   - Example: Replace multiple instances of a + b with a single computation stored in a temporary variable.

6. Code Generator:

- The code generator is a phase in the compiler design process responsible for translating intermediate code into target machine code or assembly code.
- This phase comes after code optimization and is crucial for producing an executable program that can run on a specific hardware platform.
- Key tasks of the Code Generator are:
  a. Instruction Selection: Convert intermediate code operations into the target machine's instructions.
  b. Register Allocation: Assign variables to CPU registers.
  c. Memory Management: Handle the allocation of memory for variables and ensure correct address computation.
- Ex:

Intermediate Code:

```
t1 = c * d
t2 = b + t1
a = t2
```

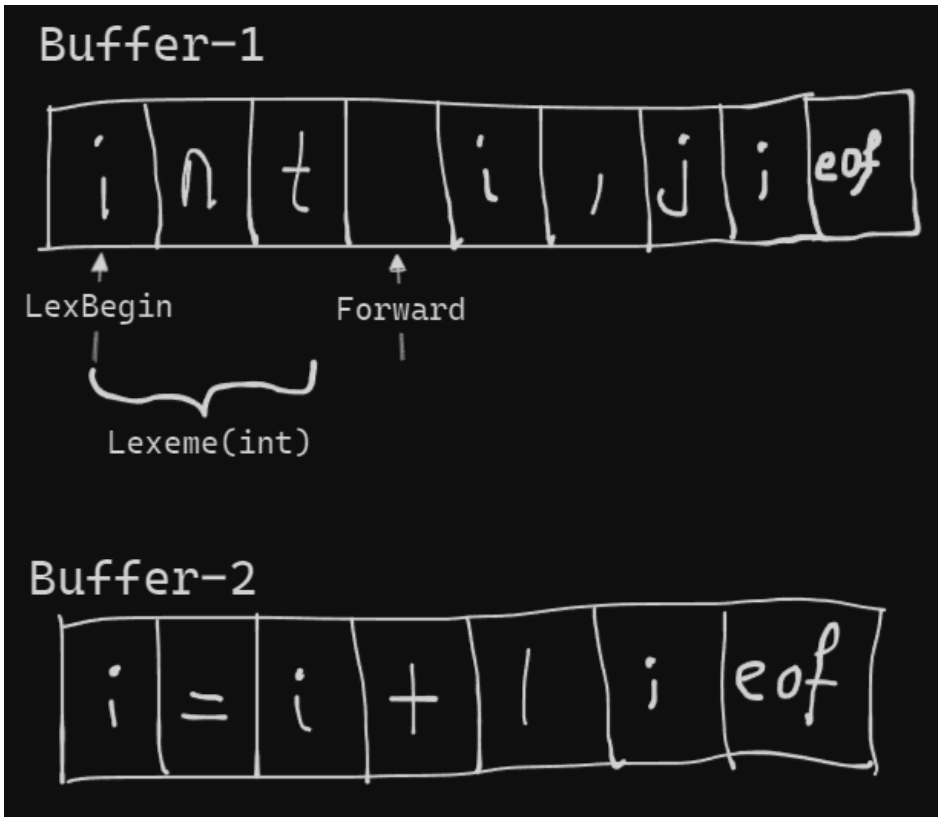
Generated Assembly code:

```
MOV EAX, [c]       ; Load value of c into EAX
IMUL EAX, [d]      ; Multiply EAX by the value of d, result in EAX
MOV EBX, [b]       ; Load value of b into EBX
ADD EBX, EAX       ; Add EAX to EBX, result in EBX
MOV [a], EBX       ; Store the result from EBX into a
```

Input Buffering:

- During lexical analysis, the source code is read character by character from the secondary storage, to form tokens.
- Each character would have to be read directly from the source file repeatedly, which can be slow and inefficient because large no.of I/O operations are performed.
- Input buffering optimizes this process by reducing the number of I/O operations.
- In Input Buffering:
    1. Two fixed sized memory blocks (known as Buffers) are allocated in the RAM.
    2. The Buffer-1 is filled with some code from the source file in secondary storage.
    3. The Lexical Analyser uses two pointers, viz **LexBegin** and **Forward** on the Buffer to read the tokens from it. Initially, **LexBegin** and **Forward** pointers are placed at first positions of a buffer.
    4. The **Forward** Pointer is sequentially moved forward to the next block until the empty block (space) is read.
    5. The space character is ignored and the sequence of characters b/w **LexBegin** and **Forward** Pointers are considered as a single lexeme.
    6. Then the lexeme is processed to generate the corresponding Token. Parallelly/Concurrently, the Buffer-2 is filled with the next block of code from the source file present in secondary storage.
        - Since, A Single Buffer can have multiple lexemes, a special character(Sentinel Character) is specified at the next position of last lexeme in the buffer to indicate the end of the buffer, known as EOF (End-Of-FIle)
    7. After the EOF is read, the two pointers are shifted to the starting position of Buffer-2, to read the next set of lexemes.

Ex:


Buffer-1
i n t i , j i eof
LexBegin
Forward
Lexeme(int)
Buffer-2
i = i + 1 i eof