

## OVERVIEW OF COMPILATION

=====

What is Compilation?

-----

Compilation is the process of converting a **high-level programming language** (such as C, C++, Java) into **machine-level language** (0s and 1s) so that a computer can understand and execute it.

### Real-Life Example:

A teacher speaks **English**, but a student understands **Hindi**. A **translator** converts English into Hindi. Similarly, a **compiler** translates human-readable code into machine-readable code.

Compiler vs Interpreter

-----

Compiler	Interpreter	Translates the whole program at once	Translates line by line	Shows all errors after compilation	Stops at first error	Execution is fast	Execution is slower	Example: C, C++	Example: Python, JavaScript
----------	-------------	--------------------------------------	-------------------------	------------------------------------	----------------------	-------------------	---------------------	-----------------	-----------------------------

## STRUCTURE OF A COMPILER

=====

A compiler works in **six major phases**:

Source Program → Lexical Analysis → Syntax Analysis → Semantic Analysis → Intermediate Code Generation → Code Optimization → Target Code Generation`

## 1\ Lexical Analysis

-----

- \* Converts characters into tokens
- \* Removes white spaces and comments

Example:

int a = 10;`

Tokens:

int | a | = | 10 | ;`

## 2\ Syntax Analysis

-----

- \* Checks grammar and structure
- \* Detects syntax errors

✗ int a = ; → Syntax error

### 3\ Semantic Analysis

---

- \* Checks meaning of statements

- \* Ensures type compatibility

✗ `int a = "hello";`

### 4\ Intermediate Code Generation

---

- \* Produces machine-independent code

Example:

`t1 = b + c   a = t1`

### 5\ Code Optimization

---

- \* Improves performance

- \* Reduces unnecessary operations

Example:

```
x = 2 * 4; `
```

Optimized:

```
x = 8; `
```

## 6\ Target Code Generation

-----

\* Produces machine-level code

## APPLICATIONS OF COMPILER TECHNOLOGY

=====

Compiler techniques are used beyond programming languages.

### Applications:

1. Programming Language Translators

2. Database Query Optimization (SQL)

3. Operating Systems

4. Artificial Intelligence

5. Web Development (JavaScript Transpilers)

### Real-Life Example:

Google Translate uses compiler principles such as **lexical analysis** and **parsing**.

## LEXICAL ANALYSIS

=====

What is a Lexical Analyzer?

-----

A lexical analyzer is the **first phase of the compiler**. It scans the source code and converts it into **tokens**.

### Example:

Input:

sum = a + b; `

Output Tokens:

` IDENTIFIER | OPERATOR | IDENTIFIER | OPERATOR | IDENTIFIER | ; `

## Role of a Lexical Analyzer

-----

1. Token generation
2. Removing white spaces and comments
3. Creating symbol table entries
4. Detecting lexical errors
5. Communicating with the syntax analyzer

### ### Real-Life Example:

When reading a sentence, we recognize **words**, not individual letters. Similarly, the lexical analyzer recognizes **tokens**, not characters.

## SPECIFICATION OF TOKENS

=====

### What is a Token?

-----

A token is the **\*\*smallest meaningful unit\*\*** in a program.

### Token Format:

<token-name, attribute-value>

Example:

<id, pointer-to-symbol-table>

Types of Tokens

-----

1. Keywords – int, float, if
2. Identifiers – variable names
3. Operators – +, -, \\*, /
4. Literals – numbers, characters
5. Separators – ;, ,, ()

RECOGNITION OF TOKENS

=====

Tokens are recognized using **\*\*regular expressions\*\***.

Token Type	Regular Expression
Identifier	\[a-zA-Z\]\[a-zA-Z0-9\]\*
Number	\[0-9\]+\.

### ### Real-Life Example:

A mobile number must have **\*\*10 digits\*\*** and start with **\*\*6–9\*\***. This is pattern recognition, similar to token recognition.

### HAND-WRITTEN LEXICAL ANALYZER

=====

What is a Hand-Written Lexical Analyzer?

-----

A lexical analyzer written manually using a programming language.

### ### Advantages:

- \* Full control
- \* Efficient for small languages

### ### Disadvantages:

- \* Time-consuming
- \* Error-prone

### Working Logic:

1. Read character
2. Group characters
3. Generate token

LEX TOOL

=====

What is LEX?

-----

LEX is an **automatic lexical analyzer generator**.

\* Input: Token rules

\* Output: C program (lex.yy.c)

Structure of a LEX Program

-----

%{ C declarations %} %% Regular expressions and actions %% User-defined functions `

### Example of a LEX Program

-----

```
%{ #include %} %% int|float { printf("Keyword\n"); } [a-zA-Z]+ { printf("Identifier\n"); }
[0-9]+ { printf("Number\n"); } . { printf("Special Character\n"); } %% int yywrap() {
return 1; } `
```

### Input:

```
` int a = 10; `
```

### Output:

```
` Keyword Identifier Special Character Number Special Character `
```

### REAL-LIFE APPLICATION OF LEX

=====

ATM machines:

- \* Card number → digits
- \* PIN → digits
- \* Commands → keywords

LEX rules identify patterns just like ATM systems.