How Compilers Translate Code to Machine Language

A **compiler** is a program that translates **high-level source code** (e.g., C++, Java, Python) into **machine-readable code** (e.g., binary or assembly). Here's a step-by-step breakdown of how it works:

1. Lexical Analysis (Scanning)

- **Goal:** Break the source code into **tokens** (keywords, identifiers, operators, etc.).
- **Process:** The **lexer/scanner** reads characters and groups them into meaningful tokens.
- **Example:**
  `x = 10 + y;` → Tokens: `x`, `=`, `10`, `+`, `y`, `;`.

2. Syntax Analysis (Parsing)

- **Goal:** Check if tokens follow the language's grammar rules.
- **Process:** The **parser** builds an **Abstract Syntax Tree (AST)** to represent the code structure.
- **Errors:** Syntax errors (e.g., missing `;` or mismatched parentheses).
- **Example:**
  The AST for `x = 10 + y` becomes a tree with `=` as the root, `x` as the left child, and `+` as the right child with `10` and `y` as leaves.

3. Semantic Analysis

- **Goal:** Ensure code "makes sense" logically (e.g., type checking, scope rules).
- **Process:** The compiler verifies variables are declared, types match, and operations are valid.
- **Example:**
  If `y` is a string, `10 + y` triggers a type error.

4. Intermediate Code Generation

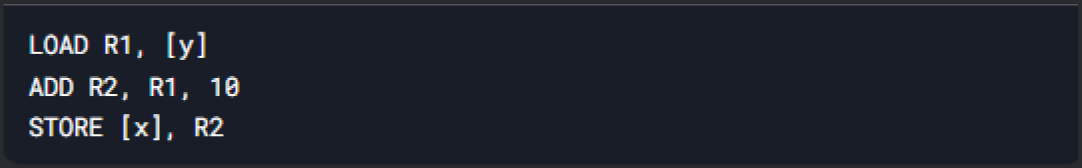
- **Goal:** Create a platform-independent intermediate representation (IR) like **three-address code**.
- **Example:**
  `x = 10 + y` → `t1 = 10 + y; x = t1`.

5. **Optimization**

- **Goal:** Improve IR for efficiency (speed/memory) without changing behavior.
- **Techniques:**
  - Remove dead code.
  - Simplify expressions (e.g., `2 * 3` → `6`).
  - Loop unrolling or inline functions.

6. **Code Generation**

- **Goal:** Translate optimized IR into **machine/assembly code** for the target CPU.
- **Process:** Maps variables to registers/memory, selects instructions, and handles addressing modes.
- **Example:**
  `x = 10 + y` →



```
LOAD R1, [y]
ADD R2, R1, 10
STORE [x], R2
```

7. **Linking (Optional)**

- **Goal:** Combine compiled code with external libraries (e.g., `printf` in C).
- **Process:** The **linker** resolves references to create an executable.

Key Differences: Compiler vs. Interpreter

- **Compiler:** Translates the entire program upfront (e.g., C, C++).
- **Interpreter:** Translates and executes line-by-line (e.g., Python, JavaScript).
- **JIT Compiler:** Hybrid approach (used in Java, .NET) that compiles code at runtime.

Summary Flow:

Source Code → Lexer → Parser → Semantic Check → IR → Optimizer → Machine Code → **Executable**.