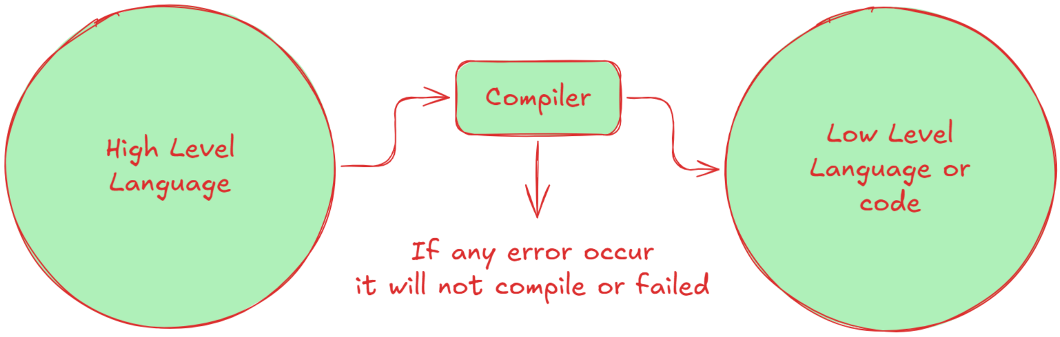
Unit 1 : Introduction to Compiler Design

Introduction to Compiler Design

Compiler is the software that convert the high level language (source code) to low level language i.e. machine understandable code. Compiler Design is the process of developing a compiler.



The development of the compilers are continuous as evolution in computer science and programming languages.

History of Compilers

- 1952 – First compiler by Grace Hopper (A-0 system).
- 1957 – FORTRAN compiler by IBM (first high-level language compiler).
- 1960s – More languages like COBOL, ALGOL; basic compiler theory developed.
- 1970s – Tools like Lex/Yacc created; C language compilers improved.
- 1980s – Focus on optimization and portable compilers (e.g., GCC).
- 1990s – Rise of C++, Java; support for object-oriented features.
- 2000s – JIT compilation (Java, .NET); LLVM introduced.
- 2010s–Now – Support for AI/ML, multi-core, and modern hardware.

Definition

The compiler is the software program that will convert the high level source code to machine code , bytecode or another programming language. The source code is written in high level language that is human readable and understand easily. (C/C++, JAVA, and Python)

Phases of Compiler Design

A compiler has two parts : Analysis and Synthesis and these two parts are further divided into 6 phases :

- Lexical Analysis – Scans source code to produce tokens.
    - Tokens :

  - A token is the smallest unit of meaning in source code.
  - It is produced by the lexical analyzer.
  - Each token has a type (like IDENTIFIER, KEYWORD, NUMBER) and a lexeme (the actual text).
  - Tokens are used by the parser to check syntax.
    Examples:
    - int → KEYWORD
    - x → IDENTIFIER
    - = → OPERATOR
    - 10 → LITERAL
    - ; → SEPARATOR
- Syntax Analysis (Parsing) –
  - Syntax analysis is the second phase of a compiler.
  - It takes tokens from the lexical analyzer as input.
  - Checks if the token sequence follows the grammar rules of the language.
  - Detects syntax errors (e.g., missing ;, wrong order of expressions).
  - Builds a parse tree (or syntax tree) showing the structure of the code.
  - Uses context-free grammar (CFG) to define valid syntax.
  - Parsing techniques include:
    - Top-down parsing (e.g., recursive descent)
    - Bottom-up parsing (e.g., LR parser)
    - Example

  ```
  x = a + b * c;
  ```

- Semantic Analysis – Validates semantics (meaning) using symbol tables.
  - Semantic analysis is the third phase of the compiler.
  - Checks the meaning of code after syntax is verified.
  - Ensures semantic correctness of statements.
  - Performs type checking, variable declarations, function calls, etc.
  - Uses a symbol table to store information about identifiers.
  - Detects errors like:
    - Using undeclared variables
    - Type mismatches `(int = float)`
    - Incorrect function arguments
      Example :

```
int x;
x = "hello";  // Semantic error: assigning string to int
```

- Intermediate Code Generation – Converts to an intermediate representation.
    - This is the fourth phase of the compiler.
    - Converts the syntax tree or parse tree into an intermediate code.
    - The intermediate code is machine-independent.
    - Acts as a bridge between the front end (analysis) and back end (code generation).
    - Easier to optimize than source code.
    - Common forms of intermediate code:
        - Three-address code (TAC)
        - Postfix notation (Polish notation)
        - Abstract Syntax Tree (AST)
        - Example (for `a = b + c * d;` in TAC):
        - ```
          t1 = c * d
          t2 = b + t1
          a = t2
          ```
- Code Optimization –
    - It is the fifth phase of the compiler.
    - Improves the performance of intermediate code.
    - Makes the code faster or uses less memory.
    - Does not change the output or meaning of the program.
    - Types:
        - Machine-independent (e.g., removing dead code)
        - Machine-dependent (e.g., using CPU-specific instructions)
    - Common techniques:
        - Constant folding: `x = 2 + 3 → x = 5`
        - Dead code elimination: Removes unused code
        - Loop optimization: Reduces repeated calculations
    - Goal: efficient and optimized final machine code.
- Code Generation – Produces machine-level code.
    - This is the final phase of the compiler.
    - Converts intermediate code into machine code or assembly code.
    - Considers target architecture (CPU, registers, instructions).
    - Ensures efficient use of CPU registers and memory.
          ```

- Generates code that is correct, optimized, and executable.
- Handles:
  - Instruction selection (choosing right machine instructions)
  - Register allocation
  - Instruction ordering
- Example:
- ```
  a = b + c;
  ```

  Intermediate Code:
- ```
  t1 = b + c
  a = t1
  ```

  Machine Code (Example):
- ```
  MOV R1, b
  ADD R1, c
  MOV a, R1
  ```
  ```