**Unit I**

**Definition of Compiler**

* A **compiler** is a software system that:
  + Reads a program written in a **source language**.
  + Translates it into an equivalent program in a **target language** (e.g., machine code, bytecode).
  + Reports **errors** detected during translation.
* **Key purpose**: Enable execution of software by translating high-level code into a form executable by machines.

**Compiler vs. Interpreter**

|  |  |
| --- | --- |
| **Compiler** | **Interpreter** |
| Translates the entire source program to target code before execution. | Directly executes the source program line-by-line without generating target code. |
| Produces a standalone executable. | Requires the source code during execution. |
| Examples: C, C++ compilers. | Examples: Python, JavaScript interpreters. |

**Classification Of Compiler**

**1. Single Pass Compiler**

* **Definition**:
* A **single-pass compiler** processes the entire source code in just **one traversal** (one pass) and generates the final machine code directly.
* It performs **lexical analysis, parsing, semantic analysis, and code generation in a single phase** without intermediate representations.
* **Key Features**:
  + **Minimal memory usage** – Since it doesn’t store an intermediate representation (IR), it requires less memory.
  + **Fast compilation** – The source code is read only once, making it quick.
  + **Suitable for simple languages – Used in early programming languages like Pascal.**
* **Drawbacks**:
  + Cannot handle backward references (e.g., jumps to labels declared later).
  + Restricted to languages with **declaration-before-use** rules.

**2. Multi Pass Compiler**

* **Definition**:
  + A **multi-pass** compiler processes the source code in multiple traversals (passes), where each pass performs a specific task.
  + Each pass performs specific tasks (e.g., parsing, optimization, code generation).
* The output of one pass is used as input for the next pass.
* It often uses **Intermediate Representations (IR)** such as **Abstract Syntax Trees (ASTs)** or **Three-Address Code (TAC)**.
* **Key Features**:
  + **Advanced optimizations** – It supports powerful optimizations like **loop unrolling, constant folding, and dead code elimination.**
  + **Handles complex language features** – It can manage **forward references, cross-module dependencies, and dynamic memory allocation**
* **Better error handling** – Since it analyzes the code multiple times, error detection is improved

**3. Load-and-Go Compiler**

* **Definition**:
  + A **Load-and-Go Compiler** compiles and executes code **immediately** without producing an object file.
  + It **compiles the source code, loads the machine code into memory, and executes it right away**.
  + There is **no separate linking step** like in traditional compilers.
* **Key Features**:
  + Suitable for **small programs** or environments with limited memory.
  + No persistent object files (output is executed directly).
  + Common in **educational systems** (e.g., early BASIC interpreters) and embedded systems.

**Parts of Compilation**

The compilation process is divided into two major phases:

1. **Analysis (Frontend)**
2. **Synthesis (Backend)**

**1. Analysis Phase (Frontend)**

**Purpose**: Understand the structure and meaning of the source program.  
**Tasks**:

* **Lexical Analysis**:
  + Breaks the source code into **tokens** (e.g., keywords, identifiers, operators).
  + Example: x = y + 5 → Tokens: [x], [=], [y], [+], [5].
* **Syntax Analysis (Parsing)**:
  + Checks syntax using **grammar rules** and builds a **parse tree** or **Abstract Syntax Tree (AST)**.
  + Example: Verifies x = y + 5 follows the language’s expression rules.
* **Semantic Analysis**:
  + Ensures the program is **meaningful** (e.g., type checking, scope resolution).
  + Example: Checks if y and 5 are compatible types for addition.

**Output**: Intermediate Representation (IR) + Symbol Table.

**2. Synthesis Phase (Backend)**

**Purpose**: Generate efficient target code from the IR.  
**Tasks**:

* **Intermediate Code Generation**:
  + Converts the AST/parse tree into a **machine-independent IR** (e.g., three-address code).
  + Example: t1 = y + 5; x = t1.
* **Code Optimization**:
  + Improves IR for efficiency (e.g., removing redundant code, loop optimizations).
  + Example: Replaces t1 = y + 5; x = t1 with x = y + 5 if possible.
* **Code Generation**:
  + Maps optimized IR to **machine-specific instructions** (assembly/machine code).
  + Example: Generates MOV R1, y; ADD R1, 5; MOV x, R1.

**Output**: Target program (e.g., executable, object code).

**Comparison: Analysis vs. Synthesis**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Analysis (Frontend)** | **Synthesis (Backend)** |
| **Focus** | Source language correctness | Target language efficiency |
| **Dependencies** | Source-language specific | Target-machine specific |
| **Key Output** | IR + Symbol Table | Optimized machine code |
| **Phases** | Lexical, Syntax, Semantic Analysis | Code Optimization, Code Generation |

**Phases of a Compiler**

**1. Lexical Analysis (Scanner)**

**Purpose**:  
Convert the **stream of characters** (source code) into a **stream of meaningful tokens**.

**What Does It Do?**

1. **Reads Input**:
   * Scans the source code character-by-character (like reading a book letter-by-letter).
2. **Groups Characters into Tokens**:
   * Tokens are the smallest meaningful units in a programming language (e.g., keywords, identifiers, numbers, operators).
3. **Removes Trivia**:
   * Ignores whitespace, comments, and other non-essential characters.
4. **Reports Errors**:
   * Detects invalid tokens (e.g., @ in a language that doesn’t support it).

**Example**:  
x = y + 5; → Tokens: [Identifier(x)], [Operator(=)], [Identifier(y)], [Operator(+)], [Number(5)], [Semicolon(;)].

**Why it matters**:

* First phase of compilation.
* Feeds tokens to the **Syntax Analyzer**.

**2. Syntax Analysis (Parsing)**

**Purpose**:

* Analyze the **grammatical structure** of the token stream produced by the lexical analyzer.
* Build a **tree-like representation** (e.g., parse tree or Abstract Syntax Tree) to depict the structure.

**What Does It Do?**

1. **Validates Syntax**:
   * Checks if the token stream follows the **grammar rules** of the programming language.
   * Example: Ensures *if (x > 10) { ... }* follows the correct syntax for an if statement.
2. **Constructs a Parse Tree or Abstract Syntax Tree (AST)**:
   * A **parse tree** represents the syntactic structure of the program.
   * An **AST** is a simplified version of the parse tree, focusing on the essential structure.

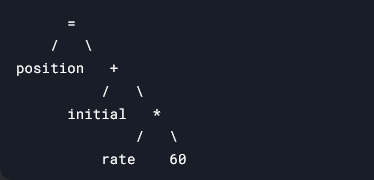
**Key Concepts**

**1. Grammar Rules**

* Syntax analysis uses context-free grammar (CFG) to define the structure of valid programs.

**2. Parse Tree**

* A tree where:
  + Interior nodes represent operations or grammar rules.
  + Leaf nodes represent tokens (e.g., identifiers, numbers).
* **Example:** For the expression position = initial + rate \* 60:



**3. Abstract Syntax Tree (AST)**

* A simplified version of the parse tree, focusing on the essential structure.

**3. Semantic Analysis**

**Purpose**:

* Ensure the program is **semantically valid** (logically meaningful) according to the language rules.
* Gather and store **type information** for use in later phases (e.g., intermediate code generation).

**Key Tasks**

1. **Type Checking**:
   * Verify that **operators** have **compatible operands** (e.g., no adding a string to an integer).
   * Example: Ensure an array index is an integer, not a float.
2. **Type Coercion/Conversions**:
   * Apply implicit or explicit type conversions when allowed by the language.
   * Example: Convert an integer to a float in mixed-type expressions.
3. **Scope Resolution**:
   * Ensure variables/functions are **declared before use** and follow scoping rules.
4. **Annotate Syntax Tree/Symbol Table**:
   * Attach type information to nodes in the syntax tree or update the symbol table.

4. **Intermediate Code Generation**

**Purpose**:

* Translate the annotated syntax tree/symbol table into a **low-level, machine-like intermediate representation (IR)**.

**Key Points:**

1. **Forms of Intermediate Representation**:
   * **Syntax trees** (used during syntax/semantic analysis).
   * **Three-address code**:
     + Sequence of assembly-like instructions with **≤3 operands per instruction**.
2. **Properties of Intermediate Code**:
   * **Easy to produce** from the source program.
   * **Easy to translate** into target machine code.
3. **Characteristics of Three-Address Code**:
   * Each instruction has **at most one operator** on the right side.
   * Fixes the **order of operations** (e.g., multiplication before addition).
   * Uses **temporary names** (e.g., t1, t2) to hold computed values.
   * Some instructions may have **fewer than three operands** (e.g., assignments).

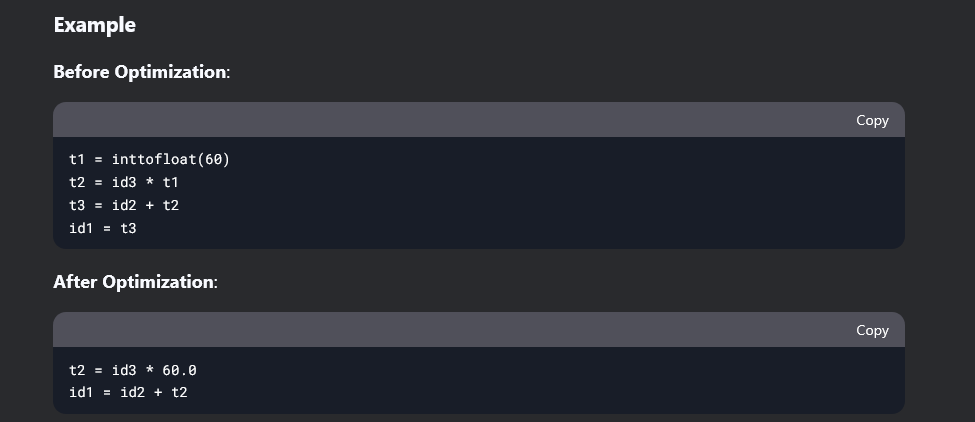
**5. Code Optimization**

**Purpose**:

* Improve the **intermediate code** to generate **better target code** (faster, shorter, or more efficient).

**Key Points:**

1. **Machine-Independent**:
   * Optimizations are applied to the intermediate code, **not** specific to the target machine.
2. **Types of Improvements**:
   * **Speed**: Make the program run faster.
   * **Size**: Reduce code length.
   * **Power Efficiency**: Optimize for lower energy consumption.
3. **Example Optimization**:
   * **Eliminate Redundant Operations**:
     + Replace *inttofloat(60)* with *60.0* (compile-time conversion).
   * **Remove Temporary Variables**:
     + If a temporary variable (e.g., *t3*) is used only once, replace it directly.
4. **Trade-offs**:
   * **Optimizing Compilers** may spend significant time on this phase.
   * **Simple Optimizations** can improve performance without slowing compilation drastically.



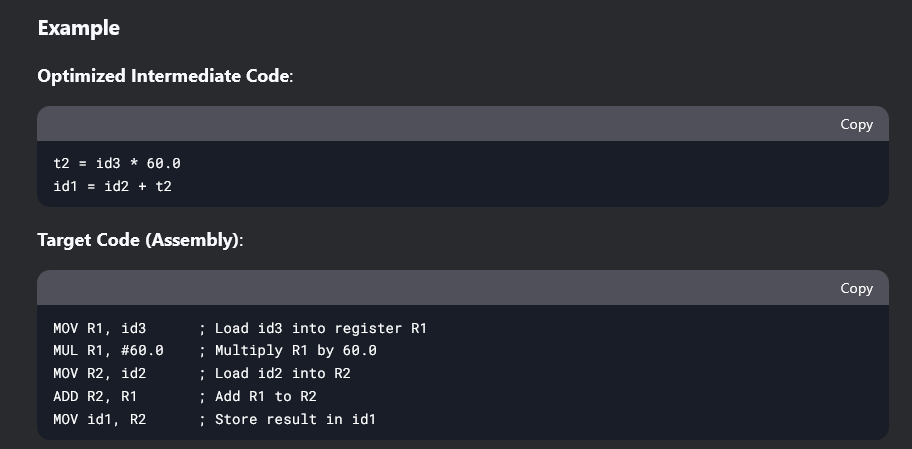
**6. Target Code Generation**

**Purpose**:

* Translate **optimized intermediate code** into **machine-specific instructions** (assembly or binary code).

**Key Points:**

1. **Machine-Dependent**:
   * Generates code tailored to the **target machine’s architecture** (registers, instruction set, memory).
2. **Tasks**:
   * **Register Allocation**: Assign variables/temporaries to CPU registers.
   * **Instruction Selection**: Map intermediate operations to machine instructions (e.g., ADD, MOV).
3. **Output**:
   * **Assembly Code** (human-readable) or **Binary Code** (executable).



**Symbol Table Manager**

**Purpose**:

* Track **variable and procedure names** used in the source program and store their **attributes**.

**Key Points:**

1. **Attributes Stored**:
   * **Variables**:
     + Storage allocation.
     + Data type.
     + Scope (where the variable can be used).
   * **Procedures**:
     + Number and types of arguments.
     + Argument-passing method (by value, by reference).
     + Return type.
2. **Symbol Table Structure**:
   * A **data structure** with a **record for each name**.
   * Designed for **fast insertion, lookup, and retrieval** of attributes.