**Chapter One**

**What is a Compiler?**

A **compiler** is a software program that translates code written in a high-level programming language (source code) into a lower-level language, typically **machine code** or **intermediate code**. The output from the compiler can be directly executed by a computer's hardware or interpreted by another program.

**Key Features of a Compiler:**

* **Translation**: Converts high-level code (C, C++, Java etc.) into machine code (binary instructions) that the processor can execute.
* **Optimization**: Improves the efficiency of the generated code, optimizing for factors like speed or memory usage.
* **Error Checking**: Detects and reports errors in the source code, such as syntax or semantic errors.
* **Code Generation**: Produces the final machine code or bytecode for execution.

**What is Compiler Design?**

**Compiler design** refers to the theory, architecture, and methodology of building compilers. It includes understanding the algorithms and data structures that are used in each phase of the compilation process and deciding how to optimize the code generation and error handling.

**Goals of Compiler Design:**

* **Correctness**: The compiler must generate correct machine code that faithfully executes the behavior of the source code.
* **Efficiency**: The generated code should be efficient in terms of speed and resource usage.
* **Portability**: The compiler should be adaptable to different target architectures, so the same source code can be compiled for multiple platforms.
* **Error Reporting and Recovery**: Provide informative feedback to programmers and attempt to recover from errors when possible.

**Key components of translation and execution**

**1. Compiler**

* **Definition**: A compiler is a program that translates high-level programming language code (e.g., C, C++, Java) into machine code (binary executable) that can be understood by the computer's hardware.
* **How it works**: Compilers convert the entire source code at once into machine code. If there are any errors, they are reported during this compilation process.
* **Example languages**: C, C++, Rust, Go
* **Output**: Machine code (executable) or intermediate code (bytecode, assembly).

**2. Interpreter**

* **Definition**: An interpreter executes the code directly, translating it line by line or statement by statement into an intermediate form or machine code at runtime.
* **How it works**: It reads the source code and executes it directly without producing an intermediate machine code file. Errors are reported during execution.
* **Example languages**: Python, Ruby, JavaScript, PHP
* **Output**: It doesn't produce machine code files; instead, it executes the code dynamically.

**3. Assembler**

* **Definition**: An assembler translates assembly language (low-level code close to machine code) into actual machine code (binary instructions that the processor understands).
* **How it works**: Assembly language is closely tied to the hardware's instruction set, and the assembler directly converts this code into machine-specific instructions.
* **Example languages**: Assembly language (ASM), often used in low-level system programming and embedded systems.
* **Output**: Machine code (binary).

**4. Preprocessor**

* **Definition**: The preprocessor is a tool that processes the source code before compilation. It handles directives (e.g., #include, #define) and performs tasks like macro substitution, file inclusion, and conditional compilation.
* **How it works**: It runs before the actual compilation begins, modifying the source code by expanding macros or including files. This modified code is then passed to the compiler.
* **Example**: C/C++ preprocessors (#include, #define), which modify the code before it reaches the compiler.
* **Output**: Modified source code with preprocessing directives resolved.

**5. Linker**

* **Definition**: A linker combines multiple object files (produced by the compiler) and resolves references between them to produce a single executable.
* **How it works**: After compilation, programs often consist of several object files. The linker brings them together and resolves symbols or addresses that need to interact between these files.
* **Output**: An executable file (e.g., .exe, .out).

**6. Loader**

* **Definition**: A loader is responsible for loading the executable into memory and preparing it for execution by the operating system.
* **How it works**: It reads the executable file, loads the necessary code and data into memory, and passes control to the starting point of the program.

**7. Virtual Machine (VM)**

* **Definition**: A virtual machine interprets intermediate bytecode into machine code for execution on the host machine.
* **How it works**: For languages like Java or C#, the compiler produces bytecode that runs on a VM (like the JVM or .NET CLR). The VM translates the bytecode into machine code or interprets it at runtime.
* **Example**: Java Virtual Machine (JVM), Common Language Runtime (CLR).

**Summary of Differences:**

* **Compiler**: Translates entire code to machine code ahead of time.
* **Interpreter**: Translates and executes code line by line at runtime.
* **Assembler**: Converts assembly language to machine code.
* **Preprocessor**: Processes source code before compilation, handling directives like #include and #define.
* **Linker**: Combines object files and resolves external references to produce an executable.
* **Loader**: Loads the executable into memory and prepares it for execution.

**Phases and Passes during the compilation process**

**Compiler Phases**

A compiler typically processes the source code in multiple **phases**, each responsible for a specific task in transforming source code into executable code. These phases can be broadly grouped into **front-end** and **back-end** categories.

**1. Analysis Phase (Front-End)**

The analysis phase breaks down the source code into an intermediate representation (IR) and checks it for correctness.

* 1. **Lexical Analysis (Scanner)**
* **Task**: Breaks the source code into tokens.
* **Input**: Raw source code (characters).
* **Output**: A sequence of tokens (identifiers, keywords, symbols, etc.).
* **Example**: Converting int x = 10; into tokens like int, x, =, 10, ;.
  1. **Syntax Analysis (Parser)**
* **Task**: Checks whether the sequence of tokens follows the grammatical structure (syntax) of the language.
* **Input**: Tokens from the lexical analyzer.
* **Output**: A syntax tree (also called a parse tree).
* **Example**: Verifies if int x = 10; is valid according to the grammar rules of the language and builds a tree structure representing the statement.
  1. **Semantic Analysis**
* **Task**: Ensures the semantic correctness of the code, such as type checking and variable declarations.
* **Input**: Syntax tree from the parser.
* **Output**: Annotated syntax tree with semantic information (e.g., type information).
* **Example**: Checks if the variable x is declared and if its type matches the assigned value.
  1. **Intermediate Code Generation**
* **Task**: Generates an intermediate representation (IR) of the source code that is easier to optimize and translate into machine code.
* **Input**: Annotated syntax tree.
* **Output**: Intermediate code, usually in a three-address code (TAC) or abstract machine language.
* **Example**: Generates IR like t1 = 10, x = t1 for int x = 10.
* **Phases of a Compiler**

**2. Synthesis Phase (Back-End)**

The synthesis phase generates the final machine code or bytecode, optimizing the code during the process.

**2.1. Code Optimization**

* **Task**: Improves the intermediate code by removing redundancies and optimizing for performance (speed, memory, etc.).
* **Input**: Intermediate code.
* **Output**: Optimized intermediate code.
* **Example**: Eliminates dead code (unused variables or unreachable code) and optimizes loops.

**2.2. Code Generation**

* **Task**: Converts the optimized intermediate code into machine code (or assembly code) specific to the target architecture (CPU).
* **Input**: Optimized intermediate code.
* **Output**: Machine code or assembly code.
* **Example**: Transforms the intermediate code t1 = 10 into machine instructions specific to the CPU.

**2.3. Code Linking and Assembly**

* **Task**: Involves assembling the machine code into an executable and linking with other object files or libraries.
* **Input**: Machine code.
* **Output**: Executable code.
* **Example**: Linking multiple object files into a final .exe or .out file.
* Here’s a summary of the key **phases** of a compiler:

|  |  |
| --- | --- |
| **Phase** | **Purpose** |
| **Lexical Analysis** | Converts source code into tokens. |
| **Syntax Analysis** | Checks token sequences against grammar rules to build a syntax tree. |
| **Semantic Analysis** | Ensures logical correctness (e.g., type-checking). |
| **Intermediate Code Generation** | Converts the source code into an intermediate representation (IR). |
| **Code Optimization** | Improves the efficiency of the intermediate code (optional phase). |
| **Code Generation** | Translates IR into machine code or bytecode. |
| **Code Linking** | Links external libraries and resolves symbol references (if needed). |

**Compiler Passes**

A **pass** refers to how many times the compiler traverses the source code (or the intermediate representation) during the compilation process. A **single-pass compiler** processes each phase once, while a **multi-pass compiler** may process the code in multiple passes for additional analysis or optimization.

**Types of Passes**

1. **Single-Pass Compiler**:
   * **Definition**: A compiler that reads the source code and performs all compilation tasks (lexical analysis, syntax analysis, semantic analysis, code generation) in one go.
   * **Advantages**: Fast and simple.
   * **Disadvantages**: Limited optimizations and complex to implement for larger languages.
   * **Example**: Early versions of Pascal compilers were single-pass.
2. **Multi-Pass Compiler**:
   * **Definition**: A compiler that processes the source code in multiple stages or passes, often going over the intermediate representation multiple times to apply further optimizations or transformations.
   * **Advantages**: Better optimization, more modular, easier to debug.
   * **Disadvantages**: Slower due to multiple passes.
   * **Example**: Modern compilers like GCC (GNU Compiler Collection) use multiple passes to optimize code.

**Passes in Compiler Phases:**

* **Front-End Passes**: Usually handle **lexical analysis**, **syntax analysis**, and **semantic analysis**. These are typically done in a single pass or as separate passes in multi-pass compilers.
* **Back-End Passes**: Include **code optimization**, **code generation**, and **assembly**. Multiple passes are often used in this stage to optimize the intermediate code.

**Example of Passes in Compilation:**

1. **First Pass**: The compiler reads the source code, performs lexical and syntax analysis, and generates an intermediate representation.
2. **Second Pass**: The compiler processes the intermediate representation to check for semantic errors and performs basic optimizations.
3. **Third Pass**: The compiler further optimizes the intermediate code and generates the target machine code.

**Summary**

* **Compiler Phases** refer to the distinct steps in the compilation process, like lexical analysis, syntax analysis, code generation, and optimization.
* **Compiler Passes** describe how many times the source code or intermediate code is processed during compilation. Single-pass compilers are faster but less powerful, while multi-pass compilers allow for greater optimization and error checking.

**Classification of compiler**

Compilers can be classified based on different criteria such as the target language they produce, the number of passes they make, the platform they run on, and how they execute the source program. Here are the major classifications of compilers:

**1. Based on the Target Language**

**1.1. Native Compiler**

* **Description**: Produces machine code or executable code for the same machine on which the compiler is running.
* **Example**: A C compiler running on a Windows machine and generating a Windows executable.
* **Use Case**: Used when the same system is used for development and execution.

**1.2. Cross Compiler**

* **Description**: Generates machine code for a different target system or platform than the one on which the compiler is running.
* **Example**: A compiler running on a Windows machine that generates code for an embedded system (e.g., ARM architecture).
* **Use Case**: Common in embedded systems development where the target platform differs from the development environment.

**1.3. Source-to-Source Compiler (Transpiler)**

* **Description**: Translates code from one high-level programming language to another high-level programming language.
* **Example**: A TypeScript-to-JavaScript compiler.
* **Use Case**: Used when migrating code between languages or converting higher-level abstractions into more widely supported languages.

**1.4. Bytecode Compiler**

* **Description**: Translates high-level language source code into an intermediate form (bytecode), which is interpreted or further compiled by another platform-specific compiler or virtual machine (VM).
* **Example**: The Java compiler that generates Java bytecode, which runs on the Java Virtual Machine (JVM).
* **Use Case**: Used in virtual machine environments where platform independence is needed.

**2. Based on the Number of Passes**

**2.1. Single-Pass Compiler**

* **Description**: The compiler processes the source code in a single pass, performing lexical analysis, syntax analysis, and code generation in one go.
* **Example**: Early versions of C compilers.
* **Use Case**: Suitable for simple languages and environments where fast compilation is critical, but limited in optimization capabilities.

**2.2. Multi-Pass Compiler**

* **Description**: The compiler goes through the source code multiple times, performing different tasks in each pass (e.g., one pass for syntax checking, another for optimization, etc.).
* **Example**: Modern C and C++ compilers, which optimize code across multiple passes.
* **Use Case**: Used for more complex languages, where optimization and detailed error checking are important.

**2.3. Two-Pass Compiler**

* **Description**: A special case of a multi-pass compiler, where the compiler makes exactly two passes over the source code. The first pass builds a symbol table and generates intermediate code, while the second pass performs code generation.
* **Example**: Pascal compilers often follow this model.
* **Use Case**: Provides better optimization than single-pass compilers without the overhead of multiple passes.

**3. Based on Execution Strategy**

**3.1. Just-in-Time (JIT) Compiler**

* **Description**: Compiles code at runtime, translating intermediate code (e.g., bytecode) into machine code just before execution.
* **Example**: Java JIT compiler within the Java Virtual Machine (JVM), or .NET CLR JIT compiler.
* **Use Case**: Used in environments like Java or .NET where the code is first compiled to an intermediate format and then executed by the JIT compiler for performance optimization.

**3.2. Ahead-of-Time (AOT) Compiler**

* **Description**: Compiles the code directly into machine code before execution (during build time), rather than at runtime.
* **Example**: Compilers for statically typed languages like C, C++, or Rust.
* **Use Case**: Suitable for languages where compilation happens before execution, optimizing for speed and resource usage at runtime.

**4. Based on Platform**

**4.1. Host Compiler**

* **Description**: Runs on the same system that it is generating code for.
* **Example**: GCC running on Linux and generating a binary for Linux.
* **Use Case**: Most common in general software development, where the development and execution environments are the same.

**4.2. Cross-Compiler**

* **Description**: Generates machine code for a platform other than the one on which the compiler itself is running. This is particularly useful when developing software for systems that have different architectures or operating environments than the development system.
* **Example**: A cross-compiler running on a Windows system but generating code for an ARM-based embedded system.
* **Use Case**: Critical for embedded system development and environments where the development and target platforms differ.

**5. Based on Output**

**5.1. Machine Code Compiler**

* **Description**: Directly generates machine code or object code that the processor can execute.
* **Example**: GCC (GNU Compiler Collection) generates x86 or ARM machine code.
* **Use Case**: Used in system-level programming where the compiled code runs directly on the hardware.

**5.2. Intermediate Code Compiler**

* **Description**: Generates an intermediate representation (IR) or bytecode, which needs to be further processed by a virtual machine or interpreter.
* **Example**: Java compiler generating Java bytecode for the JVM.
* **Use Case**: Used in virtual machine-based environments (JVM, CLR), enabling platform independence and portability.

**Summary of Compiler Classifications**

|  |  |  |
| --- | --- | --- |
| **Category** | **Type** | **Example** |
| **Target Language** | Native, Cross, Source-to-Source | GCC, ARM cross-compiler, TypeScript |
| **Number of Passes** | Single-Pass, Multi-Pass, Two-Pass | Early C compiler, Modern C++, Pascal |
| **Execution Strategy** | JIT, AOT | Java JIT, .NET CLR, C/C++ compiler |
| **Source Language** | High-Level, Scripting, DSL | GCC, Python, SQL |
| **Optimization** | Optimizing, non-optimizing | GCC with -O3, educational compilers |
| **Platform** | Host, Cross | GCC, ARM cross-compiler |
| **Output** | Machine Code, Intermediate | GCC, Java compiler |
| **Purpose** | General, Specific | Clang, VHDL compiler |
| **Incremental vs Batch** | Batch, Incremental | GCC, IDE incremental compiler |
| **Retargetable Compiler** | Retargetable | LLVM |

**Symbol Table in Compiler Design**

The **symbol table** is a critical data structure used by the compiler during the compilation process. It stores information about identifiers (like variables, functions, classes, etc.) declared in the source code. The symbol table helps the compiler keep track of **scope**, **type information**, **memory locations**, and **other attributes** of identifiers.

**Key Roles of Symbol Table:**

1. **Stores Information**:
   * **Identifiers**: Variable names, function names, constants, etc.
   * **Types**: Data types associated with the identifiers (e.g., int, float, class).
   * **Scope**: Tracks in which scope (local, global, etc.) an identifier is declared.
   * **Memory Location**: For variables and functions, it stores the memory address (or offset in assembly/machine code).
   * **Attributes**: Any additional information such as the number of parameters for a function, array dimensions, etc.
2. **Manages Scope**:
   * **Global scope**: Tracks globally defined symbols.
   * **Local scope**: Manages symbols local to functions, classes, or blocks. In block-structured languages like C or C++, the symbol table must distinguish between variables declared in different scopes.
3. **Helps in Semantic Analysis**:
   * During semantic analysis, the compiler checks the **correct usage of variables and functions**, such as checking if a variable is declared before it is used, matching types, etc.
   * The symbol table helps to enforce **type safety** and **name resolution** (binding uses of variables to their declarations).

**Operations on Symbol Table:**

* **Insert**: Add a new identifier and its associated information.
* **Lookup**: Search for an identifier to retrieve its details.
* **Delete**: Remove an identifier when it goes out of scope (especially in languages with block structures).

**Example:**

int x = 5;

float y = 3.14;

The symbol table might look like:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Identifier** | **Type** | **Scope** | **Memory Location** | **Attributes** |
| x | int | global | 0x100 | None |
| y | float | global | 0x104 | None |

**Symbol Table Structure:**

* **Hash Table**: The most common structure for symbol tables, allowing fast insertion and lookup operations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Hash Value** | **Identifier** | **Type** | **Scope** | **Attributes** |
| 1 | x | int | 1 | memory: 0x100 |
| 2 | y | float | 1 | memory: 0x104 |
| 3 | foo | func | 0 | return type: int |

**Scope Management with Hash Tables:**

* **Linked List**: Used in some implementations, especially when managing scopes with a stack-like structure.

| name: x | type: int | scope: 1 | memory: 0x100 | -> | name: y | type: float | scope: 1 | memory: 0x104 | -> NULL

**Error Handling in Compiler Design**

Error handling is an important task in compilers, ensuring that meaningful and understandable feedback is given to the programmer when something goes wrong. Errors can occur during any phase of the compilation process.

**Types of Errors:**

1. **Lexical Errors**:
   * Occur during **lexical analysis** when the compiler encounters an unrecognized token or invalid character in the source code.
   * **Example**: An invalid character like # in a language where it’s not allowed.

**Handling**: The compiler generates a message indicating an illegal character or identifier and continues parsing the remaining tokens.

1. **Syntax Errors**:
   * Occur during **syntax analysis** when the source code does not conform to the grammar rules of the language.
   * **Example**: Missing a semicolon in int x = 10

**Handling**: The compiler can use techniques like **panic-mode recovery** (the parser skips ahead in the input to a pre-designated set of tokens (like semicolons or braces) where it expects to be in a syntactically valid state) or **phrase-level recovery** (attempting to fix the structure in place, e.g., by inserting a missing semicolon).

1. **Semantic Errors**:
   * Occur during **semantic analysis** when the meaning of the code is incorrect, even though the syntax is correct.
   * **Example**: Type mismatches (e.g., trying to assign a float to an int), undeclared variables, or using a function with the wrong number of parameters.

**Handling**: The compiler reports the error and provides information about the type mismatch, undeclared variables, or incorrect function calls. It may attempt to recover from this error by continuing to check the remaining code.

1. **Logical Errors**:
   * These are not typically caught by the compiler but are logical mistakes in the code that lead to incorrect behavior at runtime.
   * **Example**: Writing x = x - 1 when the intention was x = x + 1.

**Handling**: Not the compiler's responsibility—these errors need to be handled via debugging and testing.

1. **Runtime Errors**:
   * Errors that occur during program execution, such as divide-by-zero or accessing out-of-bounds array indices.
   * **Example**: int a = 10 / 0;

**Handling**: These are typically handled by the **runtime environment** (e.g., throwing exceptions or crashing the program), but the compiler might insert code to check for such conditions in certain languages.

**Example of Error Handling in a C Compiler**

intx = 5;

y = 10;

* **Lexical error**: If y contained invalid characters (e.g., @y = 10;), it would result in a lexical error.
* **Syntax error**: Missing semicolon or an unmatched bracket.
* **Semantic error**: y is undeclared, which would lead to an undeclared variable error.

error: 'y' undeclared (first use in this function)

**Note**: This allows the programmer to fix the problem before proceeding further.

**Compiler Design Tools**

Some of the tools used in compiler construction include:

* **Lex**: A tool used to generate lexical analyzers (scanners).
* **Yacc/Bison**: Tools for generating parsers based on context-free grammars.
* **LLVM(Low-Level Virtual Machine)**: A collection of modular and reusable compiler and toolchain technologies used to create intermediate representations, code optimizations, and code generation.