Parul® University
Vadodara, Gujarat

NAAC GRADE A++

Information and Communication Technology

Overview of compilation

Study Guide

Prof. Arpita Vaidya
CSE, PIT
Parul University

1.1 Introduction to Compiler

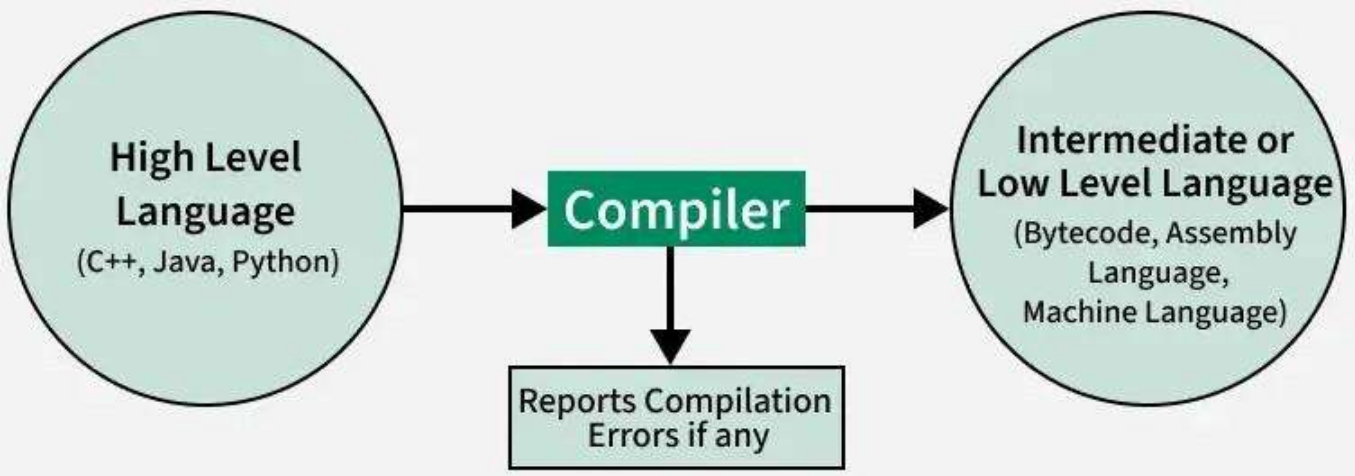
1. Introduction to Compiler

A compiler is a program that translates source code written in a high-level programming language (like C, C++, or Java) into machine code (object code) that can be executed by a computer's CPU.

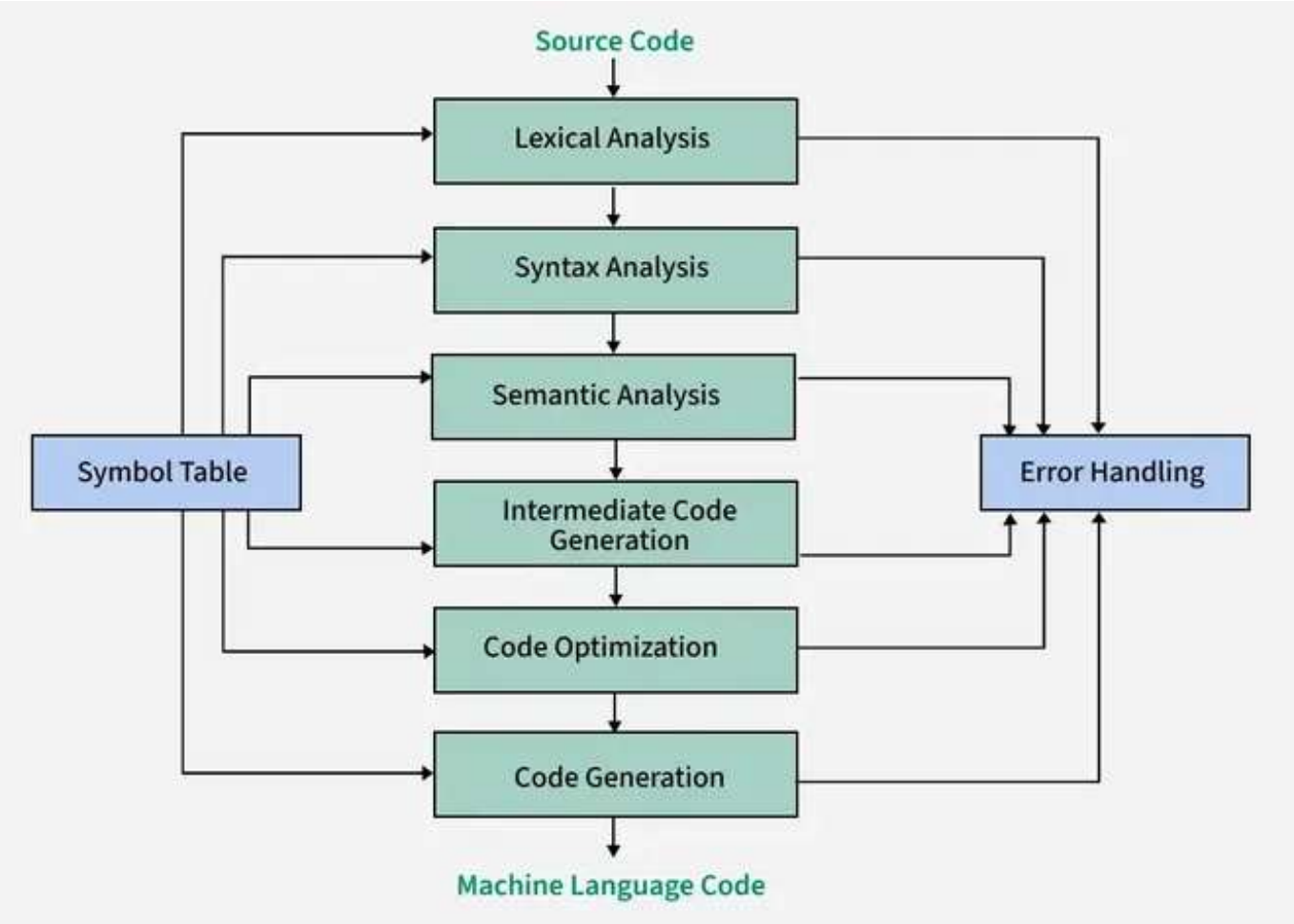
Example:

C source code → Compiler → Machine code (Executable file)

1. A compiler is software that translates or converts a program written in a high-level language (Source Language) into a low-level language (Machine Language or Assembly Language). Compiler design is the process of developing a compiler.
2. The Key objectives of compiler design are to automate the translation process, check correctness of input code, and reporting errors in source code.
3. It acts as the "translator" of the programming world, bridging the gap between human-readable code and machine-understandable instructions.
4. It involves many stages like lexical analysis, syntax analysis (parsing), semantic analysis, code generation, optimization, etc.
5. Most of the early high level languages like C and C++ were compiled. However modern languages like Java and Python use both interpreter and compiler. Please note that compiler only translates but interpreter runs as well.



1.2. The structure of a compiler



1. Lexical Analysis
Lexical analysis is responsible for converting the raw source code into a sequence of tokens. A token is the smallest unit of meaningful data in a programming language.

The lexical analyzer scans the source code character by character, grouping these characters into meaningful units (tokens) based on the language's syntax rules.
These tokens can represent keywords, identifiers, constants, operators, or punctuation marks.
By converting the source code into tokens, lexical analysis simplifies the process of understanding and processing the code in later stages of compilation.
Example: int x = 10;

The lexical analyzer would break this line into the following tokens:

int - Keyword token (data type)
x - Identifier token (variable name)
= - Operator token (assignment operator)
10 - Numeric literal token (integer value)
; - Punctuation token (semicolon, used to terminate statements)
To know more about Lexical Analysis refer to this article - Lexical Analysis.

2. Syntax Analysis or Parsing
This phase ensures that the code follows the correct grammatical rules of the programming language.

Verifies that the sequence of tokens produced by the lexical analyzer is arranged in a valid way according to the language's syntax.
It checks whether the code adheres to the language's rules, such as correct use of operators, keywords, and parentheses.
If the source code is not structured correctly, the syntax analyzer will generate errors.
To represent the structure of the source code, syntax analysis uses parse trees or syntax trees.

Parse Tree: A parse tree is a tree-like structure that represents the syntactic structure of the source code. It shows how the tokens relate to each other according to the grammar rules. Each branch in the tree represents a production rule of the language, and the leaves represent the tokens.
Syntax Tree: A syntax tree is a more abstract version of the parse tree. It represents the hierarchical structure of the source code but with less detail, focusing on the essential syntactic structure. It helps in understanding how different parts of the code relate to each other.

3. Semantic Analysis
Semantic analysis is the phase of the compiler that ensures the source code makes sense logically.

- It checks whether the program has any semantic errors, such as type mismatches or undeclared variables.
- Checks the meaning of the program by validating that the operations performed in the code are logically correct.
- This phase ensures that the source code follows the rules of the programming language in terms of its logic and data usage
- Some key checks performed during semantic analysis include:
- Type Checking: The compiler ensures that operations are performed on compatible data types. For example, trying to add a string and an integer would be flagged as an error because they are incompatible types.
- Variable Declaration: It checks whether variables are declared before they are used. For example, using a variable that has not been defined earlier in the code would result in a semantic error.

4. Intermediate Code Generation
Intermediate code is a form of code that lies between the high-level source code and the final machine code.

- It is not specific to any particular machine, making it portable and easier to optimize.
- Intermediate code acts as a bridge, simplifying the process of converting source code into executable code.

The use of intermediate code plays a crucial role in optimizing the program before it is turned into machine code.

- **Platform Independence:** Since the intermediate code is not tied to any specific hardware, it can be easily optimized for different platforms without needing to recompile the entire source code. This makes the process more efficient for cross-platform development.
- **Simplifying Optimization:** Intermediate code simplifies the optimization process by providing a clearer, more structured view of the program. This makes it easier to apply optimization techniques such as:
    - **Dead Code Elimination:** Removing parts of the code that don’t affect the program’s output.

- **Loop Optimization:** Improving loops to make them run faster or consume less memory.
- **Common Subexpression Elimination:** Reusing previously calculated values to avoid redundant calculations.

- **Easier Translation:** Intermediate code is often closer to machine code, but not specific to any one machine, making it easier to convert into the target machine code. This step is typically handled in the back end of the compiler, allowing for smoother and more efficient code generation.

  *Example: a = b + c * d;*
  *t1 = c * d*
  t2 = b + t1
  a = t2

5. Code Optimization

Code Optimization is the process of improving the intermediate or target code to make the program run faster, use less memory, or be more efficient, without altering its functionality.

- Involves techniques like removing unnecessary computations, reducing redundancy, and reorganizing code to achieve better performance.
- Optimization is classified broadly into two types Machine-Independent & Dependent

Common Techniques:

- **Constant Folding**: Precomputing constant expressions.
- **Dead Code Elimination**: Removing unreachable or unused code.
- **Loop Optimization**: Improving loop performance through invariant code motion or unrolling.
- **Strength Reduction**: Replacing expensive operations with simpler ones.

Example:

Code Before Optimization	Code After Optimization

for (int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
	x = y + z ;
	for (int j = 0 ; j < n ; j ++)
	{
	a[j] = 6 x j;
	}

s

6. Code Generation

Code Generation is the final phase of a compiler, where the intermediate representation of the source program (e.g., three-address code or abstract syntax tree) is translated into machine code or assembly code.

The source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

1.3. Applications of compiler technology

Compiler technology is used in **many domains** beyond programming languages:

Application Area	Description / Example
1. Programming Languages	Traditional compilers for C, C++, Java, Python
2. Database Systems	SQL queries translated into query plans using compiler techniques
3. Operating Systems	System software like assemblers and loaders use compiler design principles
4. AI / Machine Learning	Model optimization and execution (e.g., TensorFlow XLA, PyTorch JIT)
5. Just-In-Time (JIT) Compilers	Used in Java Virtual Machine (JVM), .NET CLR for runtime compilation
6. Embedded Systems	Cross-compilers generate code for microcontrollers
7. Natural Language Processing (NLP)	Tokenization and parsing of human languages use lexical analysis concepts
8. Web Browsers	JavaScript engines (like V8, SpiderMonkey) use compiler and JIT principles

2. Summary of Key Concepts

A compiler translates high-level source code into machine code through systematic phases — analysis (front-end) and synthesis (back-end).

It performs lexical, syntax, and semantic analysis, optimization, and code generation.

Compiler technology is applied in programming, AI, databases, embedded systems, and NLP.

Next Steps

- Explore other basic functionality of tole of lexical analyzer.

References:

1. **Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007).**
   Compilers: Principles, Techniques, and Tools (2nd Edition).
   Pearson Education. *(Also known as "The Dragon Book")*
2. **Holub, A. I. (1990).**
   Compiler Design in C.
   Prentice-Hall of India.

Parul® University
Vadodara, Gujarat

NAAC GRADE A++

https://paruluniversity.ac.in/