# Compiler Design Notes

***1. Compiler Functionality***

- A compiler is software that converts a program written in a high-level language (source language) into a low-level language (object/target/machine language).
- It serves as a type of translator, transforming high-level programming language code into machine code or assembly language code.

***2. Terminology***

- The program written in a high-level language is called the *source program.*
- The program converted into a low-level language is referred to as an *object or target program.*

***3. Essential Role of Compilation***

- High-level language programs must undergo compilation before they can be executed.
- Each programming language has its own compiler, but the core functions of all compilers are similar.

***4. Compilation Stages***

- Compilation involves multiple stages, including:
  - *Lexical analysis:* Analyzing the program's basic elements, such as tokens.
  - *Syntax analysis:* Checking the program's structure against the language's grammar rules.
  - *Semantic analysis*: Ensuring that the program's meaning is correct.
  - *Code generation:* Producing the equivalent low-level code.
  - *Optimization:* Enhancing the generated code for efficiency.

***5. Comparison with Assembler***

- Compilers are more sophisticated and intelligent compared to assemblers.
- They validate various aspects of the code, including limits, ranges, and errors.

***6. Performance Considerations***

- Compilers are relatively slower and *consume* a *significant amount of memory.*
  - This is because they perform a full program translation and analysis.
- They can be categorized as self compilers (produce machine code for the same machine) or cross compilers (generate code for different machines).

***High Level Programming Language:***

- A HLPL is a language that has an abstraction of attributes of the computer. High-level programming is more convenient to the user when writing a program.

***Low Level Programming Language:***

- A LLPL is a language that does not require programming ideas or concepts \

***Stages of Compiler Design***

1. ***Lexical Analysis:*** The first stage of compiler design is lexical analysis, also known as scanning. In this stage, the compiler reads the source code character by character and breaks it down into a series of tokens, such as *keywords*, *identifiers*, and *operators*. These tokens are then passed on to the next stage of the compilation process.

2. ***Syntax Analysis:*** The second stage of compiler design is syntax analysis, also known as *parsing*. In this stage, the compiler checks the syntax of the source code to ensure that it conforms to the rules of the programming language. The compiler builds a parse tree, which is a hierarchical representation of the program's structure, and uses it to check for syntax errors.

3. ***Semantic Analysis:*** The third stage of compiler design is semantic analysis. In this stage, the compiler checks the meaning of the source code to ensure that it makes sense. The compiler performs *type checking,* which ensures that variables are used correctly and that operations are performed on compatible data types. The compiler also checks for other semantic errors, such as undeclared variables and incorrect function calls.

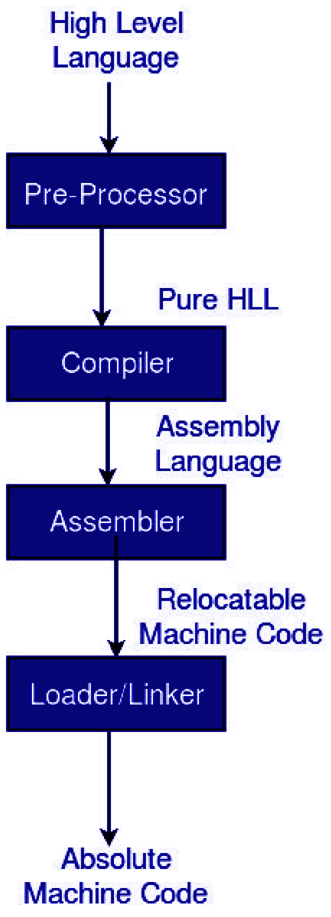
4. ***Code Generation:*** The fourth stage of compiler design is code generation. In this stage, the *compiler translates the parse tree into machine code* that can be executed by the computer. The code generated by the compiler must be efficient and optimized for the target platform.
5. ***Optimization:*** The final stage of compiler design is optimization. In this stage, the *compiler analyzes the generated code and makes optimizations to improve its performance.* The compiler may perform optimizations such as constant folding, loop unrolling, and function inlining.

- ***Cross Compiler*** that runs on a machine 'A' and produces a code for another machine 'B'. It is capable of creating code for a platform other than the one on which the compiler is running.

- ***Source-to-source*** Compiler or transcompiler or transpiler is a compiler that translates source code written in one programming language into the source code of another programming language.

## 1. ***Language Processing Systems***

- Computers consist of both software and hardware.
- Hardware understands a complex language, while humans prefer to write programs in high-level languages for ease of understanding and maintenance.
- Language processing systems help in transforming high-level language programs into a form usable by machines.



## 2. ***High-Level Language to Machine Code Conversion***

- High-Level Language (HLL): Programs with pre-processor directives like #include or #define are considered HLL. They are closer to human understanding but distant from machine comprehension.
- Pre-Processor: Removes #include and #define directives through file inclusion and macro expansion.
- Assembly Language: An intermediate state combining machine instructions and necessary data for execution.
- Assembler: Platform-specific; translates assembly language into machine code, generating an object file.

## 3. ***Interpreter vs. Compiler***

- Interpreter and compiler both convert high-level language into low-level machine language.
    - Difference: Interpreters process input line by line, whereas compilers read and translate the entire program at once.
    - Interpreted programs are generally slower than compiled ones.

## 4. ***Pre-Processor and Linker***

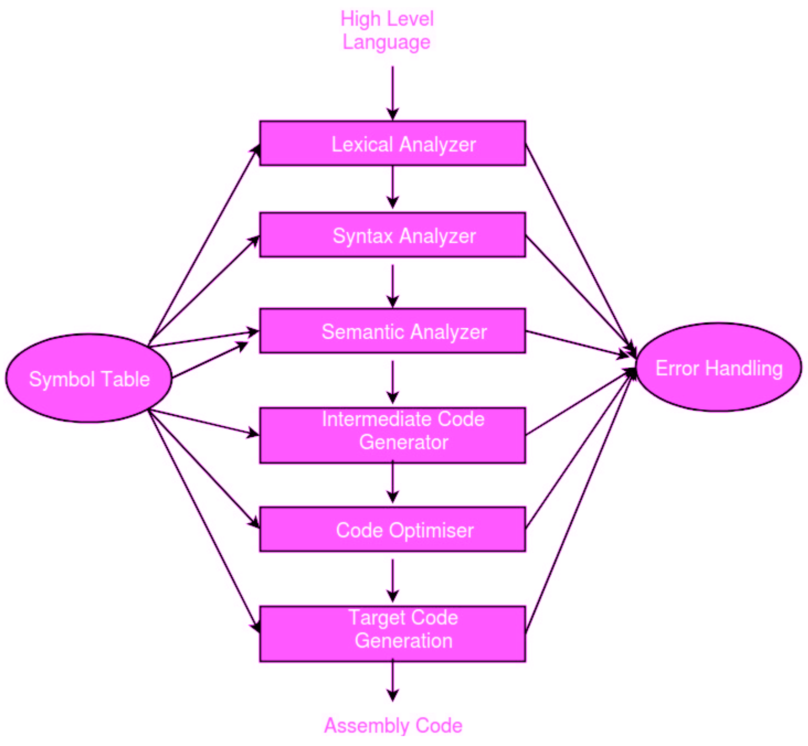
- Pre-Processor replaces directives like #include "Stdio.h" with their contents in the output.
- Linker's role: Merging object codes produced by the compiler, assembler, standard library functions, and operating system resources.
- These codes are relocatable, meaning their starting location in memory is not fixed.

## 5. ***Relocatable Machine Code***

- This code can be loaded at any point and run, as its addresses are flexible and can cooperate with program movements.

6. ***Loader/Linker***

- The loader/linker converts relocatable code into absolute code and attempts to run the program.
- Linker combines various object files into a single executable file, while the loader loads it into memory and executes it.



***Single Pass Compiler:***

- When all the phases of the compiler are present inside a single module, it is simply called a single-pass compiler. It performs the work of converting source code to machine code.

***Two Pass Compiler:***

- Two-pass compiler is a compiler in which the program is translated twice, once from the front end and the back from the back end known as Two Pass Compiler.

***Multipass Compiler:***

- When several intermediate codes are created in a program and a syntax tree is processed many times, it is called Multi pass Compiler. It breaks codes into smaller programs.

***Phases of a Compiler:***

There are two major phases of compilation, which in turn have many parts. Each of them takes input from the output of the previous level and works in a coordinated way.

***Analysis Phase***

- Intermediate representation created from source code.
- Components:
- Lexical Analyzer: Divides the program into "tokens."
- Syntax Analyzer: Recognizes program "sentences" based on language syntax.
- Semantic Analyzer: Checks static semantics of program constructs.

- Intermediate Code Generator: Generates "abstract" code.

***Synthesis Phase:***
- Creation of an equivalent target program from intermediate representation.

***Comprises:***
- Code Optimizer: Optimizes the abstract code.
- Code Generator: Translates abstract intermediate code into specific machine instructions.

***Operations of Compiler:***
- Compiler functions include:
    - Breaking source programs into smaller parts.
    - Building symbol tables and intermediate representations.
    - Facilitating code compilation and error detection.
    - Saving all codes and variables.
    - Comprehensive analysis and translation of the full program.
    - Converting source code to machine code.

***Advantages of Compiler Design:***
- Efficiency: Compiled programs are typically more efficient than interpreted ones due to optimized machine code for specific hardware.
- Portability: Compiled code can run on compatible hardware and operating systems, enhancing portability.
- Error Checking: Compilers identify syntax, semantic, and logical errors before execution.
- Optimizations: Compilers can optimize code for better performance.

***Disadvantages of Compiler Design:***
- Longer Development Time: Developing a compiler is complex and time-consuming.
- Debugging Difficulties: Debugging compiled code can be challenging as machine code may be less readable.
- Lack of Interactivity: Compiled programs require compilation before running, slowing down development and testing.
- Platform-Specific Code: Code generated by platform-specific compilers may not be portable to other platforms.

# Compiler Construction Tools:

***Compiler Construction Tools***

Key for Compiler Development…

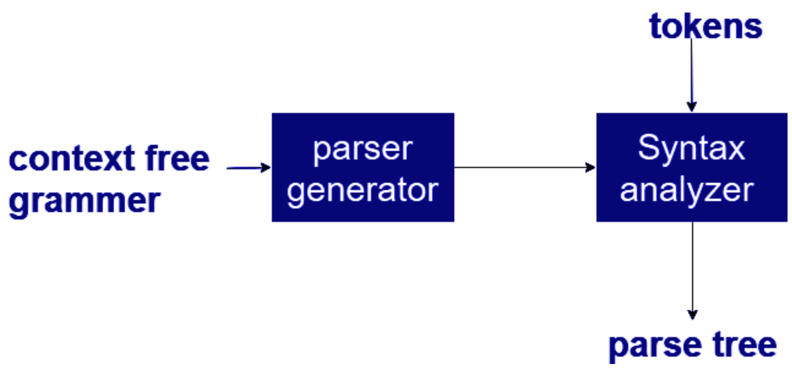
Specialized tools crucial for creating compilers and their components.

***Common Compiler Construction Tools***

1. ***Parser Generator:***

- Produces syntax analyzers from grammatical descriptions or context-free grammars.
- Simplifies complex syntax analysis, saving time.

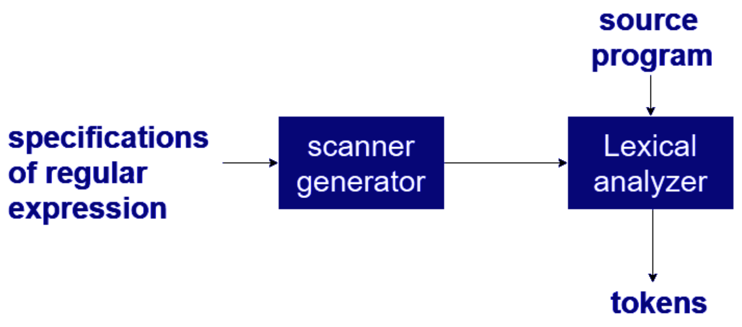
Examples: PIC, EQM.



2. ***Scanner Generator:***

- Generates lexical analyzers from regular expressions.
- Employs finite automata to recognize tokens.

Example: Lex.



3. ***Syntax Directed Translation Engines:***

- Generate intermediate code in three-address format from parse trees.

- Traverse parse trees with routines to create intermediate code.
- Parse tree nodes linked to one or more translations.

4. ***Automatic Code Generators:***
- Transforms intermediate language operations into machine language.
- Utilizes rules and templates to replace intermediate language statements.

5. ***Data-Flow Analysis Engines:***
- Used for code optimization.
- Gathers data flow information within the program.

6. ***Compiler Construction Toolkits:***
- Offer integrated routines for compiler component and phase construction.

***Features of Compiler Construction Tools***
- Lexical Analyzer Generator:
- Creates lexical analyzers based on regular expressions.

***Essential for tokenizing source code.***
- ***Parser Generator:***
    - Produces parsers from context-free grammars.
    - Builds abstract syntax trees.

- ***Code Generation Tools:***
    - Generate target code from abstract syntax trees.
    - Crucial for generating machine code.

- ***Optimization Tools:***
    - Enhance generated code for efficiency.
    - Perform optimizations like dead code elimination.

- ***Debugging Tools:***
    - Aid in debugging compilers and compiled programs.
    - Provide debugging information such as symbol tables.

- ***Profiling Tools:***
    - Profile compilers and code for performance optimization.
    - Identify and address performance bottlenecks.

- ***Documentation Tools:***

    - Generate documentation for the compiler and programming language.
    - Document syntax, semantics, and usage of the language.

- ***Language Support:***
    - Designed for various programming languages, from high-level to low-level.

- ***Cross-Platform Support:***
    - Some tools work across multiple platforms, enhancing versatility.

- ***User Interface:***
    - User-friendly interfaces make working with tools more efficient and accessible.

## Symbol Table in Compiler

***Symbol Table Definition***

- Symbol table consists of Name and Value pairs.
- Vital data structure for the compiler, tracking variable semantics.
- Stores scope and binding information, entity instances (e.g., variables, functions, classes).

***Symbol Table Usage***

- Integral to compiler in ensuring compile-time efficiency.
- Populated during lexical and syntax analysis phases.
- Information collected in analysis phases used for code generation in synthesis phases.

***Role in Compiler Phases***

- Lexical Analysis:
    - Creates table entries (e.g., tokens).

- Syntax Analysis:
    - Adds attributes like type, scope, dimension, line of reference, and use.

- Semantic Analysis:
    - Utilizes table information to verify semantic correctness (type checking) of expressions and assignments.

- Intermediate Code Generation:

    - References symbol table for runtime allocation and temporary variable data.

- Code Optimization:
    - Uses symbol table data for machine-dependent optimizations.

- Target Code Generation:
    - Generates code using identifier address information from the table.

***<u>Symbol Table Entries</u>***

- Each entry in the symbol table contains attributes supporting different compiler phases.

***<u>Use of Symbol Table</u>***

- Primarily used in compilers to store identifiers from the scanned application program.
- Stores identifiers with name, value, address, and data type.
- Tracks all necessary information about identifiers.

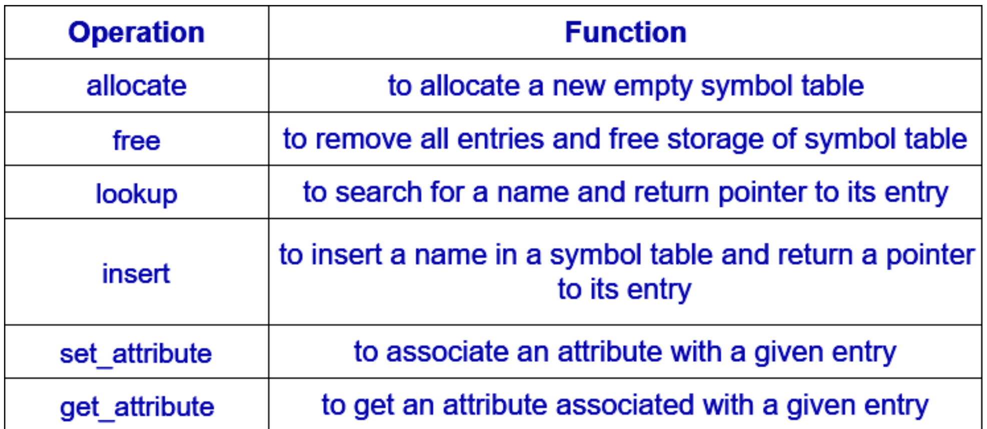
***<u>Items Stored in Symbol Table</u>***

- Variable names and constants.
- Procedure and function names.
- Literal constants and strings.
- Compiler-generated temporaries.
- Labels in source languages.

***<u>Information from Symbol Table</u>***

- Data type and name.
- Declaring procedures.
- Storage offset.
- For structures or records, a pointer to structure table.
- Parameter passing information (by value or reference).
- Number and types of arguments passed to functions.
- Base address.

Operations of Symbol table – The basic operations defined on a symbol table include:



| Operation | Function |
|---|---|
| allocate | to allocate a new empty symbol table |
| free | to remove all entries and free storage of symbol table |
| lookup | to search for a name and return pointer to its entry |
| insert | to insert a name in a symbol table and return a pointer to its entry |
| set_attribute | to associate an attribute with a given entry |
| get_attribute | to get an attribute associated with a given entry |

***Operations on Symbol Table***

1. Insertion: Add an item to the symbol table.
2. Deletion: Remove an item from the symbol table.
3. Searching: Find an item in the symbol table.

***Common Data Structures for Implementation***

1. List:
    - Stores names and associated information in an array.
    - Names added in order of appearance.
    - Lookup from the end to the beginning.
    - Fast insertion (O(1)), slow lookup for large tables (O(n)).

2. Linked List:
    - Uses linked list implementation.
    - Fast insertion (O(1)), slow lookup for large tables (O(n)).

3. Hash Table:
    - Combines a hash table and symbol table.
    - Quick insertion and lookup (O(1)).
    - Requires a hash function for searching.
    - Offers fast search but can be complex to implement.

4. Binary Search Tree:
    - Utilizes a binary search tree with left and right child links.
    - Maintains a tree structure following binary search tree properties.
    - Average insertion and lookup are $O(\log_2 n)$.

***Advantages of Symbol Table***

- Enhances program efficiency by offering quick access to crucial data.
- Organizes and simplifies code, making it easier to understand and troubleshoot.
- Speeds up code execution by optimizing memory access.
- Increases code portability and facilitates code reuse.

- Aids in debugging by providing access to the program's state during execution.

***<u>Disadvantages of Symbol Table</u>***

- Increased memory consumption in systems with limited resources.
- Processing time for creating and managing symbol tables can be long.
- Complexity in construction and maintenance, especially for those not well-versed in compiler design.
- Limited scalability for large-scale projects with vast data management needs.
- Regular upkeep can be resource-intensive.
- May not offer all required features, necessitating additional tools or libraries.

***<u>Applications of Symbol Table</u>***

- Resolving variable and function names, including data types and memory locations.
- Resolving scope issues and handling naming conflicts.
- Optimizing code execution by providing memory location information.
- Generating machine code from source code.
- Error checking and code debugging by offering program state details.
- Organizing and documenting code structure for improved readability.