**Assignment nO.1**

**COMPILER DESIGN**

**Answer Sheet**

1.What is language processor? list its types.

A **Language Processor** is a software tool that translates programs written in one language into another to make them executable by a computer. It is essential for converting **high-level language (HLL)** or **assembly language** into **machine code** that the processor can understand.

Types of Language Processors:

1.Compiler

Example: C, C++ compilers (GCC, Clang).

2. Interpreter

Example: Python, JavaScript interpreters.

3. Assembler

Example: NASM, MASM.

The Brief Explanation is :

**1. Compiler**

The language processor that reads the complete source program written in high-level language as a whole in one go and translates it into an equivalent program in machine language is called a Compiler.  Example: [C, C++](https://www.geeksforgeeks.org/difference-between-c-and-c/), C#.

In a compiler, the source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of the compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again the object program can be executed number of times without translating it again.



**2. Assembler**

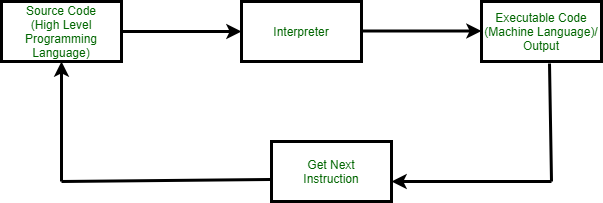
The Assembler is used to translate the program written in Assembly language into machine code. The source program is an input of an assembler that contains assembly language instructions. The output generated by the assembler is the object code or machine code understandable by the computer. Assembler is basically the 1st interface that is able to communicate humans with the machine. We need an assembler to fill the gap between human and machine so that they can communicate with each other. code written in assembly language is some sort of mnemonics(instructions) like ADD, MUL, MUX, SUB, DIV, MOV and so on. and the assembler is basically able to convert these mnemonics in binary code. Here, these mnemonics also depend upon the architecture of the machine.

For example, the [architecture of intel 8085](https://www.geeksforgeeks.org/architecture-of-8085-microprocessor/) and intel [8086](https://www.geeksforgeeks.org/architecture-of-8086/) are different.



**3. Interpreter**

The translation of a single statement of the source program into machine code is done by a language processor and executes immediately before moving on to the next line is called an interpreter. If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message. The interpreter moves on to the next line for execution only after the removal of the error. An Interpreter directly executes instructions written in a programming or [scripting language](https://www.geeksforgeeks.org/introduction-to-scripting-languages/) without previously converting them to an object code or machine code. An interpreter translates one line at a time and then executes it. Example: [Perl](https://www.geeksforgeeks.org/introduction-to-perl/), Python and [Matlab](https://www.geeksforgeeks.org/applications-of-matlab/).

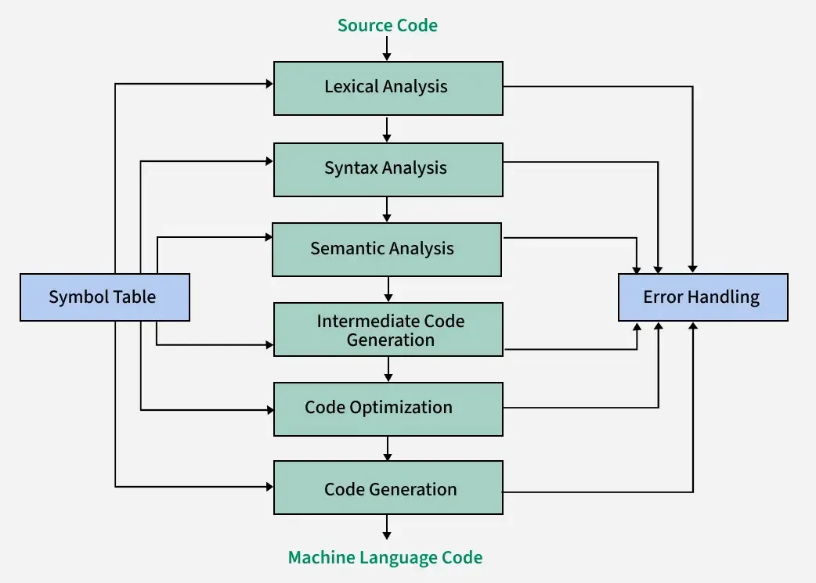


**Difference Between Compiler and Interpreter**

| **Compiler** | **Interpreter** |
| --- | --- |
| A compiler is a program that converts the entire source code of a[programming language](https://www.geeksforgeeks.org/introduction-to-programming-languages/)into executable machine code for a CPU. | An interpreter takes a source program and runs it line by line, translating each line as it comes to it. |
| The compiler takes a large amount of time to analyze the entire source code but the overall execution time of the program is comparatively faster. | An interpreter takes less amount of time to analyze the source code but the overall execution time of the program is slower. |
| The compiler generates the error message only after scanning the whole program, so debugging is comparatively hard as the error can be present anywhere in the program. | Its [Debugging](https://www.geeksforgeeks.org/software-engineering-debugging/) is easier as it continues translating the program until the error is met. |
| The compiler requires a lot of memory for generating object codes. | It requires less memory than a compiler because no object code is generated. |
| Generates intermediate object code. | No intermediate object code is generated. |
| For Security purpose compiler is more useful. | The interpreter is a little vulnerable in case of security. |
| Examples: C, C++, C# | Examples: Python, Perl, JavaScript, Ruby. |

2.Explain the phases of compiler.

A compiler is a software tool that converts high-level programming code into machine code that a computer can understand and execute. It acts as a bridge between human-readable code and machine-level instructions, enabling efficient program execution. The process of compilation is divided into six phases:



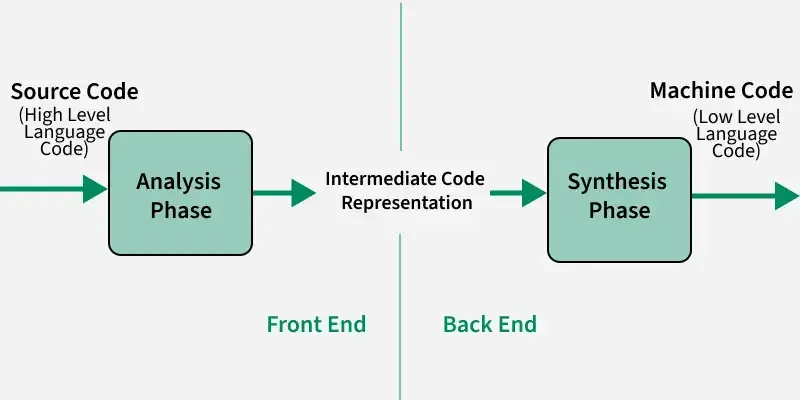
Dia: Language Processing System

Mnemonic trick : “Lazy Students Sleep In Cold Tent”

The whole compilation process is divided into two parts, front-end and **back-end**. These six phases are divided into two main parts, front-end and back-end with the intermediate code generation phase acting as a link between them. The front end analyzes source code for syntax and semantics, generating intermediate code, while ensuring correctness. The back end optimizes this intermediate code and converts it into efficient machine code for execution. The front end is mostly machine-independent, while the back end is machine-dependent.

The compilation process is an essential part of transforming high-level source code into machine-readable code. A compiler performs this transformation through several phases, each with a specific role in making the code efficient and correct. Broadly, the compilation process can be divided into two main parts:

1. **Analysis Phase:** The analysis phase breaks the source program into its basic components and creates an intermediate representation of the program. It is sometimes referred to as front end.
2. **Synthesis Phase:**The synthesis phase creates the final target program from the intermediate representation. It is sometimes referred to as back end.



**Brief Explanation Of Phases of a Compiler**

**1. Lexical Analysis**

Lexical analysis is the first phase of a compiler, 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. Lexical analysis involves scanning the source code, recognizing patterns, and categorizing groups of characters into distinct tokens.

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)*

Each of these tokens is then passed on to the next phase of the compiler for further processing, such as syntax analysis.

To know more about Lexical Analysis refer to this article – [*Lexical Analysis*](https://www.geeksforgeeks.org/introduction-of-lexical-analysis/).

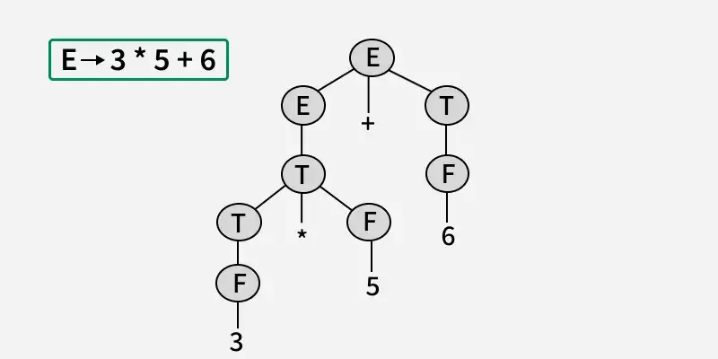
**2. Syntax Analysis**

Syntax analysis, also known as parsing, is the second phase of a compiler where the structure of the source code is checked. This phase ensures that the code follows the correct grammatical rules of the programming language.

The role of syntax analysis is to verify 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.



**. Semantic Analysis**

Semantic analysis is the phase of the compiler that ensures the source code makes sense logically. It goes beyond the syntax of the code and checks whether the program has any semantic errors, such as type mismatches or undeclared variables.

Semantic analysis 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.

*Example:*

*int a = 5;  
float b = 3.5;  
a = a + b;*

***Type Checking:***

* *a is int and b is float. Adding them (a + b) results in float, which cannot be assigned to int a.*
* ***Error:*** *Type mismatch: cannot assign float to int.*

To know more about Semantic Analysis refer to this article – [*Semantic Analysis*](https://www.geeksforgeeks.org/semantic-analysis-in-compiler-design/).

**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*

To know more about Intermediate Code Generation refer to this article – [*Intermediate Code Generation*](https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/).

**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. It involves techniques like removing unnecessary computations, reducing redundancy, and reorganizing code to achieve better performance. Optimization is classified broadly into two types:

* Machine-Independent
* Machine-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;  } |

To know more about Code Optimization refer to this article – [*Code Optimization*](https://www.geeksforgeeks.org/code-optimization-in-compiler-design/).

**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. This machine code is specific to the target platform and can be executed directly by the hardware.

The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. 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.

*Example:*

| **Three Address Code** | **Assembly Code** |
| --- | --- |
| **t1 = c \* d**  **t2 = b + t1**  **a = t2** | LOAD R1, c ; Load the value of ‘c’ into register R1  LOAD R2, d ; Load the value of ‘d’ into register R2  MUL R1, R2 ; R1 = c \* d, store result in R1  LOAD R3, b ; Load the value of ‘b’ into register R3  ADD R3, R1 ; R3 = b + (c \* d), store result in R3  STORE a, R3 ; Store the final result in variable ‘a’ |

**Symbol Table –**It is a data structure being used and maintained by the compiler, consisting of all the identifier’s names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.

To know more about Symbol Table refer to this article – [*Symbol Table*](https://www.geeksforgeeks.org/symbol-table-compiler/).

**Error Handling in Phases of Compiler**

Error Handling refers to the mechanism in each phase of the compiler to detect, report and recover from errors without terminating the entire compilation process.

* **Lexical Analysis**:Detects errors in the character stream and ensures valid token formation.
  + **Example**: Identifies illegal characters or invalid tokens (e.g., @var as an identifier).
* **Syntax Analysis:**Checks for structural or grammatical errors based on the language’s grammar.
  + **Example**: Detects missing semicolons or unmatched parentheses.
* **Semantic Analysis**: Verifies the meaning of the code and ensures it follows language semantics.
  + **Example**: Reports undeclared variables or type mismatches (e.g., adding a string to an integer).
* **Intermediate Code Generation**: Ensures the correctness of intermediate representations used in further stages.
  + **Example**: Detects invalid operations, such as dividing by zero.
* **Code Optimization**: Ensures that the optimization process doesn’t produce errors or alter code functionality.
  + **Example**: Identifies issues with unreachable or redundant code.
* **Code Generation**: Handles errors in generating machine code or allocating resources.
  + **Example**: Reports insufficient registers or invalid machine instructions.

3. Describe the working of symbol table, error handler and parser as the components of a compiler.

**Symbol Table in Compiler**

Every compiler uses a symbol table to track all variables, functions, and identifiers in a program. It stores information such as the name, type, scope, and memory location of each identifier. Built during the early stages of compilation, the symbol table supports error checking, scope management, and code optimization for runtime efficiency. It plays a crucial role in ensuring the correct usage of identifiers according to language rules.

**Role of Symbol Table in Compiler Phases**

The symbol table acts as a bridge between the analysis and synthesis phases of the compiler. It collects information during the analysis phases and utilizes it during the synthesis phases to generate efficient code, ultimately enhancing compile-time performance.

**It is used by various**[**phases of the compiler**](https://www.geeksforgeeks.org/phases-of-a-compiler/)**as follows:-**

* **Lexical Analysis:** Creates new table entries in the table, for example, entries about tokens.
* **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
* **Semantic Analysis:** Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.
* **Intermediate Code Generation:** Refers to symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
* **Code Optimization:** Uses information present in the symbol table for machine-dependent optimization.
* **Target Code generation:** Generates code by using the address information of the identifier present in the table.
* **Symbol Table entries –** Each entry in the symbol table is associated with attributes that support the compiler in different phases.

**Example of Using Symbol Table**

Imagine a program that includes a series of mathematical expressions, such as:

* A variable distance representing the distance traveled.
* A constant pi representing the value of Pi.
* A function calculateArea that computes the area of a circle.

| **Name** | **Type** | **scope** | **Memory address** | **value** | **Additional Info** |
| --- | --- | --- | --- | --- | --- |
| **distance** | variable | Global | 0x1000 | Uninitialized | Data type: float |
| **pi** | constant | Global | 0x1004 | 3.14159 | Data type: float, read-only |
| **calculateArea** | function | Global | 0x1008 | N/A | Return type: float |
| **radius** | parameter | Local | 0x2000 | 0x1000 | Data type: float |

**In this example:**

* The symbol table records that distance is a global variable of type float that has not been initialized.
* Pi is a global value of type float with a constant value of 3.14159 and is marked as read-only.
* It registers the function calculateArea that returns a value of type float.
* The parameter radius is declared as a local variable – in the scope of the function – also of type float.
* It is this organization that serves the compiler when it does various tasks, such as checking for type errors, optimization of code, knowing the value of pi because it is a constant, declaring and using variables according to its scope.

**Items stored in Symbol table**

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

**Information used by the compiler from Symbol table**

* Data type and name
* Declaring procedures
* Offset in storage
* If structure or record then, a pointer to structure table.
* For parameters, whether parameter passing by value or by reference
* Number and type of arguments passed to function
* Base Address

**Operations of Symbol table**

The basic operations defined on a symbol table include

Operations of Symbol Table

**Operations on Symbol Table**

Following operations can be performed on symbol table-

* Insertion of an item in the symbol table.
* Deletion of any item from the symbol table.
* Searching of desired item from symbol table.

**Implementation of Symbol table**

Following are commonly used data structures for implementing symbol table:

**List**

We use a single array or equivalently several arrays, to store names and their associated information ,New names are added to the list in the order in which they are encountered . The position of the end of the array is marked by the pointer available, pointing to where the next symbol-table entry will go.  The search for a name proceeds backwards from the end of the array to the beginning. when the name is located the associated information can be found in the words following next.

| id1 | info1 | id2 | info2 | …….. | id\_n | info\_n |
| --- | --- | --- | --- | --- | --- | --- |

* In this method, an array is used to store names and associated information.
* A pointer**“available”**is maintained at end of all stored records and new names are added in the order as they arrive
* To search for a name we start from the beginning of the list till available pointer and if not found we get an error**“use of the undeclared name”**
* While inserting a new name we must ensure that it is not already present otherwise an error occurs i.e.**“Multiple defined names”**
* Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
* The advantage is that it takes a minimum amount of space.

**Linked List**

* This implementation is using a linked list. A link field is added to each record.
* Searching of names is done in order pointed by the link of the link field.
* A pointer**“First”**is maintained to point to the first record of the symbol table.
* Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

**Hash Table**

* In hashing scheme, two tables are maintained – a hash table and symbol table and are the most commonly used method to implement symbol tables. A hash table is an array with an index range: 0 to table size – 1. These entries are pointers pointing to the names of the symbol table.
* To search for a name we use a hash function that will result in an integer between 0 to table size – 1.
* Insertion and lookup can be made very fast – O(1).
* The advantage is quick to search is possible and the disadvantage is that hashing is complicated to implement.

**Binary Search Tree**

* Another approach to implementing a symbol table is to use a binary search tree i.e. we add two link fields i.e. left and right child.
* All names are created as child of the root node that always follows the property of the binary search tree.
* Insertion and lookup are O(log 2 n) on average.

Please refer [C++ Program to implement Symbol Table](https://www.geeksforgeeks.org/cpp-program-to-implement-symbol-table) for implementation.

**Advantages of Symbol Table**

* The efficiency of a program can be increased by using symbol tables, which give quick and simple access to crucial data such as variable and function names, data kinds, and memory locations.
* better coding structure Symbol tables can be used to organize and simplify code, making it simpler to comprehend, discover, and correct problems.
* Faster code execution: By offering quick access to information like memory addresses, symbol tables can be utilized to optimize code execution by lowering the number of memory accesses required during execution.
* Symbol tables can be used to increase the portability of code by offering a standardized method of storing and retrieving data, which can make it simpler to migrate code between other systems or programming languages.
* Improved code reuse: By offering a standardized method of storing and accessing information, symbol tables can be utilized to increase the reuse of code across multiple projects.
* Symbol tables can be used to facilitate easy access to and examination of a program’s state during execution, enhancing debugging by making it simpler to identify and correct mistakes.

**Disadvantages of Symbol Table**

* **Increased memory consumption:** Systems with low memory resources may suffer from symbol tables’ high memory requirements.
* **Increased processing time:** The creation and processing of symbol tables can take a long time, which can be problematic in systems with constrained processing power.
* **Complexity:** Developers who are not familiar with compiler design may find symbol tables difficult to construct and maintain.
* **Limited scalability:** Symbol tables may not be appropriate for large-scale projects or applications that require o the management of enormous amounts of data due to their limited scalability.
* **Upkeep:** Maintaining and updating symbol tables on a regular basis can be time- and resource-consuming.
* **Limited functionality:** It’s possible that symbol tables don’t offer all the features a developer needs, and therefore more tools or libraries will be needed to round out their capabilities.

**Applications of Symbol Table**

* **Resolution of variable and function names:** Symbol tables are used to identify the data types and memory locations of variables and functions as well as to resolve their names.
* **Resolution of scope issues:** To resolve naming conflicts and ascertain the range of variables and functions, symbol tables are utilized.
* Symbol tables, which offer quick access to information such as memory locations, are used to optimize code execution.
* **Code generation:** By giving details like memory locations and data kinds, symbol tables are utilized to create machine code from source code.
* **Error checking and code debugging:** By supplying details about the status of a program during execution, symbol tables are used to check for faults and debug code.
* **Code organization and documentation:** By supplying details about a program’s structure, symbol tables can be used to organize code and make it simpler to understand.

**Error Handling in Compiler Design**

Last Updated : 04 Feb, 2025

During the process of language translation, the compiler can encounter errors. While the compiler might not always know the exact cause of the error, it can detect and analyze the visible problems. The main purpose of error handling is to assist the programmer by pointing out issues in their code. Error handling doesn’t happen often compared to other tasks in the compiler, so the time it takes to fix errors isn’t a major concern.

Error handlers are specialized programs designed to manage errors in applications. The best error handlers prevent errors if possible, recover from them without closing the application, or shut down the affected application and save the error details in a log file.

In programming, some errors can be prevented. These errors can occur during the syntax phase (such as typing mistakes or wrong use of special characters like semicolons) or the logical phase (when the code doesn’t produce the expected results, also called bugs). Syntax errors are usually caught by proofreading the code, while logical errors are best handled through thorough debugging.

Runtime errors occur while a program is running, often due to issues like insufficient memory or invalid input data, such as a memory conflict.

*Error handler = Error Detection + Error Report + Error Recovery.*

**Sources of Error in Error Handling**

* An Error is the blank entries in the symbol table. Errors in the program should be detected and reported by the parser.
* Whenever an error occurs, the [parse](https://www.geeksforgeeks.org/introduction-of-parsing-ambiguity-and-parsers-set-1/)r can handle it and continue to parse the rest of the input.
* Although the parser is mostly responsible for checking for errors, errors may occur at various stages of the compilation process.
* Error handling is a process that helps identify errors in a program, reports them to the user, and then applies recovery strategies to manage the errors. During this process, it’s important that the program’s processing time is not slowed down too much. One common error source is blank entries in the symbol table.

There are two main types of errors:

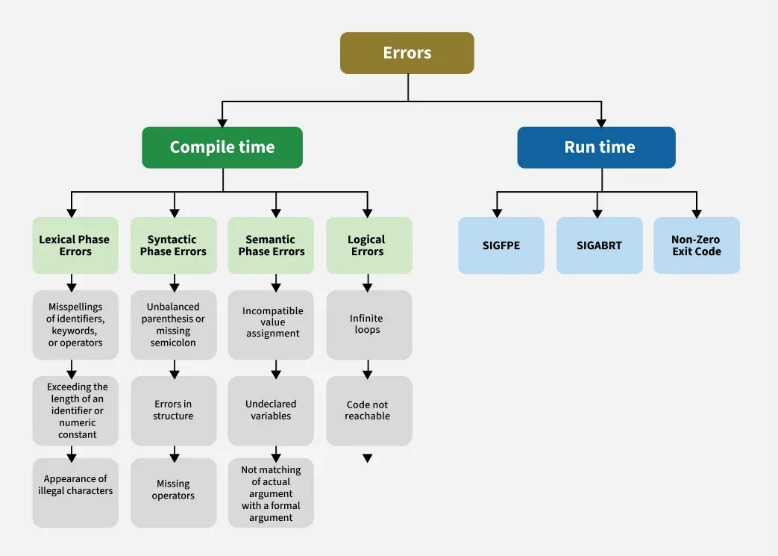
1. **Run-Time Errors**

These errors happen while the program is running. They usually occur because of issues like incorrect system settings or invalid input data. Examples of run-time errors include:

* **Lack of memory** to run the program.
* **Memory conflicts** with other programs.
* **Logical errors**, where the program doesn’t produce the expected results. These can be fixed by carefully debugging the code.

**2. Compile-Time Errors**

These errors occur before the program starts running, during the compilation process. They stop the program from compiling successfully.  
Examples of compile-time include:

* **Syntax errors** (like missing semicolons or incorrect statements).
* **Missing file references** that prevent the program from compiling.
* 

**Finding error or reporting an error**

Viable-prefix is the property of a parser that allows early detection of [syntax errors](https://www.geeksforgeeks.org/what-is-a-syntax-error-and-how-to-solve-it/).

* **Goal** detection of an error as soon as possible without further consuming unnecessary input
* **How:** detect an error as soon as the prefix of the input does not match a prefix of any string in the language.

**Example:** for(**;**), this will report an error as for having two semicolons inside braces.

**Error Recovery**

There are several methods that a compiler uses to recover from errors. These methods help the compiler continue processing the code instead of stopping immediately.  
Common recovery methods include:

* **Panic Mode Recovery** – Skips erroneous code and resumes from the next valid statement.
* **Phase-Level Recovery** – Replaces small incorrect code segments with valid ones.
* **Error Productions** – Recognizes common errors and provides specific suggestions.
* **Global Correction** – Makes multiple changes to fix errors optimally.

For a detailed explanation, refer to [Error detection and Recovery in Compiler](https://www.geeksforgeeks.org/error-detection-recovery-compiler/)

**Advantages of Error Handling in Compiler Design**

1. **Robustness** – Ensures the compiler can handle errors smoothly without crashing, allowing it to continue processing and provide meaningful error messages.
2. **Error Detection** – Identifies various errors like syntax, semantic, and type errors to prevent unexpected program behavior.
3. **Error Reporting** – Provides clear and precise error messages, helping developers quickly locate and fix issues.
4. **Error Recovery** – Tries to fix or bypass errors so the compilation process can continue instead of stopping abruptly.
5. **Incremental Compilation** – Allows compiling and testing correct sections of code even if other parts contain errors, useful for large projects.
6. **Efficiency** – Saves time by reducing debugging effort with accurate error messages and recovery mechanisms.
7. **Language Development** – Helps define and enforce language rules, improving reliability and consistency in programming.

**Disadvantages of error handling in compiler design**

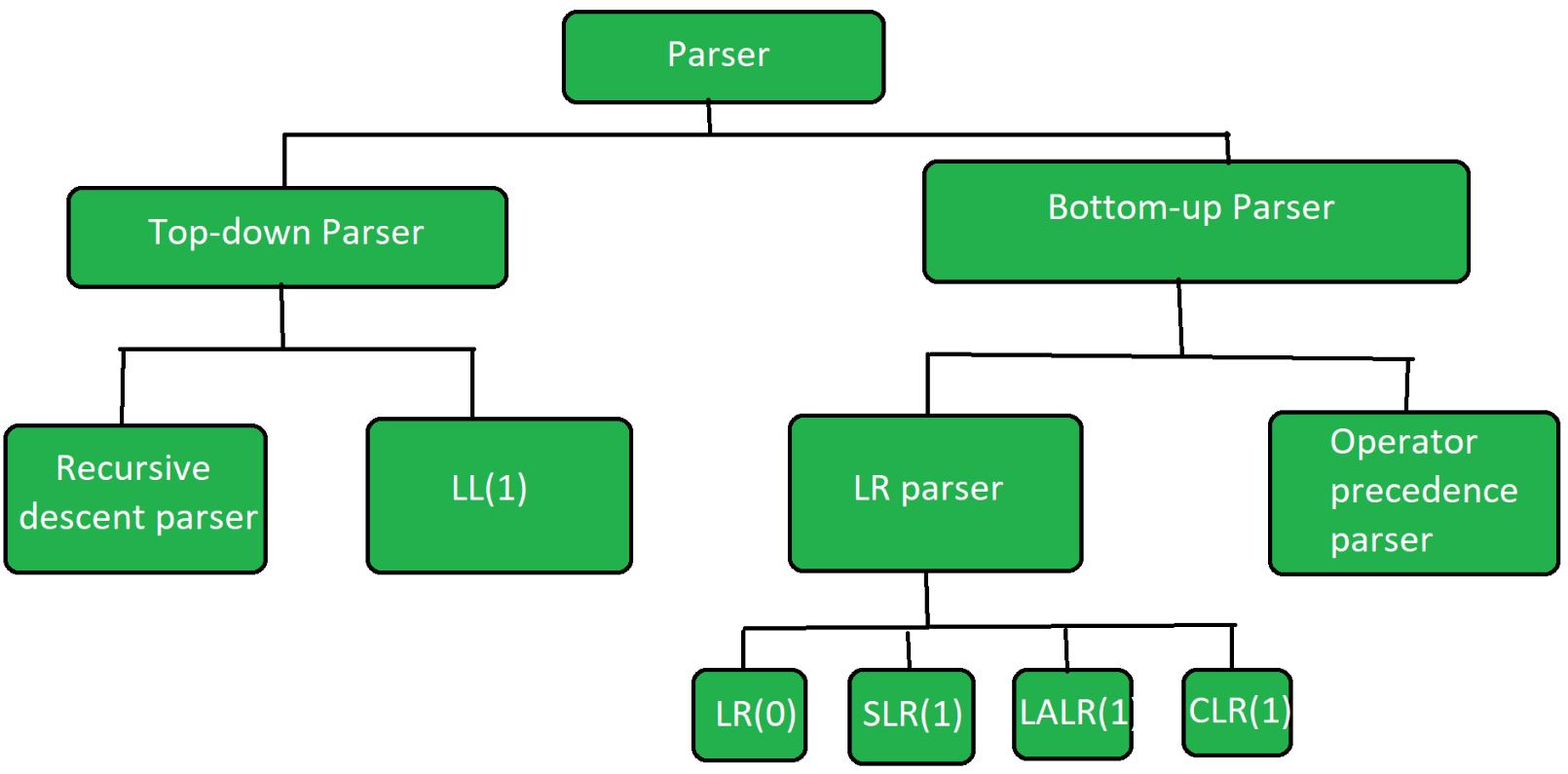
1. **Increased Complexity** – Makes the compiler harder to develop, test, and maintain due to the complexity of handling various errors.
2. **Reduced Performance** – Error handling can slow down compilation if it is computationally intensive or requires additional processing.
3. **Longer Development Time** – Implementing and testing an effective error handling system takes time, delaying the compiler’s development.
4. **Difficulty in Error Detection** – Some errors may be masked by the error handling system, making them harder to detect or debug.

**Types of Parsers in Compiler Design**

Compiler design has many functional modules one of them is the parser which takes the output of the lexical analyzer (often a set of tokens list) and builds a parse tree. The main responsibility of the parser is to confirm whether the generated language can produce the input string and helps in the analysis of syntax.

**What is a Parser?**

The parser is one of the phases of the compiler which takes a token of string as input and converts it into the corresponding **Intermediate Representation (IR)** with the help of an existing grammar. The parser is also known as Syntax Analyzer.



**Types of Parsers**

The parser is mainly classified into two categories.

1. Top-down Parser
2. Bottom-up Parser

**Top-Down Parser**

[Top-down parser](https://www.geeksforgeeks.org/working-of-top-down-parser/?ref=header_ind)is the parser that generates parse tree for the given input string with the help of grammar productions by expanding the non-terminals. It starts from the start symbol and ends down on the terminals. It uses left most derivation.

Further Top-down parser is classified into 2 types: 1) Recursive descent parser and 2) non-recursive descent parser.

1. [Recursive descent parser](https://www.geeksforgeeks.org/recursive-descent-parser/?ref=header_ind) is also known as the Brute force parser or the backtracking parser. It basically generates the parse tree by using brute force and backtracking techniques.
2. Non-recursive descent parser is also known as [LL(1) parser](https://www.geeksforgeeks.org/construction-of-ll1-parsing-table/?ref=header_ind) or predictive parser or without backtracking parser or dynamic parser. It uses a parsing table to generate the parse tree instead of backtracking.

**Bottom-Up Parser**

[Bottom-up Parser](https://www.geeksforgeeks.org/working-of-bottom-up-parser/?ref=header_ind) is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the terminals. It starts from terminals and ends upon the start symbol. It uses the rightmost derivation in reverse order.

Bottom-up parser is classified into two types:

LR parser: This is a bottom-up parser that generates the parse tree for the given string by using unambiguous grammar. It follows the reverse of the rightmost derivation. [LR parser](https://www.geeksforgeeks.org/lr-parser/) is classified into four types:

* LR(0)
* SLR(1)
* LALR(1)
* CLR(1)

Operator precedence parser: [Operator Precedence Parser](https://www.geeksforgeeks.org/role-of-operator-precedence-parser/) generates the parse tree from given grammar and string but it has the condition that two consecutive non-terminals and epsilon will never appears on the right-hand side of any production. The operator precedence parsing technique is applied to Operator grammars.

**What is Operator Grammar?**

An operator precedence grammar is a context-free grammar that has the property that no production has either:

* An empty on the right-hand side
* Two adjacent non-terminals in its right-hand side.

**Conclusion**

Parser is one of the important phases in compiler design that takes a token of string as input and converts into the intermediate representation. If the programming code does not follow the rules, then the compiler will generate an error message and the compilation process will be stopped. Parser is classified into two types namely Top-down Parser and Bottom-up Parser. Based on the requirements and the goals of the programming language the appropriate parsing technique is used.

4. Explain the role of Lexical analyzer in compiler designing.

5.What is scanner generator? Explain in detail.

6.Illustate the role of lexical analyzer in compilation process & justify why it is implemented as a separate phase.

7.Design explain the structure of a language processor in brief. [5]

8. List down and elaborate the commonly used compiler construction tools. [5]

9. Describe the interaction between the Lexical analyzer and the parser in detail. [5]

10. Consider the grammar G: S S +S |S\*S|(S)a. show that the string “a+a\*a” has parse trees and grammar is ambiguous or not. [5]

11. Write short note on: a) top -down parsing and predictive parsing b) Tree traversal