**Unit I & II & V**

**Overview of System Software and Text Editors:**

**System Software-**

* Operating System
* Language Translators
* Device Drivers
* Firmware
* Macro Processor

**Systems Programming Tools-**

System programming tools are software utilities and applications designed to assist developers in creating, maintaining, debugging, and optimizing system-level software. These tools are particularly useful when working with low-level programming tasks, such as interacting with hardware, managing memory, handling interrupts, and optimizing performance. System programming tools are essential for operating system development, device driver creation, embedded systems programming, and other similar tasks. Here are some common system programming tools:

1. Compiler: A compiler is a fundamental tool used to translate high-level programming languages (e.g., C, C++, Rust) into machine code or an intermediate representation that can be executed directly by the computer's hardware.
2. Assembler: Assemblers are used to convert assembly language code (human-readable low-level language) into machine code. It directly maps to the computer's architecture, making it easier to work with specific hardware components.
3. Debugger: A debugger is a critical tool for finding and fixing errors in code. It allows developers to pause the program's execution, inspect variables, explore the call stack, and step through the code line-by-line for better understanding and bug fixing.
4. Profiling Tools: Profilers help analyze the performance of a program by identifying bottlenecks and areas that need optimization. They measure resource usage, such as CPU time, memory allocation, and I/O operations, to pinpoint performance issues. Profiling is the process of collecting program parameters while it is running. The execution duration and number of calls of specific functions and program code lines are measured during profiling. The programmer can use this tool to locate and optimize the slowest code portions.
5. Linker: The linker is responsible for combining compiled code modules and libraries into an executable binary.
6. Loader: Loaders are used to loading executable programs into memory for execution. They handle tasks such as memory allocation, relocation of code, and resolving external references.
7. Version Control Systems (VCS): While not exclusively for system programming, VCS like Git are invaluable for managing and tracking changes in the source code, especially in collaborative development environments.
8. System Call Tracers: These tools monitor the system calls made by a program to understand its interactions with the operating system. They are helpful in debugging and understanding the program's behavior.
9. Memory Debugging Tools: Tools like Valgrind help detect memory-related errors like memory leaks, illegal memory access, and uninitialized variables.
10. Performance Monitoring Tools: These tools provide real-time insights into system performance metrics such as CPU usage, *memory consumption*, disk I/O, and network activity.
11. Virtualization Software: Virtualization tools like VirtualBox and VMware are helpful for creating virtual machines to test and develop software on different operating systems or configurations.

Each system programming tool serves a specific purpose, and developers often use a combination of these tools to ensure the stability, efficiency, and correctness of their low-level code.

**Life Cycle of a Source Program-**

The life cycle of a source program refers to the stages it goes through from its creation to its execution. Let's break down the life cycle into various phases and illustrate them with examples-

1. **Creation/Editing Phase:**

In this phase, the source code of the program is written or edited by the developer using a text editor or integrated development environment (IDE). This code is typically written in a high-level programming language.

1. **Compilation Phase:**

In this phase, the source code is transformed into machine-readable code by a compiler or interpreter. Compilation is performed by translating the high-level code into low-level code or machine code. If the language uses a compiler, the source code is compiled into an intermediate code or directly into machine code. If the language uses an interpreter, the source code is translated line-by-line during execution.

1. **Linking/Loading Phase:**

For compiled languages, the resulting machine code may need to be linked with other libraries or object files to create an executable program. This is known as linking. For interpreted languages, the code is loaded into memory directly during runtime.

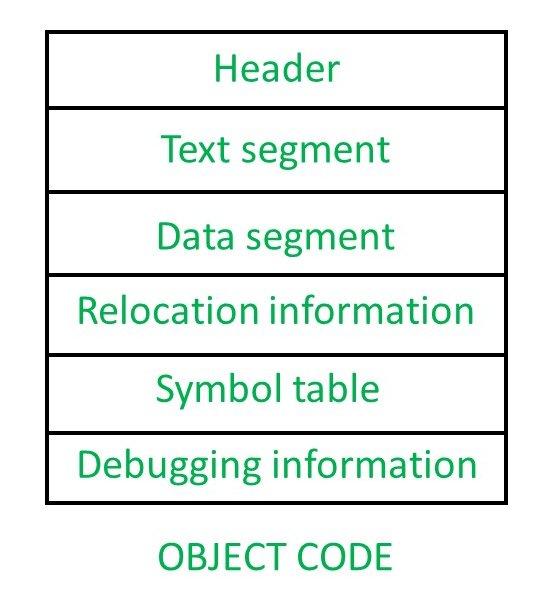
1. **Execution Phase:**

In this phase, the program is executed, and it performs the intended tasks as per the instructions provided in the source code. For interpreted languages, the code is executed directly line-by-line. For compiled languages, the executable program is run by the operating system, which follows the machine code instructions to perform the desired operations.

1. **Termination Phase:**

After the execution is complete, the program terminates, and any allocated resources are released. The result may be displayed to the user or stored in a file or database for future use.

This represents the basic life cycle of a source program, showcasing its journey from creation to execution. It's essential to note that this process can be more complex for larger projects involving multiple source files and dependencies. Additionally, modern software development often involves version control, testing, and debugging, which extend the life cycle further.



**Programming Errors (Debugging)**

Errors are the problems or the faults that occur in the program, which makes the behavior of the program abnormal, and experienced developers can also make these faults. Programming errors are also known as the bugs or faults, and the process of removing these bugs is known as debugging.

These errors are detected either during the time of compilation or execution. Thus, the errors must be removed from the program for the successful execution of the program.

There are mainly five types of errors exist in C programming:

* Syntax error
* Run-time error
* Linker error
* Logical error
* Semantic error

**Syntax error**

Syntax errors are also known as the compilation errors as they occurred at the compilation time, or we can say that the syntax errors are thrown by the compilers.

If we want to declare the variable of type integer,

int a; // this is the correct form

Int a; // this is an incorrect form.

Commonly occurred syntax errors are:

* If we miss the parenthesis (}) while writing the code.
* Displaying the value of a variable without its declaration.
* If we miss the semicolon (;) at the end of the statement.

**Run-time error**

Sometimes the errors exist during the execution-time even after the successful compilation known as run-time errors. The division by zero is the common example of the run-time error. These errors are very difficult to find, as the compiler does not point to these errors.

#include <stdio.h>

int main()

{

int a=2;

int b=2/0;

printf("The value of b is : %d", b);

return 0;

}

**Linker error**

Linker errors are mainly generated when the executable file of the program is not created. This can be happened either due to the wrong function prototyping or usage of the wrong header file. The most common linker error that occurs is that we use Main() instead of main().

#include <stdio.h>

int Main()

{

Sum();

int a=78;

printf("The value of a is : %d", a);

return 0;

}

**Logical error**

The logical error is an error that leads to an undesired output. These errors produce the incorrect output, but they are error-free, known as logical errors. These types of mistakes are mainly done by beginners. The occurrence of these errors mainly depends upon the logical thinking of the developer. If the programmers sound logically good, then there will be fewer chances of these errors.

**Semantic error**

Semantic errors are the errors that occurred when the statements are not understandable by the compiler.

The following can be the cases for the semantic error:

Use of an un-initialized variable.

int i;

i=i+2;

Errors in expressions

int a, b, c;

a+b = c;

Array index out of bound

int a[10];

a[10] = 34;

**Text Editors**

Text editors are software applications designed for creating, modifying, and managing plain text files. They are fundamental tools for developers and anyone who needs to work with text-based content.

Here's an overview of the editing process, user interface, and editor structure of typical text editors:

**Editing Process**

When you open a text file in a text editor, you can perform various operations on the text, such as writing new content, deleting existing content, copying, cutting, pasting, and searching for specific words or phrases. The editing process is typically straightforward and involves making changes directly to the text within the editor.

**User Interface (GUI/CLI)**

The user interface of a text editor is designed to be simple and minimalistic, as it primarily focuses on displaying the text and providing essential editing tools. The common components of a text editor's user interface are:

**Text Area:** This is the central part of the editor where the content of the text file is displayed and edited. Users can type, modify, and delete text within this area.

**Menu Bar:** The menu bar contains various dropdown menus that offer access to additional functionalities such as opening or saving files, changing preferences, and accessing advanced features.

**Toolbar:** The toolbar usually contains icons or buttons for frequently used actions like cut, copy, paste, undo, redo, and search.

**Status Bar:** The status bar displays information about the current state of the text editor, such as the line and column number, text encoding, and other relevant details.

**Editor Structure**

Text editors can have different structures based on their complexity and features. Here are some common types:

**Basic Text Editor:** This type of editor provides the most essential features for basic text manipulation, like Notepad on Windows or TextEdit on macOS.

**Code Editor:** Code editors are designed specifically for programming and offer additional functionalities like syntax highlighting, code completion, and error checking. Examples include Visual Studio Code, Sublime Text etc.

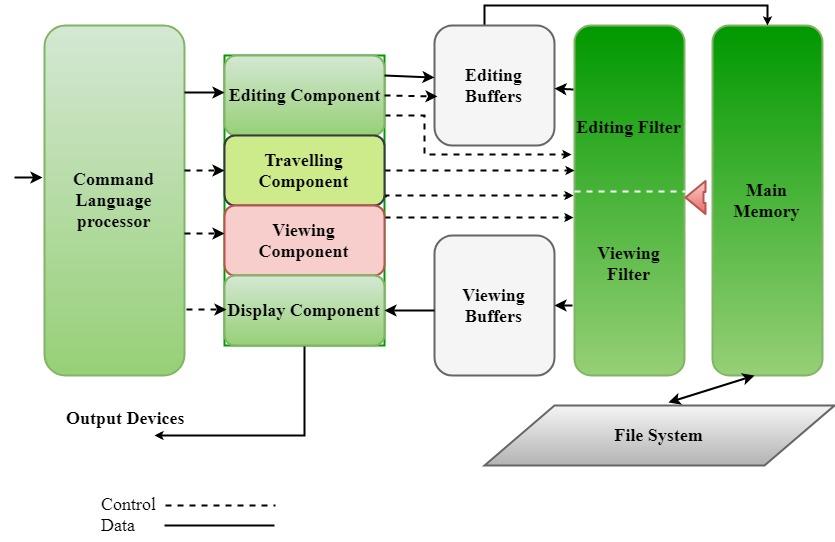
**Integrated Development Environment (IDE):** An IDE is a comprehensive software suite that combines a code editor with various development tools, such as debugging, version control integration, and build management. Examples include Eclipse and Visual Studio.

**Command Line Text Editor:** Some text editors, like Vim and Emacs, operate entirely within the command-line interface and are highly customizable with powerful features.

The editor structure can significantly impact the user's workflow and productivity, as different editors cater to specific needs and programming languages.

In conclusion, text editors play a crucial role in the digital world, allowing users to work with plain text efficiently. Their simple user interfaces and various editor structures make them versatile tools for a wide range of tasks, from writing code to drafting documents and taking notes.

Architecture of Text Editor-



**Programming Languages and Language Processors**

Programming languages are categorized into different levels based on their proximity to human-readable code and machine-executable code. The levels include low-level, mid-level, and high-level programming languages. Here's an explanation of each level:

**1. Low-Level Programming Languages:**

Low-level languages are those that are very close to the machine code and hardware of a computer. They provide a minimal abstraction over the hardware architecture and are often specific to a particular machine or processor. Low-level languages are generally difficult to read and write compared to high-level languages. There are two types of low-level languages:

**Assembly Language:** Assembly language is a symbolic representation of machine code instructions. Instead of writing binary machine code, programmers write human-readable mnemonics that correspond to specific machine instructions. Assembly languages are specific to the architecture of the computer's central processing unit (CPU).

**Machine Code:** Machine code is the lowest level of programming language, consisting of binary instructions that the computer's CPU can directly execute. Each instruction corresponds to a specific operation that the CPU can perform.

Low-level languages are used in tasks where direct hardware control or performance optimization is critical, such as device drivers, embedded systems, and operating system components.

**2. Mid-Level Programming Languages:**

Mid-level languages, also known as system programming languages, strike a balance between low-level and high-level languages. They offer a higher level of abstraction than assembly or machine code while still providing a level of control over hardware. Mid-level languages are designed to make programming more efficient and portable across different hardware platforms. C and C++ are examples of mid-level languages.

Mid-level languages are used for various purposes, including system programming, development of operating systems and utilities, and tasks that require a combination of high-level abstractions and low-level control.

**3. High-Level Programming Languages:**

High-level languages are designed with a focus on human readability and ease of use. They offer a high level of abstraction from the hardware details and provide a wide range of built-in functions and libraries to simplify programming tasks. High-level languages are often platform-independent, meaning code written in one high-level language can be executed on different platforms with the appropriate interpreter or compiler.

Examples of high-level languages include Python, Java, C#, Ruby, and many more. High-level languages are commonly used for application development, web development, data analysis, scripting, and more.

**Language Processors**

Computer programs are usually written in high-level programming languages (like C++, Python, and Java). Further, to make them understandable by the computer, a language processor needs to translate the source code into machine code (also known as object code, which is made up of ones and zeroes).

There are three types of language processors: assembler, compiler, and interpreter.

* Assembler
* Compiler
* Interpreter

**Comparison between Interpreter and Compiler**

A compiler and an interpreter are both tools used in language processing, but they have distinct differences in how they process and execute programming code. Here are the key differences between a compiler and an interpreter:

**Compilation vs. Execution:**

* Compiler: A compiler translates the entire source code of a program into machine code or an intermediate representation in a single pass. The resulting machine code can be executed multiple times without the need for recompilation.
* Interpreter: An interpreter processes the source code line by line or statement by statement and executes it immediately. It translates each line or statement into machine code or performs the required operations as it encounters them.

**Output:**

* Compiler: The output of a compiler is usually a standalone executable file or a binary code that can be executed independently.
* Interpreter: The interpreter doesn't produce a separate executable file. It directly executes the source code and produces output based on the input provided during execution.

**Efficiency:**

* Compiler: Compilers generally produce more efficient code since they perform extensive optimizations during the compilation process. Once compiled, the code can be executed quickly.
* Interpreter: Interpreters are generally slower than compiled code because they need to perform the translation and execution steps simultaneously. However, modern interpreters often include Just-In-Time (JIT) compilation to improve performance by dynamically translating parts of the code into machine code as needed.

**Debugging and Error Handling:**

* Compiler: Debugging compiled code can be more challenging since errors are detected after the entire code is compiled. Debugging typically involves examining machine code or using debugging tools.
* Interpreter: Interpreters provide a more interactive environment for debugging. Errors are often reported in the context of the specific line or statement being executed, which can make it easier to identify and fix issues.

**Portability:**

* Compiler: Compiled code is often specific to the target architecture and requires recompilation if the code needs to be run on a different platform.
* Interpreter: Interpreted code is generally more portable since the interpreter can be run on different platforms, and the source code itself is usually platform-independent.

**Memory Usage:**

* Compiler: Compiled programs tend to have lower memory requirements since they don't need to include the interpreter's runtime overhead.
* Interpreter: Interpreted programs may have higher memory usage due to the presence of the interpreter itself during execution.

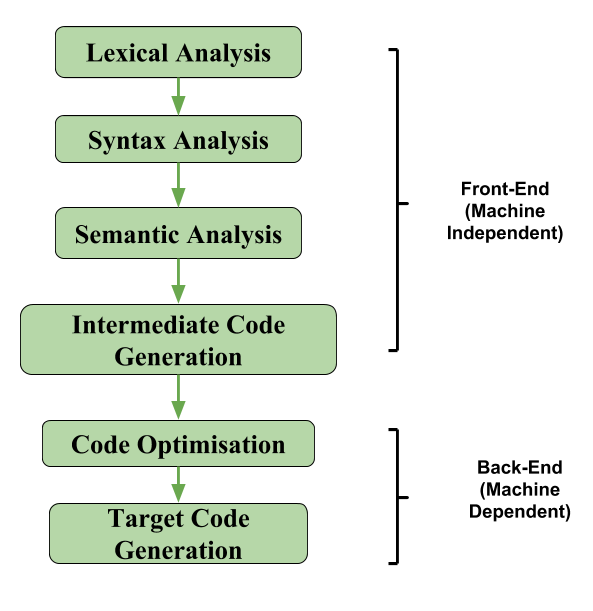
**Development Workflow:**

* Compiler: Compilation can introduce a delay before the program can be executed, but subsequent executions are generally faster since the compiled code is ready for direct execution.
* Interpreter: Interpreters offer a more immediate development cycle since changes to the source code take effect immediately without requiring a separate compilation step.

In summary, compilers translate the entire source code into machine code or an intermediate representation before execution, resulting in potentially more efficient and standalone executable files. Interpreters execute the source code line by line, providing a more interactive debugging environment and greater portability, but typically at the cost of execution speed. The choice between a compiler and an interpreter depends on factors such as performance, debugging needs, portability, and the development workflow.

**Language Processing Activities**

Language processing activities in system software involve the handling and manipulation of human-readable programming languages by a computer system. These activities are crucial for converting the high-level instructions written by programmers into machine-executable code that the computer can understand and execute. The primary components of language processing activities include:



* **Lexical Analysis (Scanning):** it involves three major activities- Tokenization, Gives Error Message (exceeding length, unmatched string/illegal character) and Elimination of comments, white spaces, tab and new line.
* Takes source code as I/P.
* Convert them into tokens.
* This process is known as tokenization which is performed by a lexer/scanner program.

Tokens

* Identifier (name of variables, functions )
* Separators (brackets, semi colon, comma)
* Keywords (int, main, char etc.)
* Operators
* Constants
* Special characters

Q. N. 1- find the total number of tokens in the following programs-

Void main()

{

Int a=5, b=10;

/\* print the greatest value\*/

If(a>b)

return a;

else

return b;

}

Q. N. 2-

Void main()

{

int number;

// print the value of a given number

Printf(“enter the no.”);

Scanf(“”%d”, &number);

printf(“the value of entered no is =%d”,number);

}

Q. N.3- int main ) (

}

Int x, y, z;

X=y+z;

Printf(“%d%d”, x);

{

Q.N.4- main( )

{

X=y +++----+++==;

Printf(“%d%d”, x,y);

}

QN. 5- main( )

{

Int x=5;

Char y=”xyz”;

In t c=20;

Ch ar d=”abc”;

in /\* comments \*/ t m=20;

}

* **Syntax Analysis (Parsing):** The next step is to analyze the structure of the tokens to ensure they conform to the grammar of the programming language. The grammar defines the rules for how different tokens can be combined to form valid statements and expressions. A syntax analyzer, often using a parsing technique like a context-free grammar, checks whether the arrangement of tokens follows the language's syntax rules. If errors are detected, the programmer is usually notified with an error message.
* **Semantic Analysis:** Once the syntax is validated, the system checks for semantic correctness. This involves examining the meaning and context of statements to ensure they make sense according to the rules of the programming language. It includes tasks like type checking, which verifies that data types are used correctly, and scope resolution, which ensures variables and identifiers are used in appropriate scopes.
* **Intermediate Code Generation:** After ensuring the program's syntax and semantics are correct, the system may create an intermediate representation of the code. This intermediate code is an abstract and machine-independent representation that simplifies further processing steps, such as optimization and code generation. Intermediate code is often used to enhance portability and facilitate optimization techniques. The following are commonly used intermediate code representations:
* Syntax Tree
* Postfix Notation
* Three-Address Code

**Advantages of Intermediate Code Generation:**

* Easier to implement: Intermediate code generation can simplify the code generation process by reducing the complexity of the input code, making it easier to implement.
* Facilitates code optimization: Intermediate code generation can enable the use of various code optimization techniques, leading to improved performance and efficiency of the generated code.
* Platform independence: Intermediate code is platform-independent, meaning that it can be translated into machine code or byte code for any platform.
* Code reuse: Intermediate code can be reused in the future to generate code for other platforms or languages.
* Easier debugging: Intermediate code can be easier to debug than machine code or byte code, as it is closer to the original source code.

**Disadvantages of Intermediate Code Generation:**

* Increased compilation time: Intermediate code generation can significantly increase the compilation time, making it less suitable for real-time or time-critical applications.
* Additional memory usage: Intermediate code generation requires additional memory to store the intermediate representation, which can be a concern for memory-limited systems.
* Increased complexity: Intermediate code generation can increase the complexity of the compiler design, making it harder to implement and maintain.
* Reduced performance: The process of generating intermediate code can result in code that executes slower than code generated directly from the source code.
* **Code Optimization:** This phase involves analyzing the intermediate code to improve the efficiency of the resulting machine code. Various techniques, such as **loop unrolling, constant folding, and common sub expression elimination, are applie**d to optimize the code's execution speed and memory usage while maintaining its functionality.

**Loop Unrolling-**

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program’s speed by eliminating loop control instruction and loop test instructions.

Program 1:

// This program does not use loop unrolling.

#include<stdio.h>

int main()

{

for (int i=0; i<5; i++)

printf("Hello\n"); //print hello 5 times

return 0;

}

// This program uses loop unrolling.

#include<stdio.h>

int main()

{

printf("Hello\n");

printf("Hello\n");

printf("Hello\n");

printf("Hello\n");

printf("Hello\n");

return 0;

}

**Constant Folding-**

Consider the following pseudocode :

a = 30

b = 20 - a /2

c = b \* ( 30 / a + 2 ) - a

We can see that in the first expression value of a have assigned a constant value that is 30. Now, when the compiler comes to execute the second expression it encounters a, so it goes up to the first expression to look for the value of a and then assign the value of 30 to a again, and then it executes the second expression. Now it comes to the third expression and encounters b and a again, and then it needs to evaluate the first and second expression again in order to compute the value of c. Thus, a needs to be propagated 3 times This procedure is very time consuming.

We can instead , rewrite the same code as :

a = 30

b = 20 - 30/2

c = b \* ( 30 / 30 + 2) - 30

This updated code is faster as compared to the previous code as the compiler does not need to again and again go back to the previous expressions looking up and copying the value of a variable in order to compute the current expressions. This saves a lot of time and thus, reducing time complexity and perform operations more efficiently.

* **Code Generation:** In this phase, the optimized or intermediate representation is translated into machine-specific instructions. These instructions are expressed in the machine's assembly language or directly in binary code. Code generation involves allocating registers, managing memory, and generating efficient sequences of machine instructions.
* **Symbol Table Management:** Throughout the language processing activities, a symbol table is maintained. This table stores information about variables, constants, functions, and other identifiers used in the program. It helps keep track of data types, scope, memory locations, and other attributes associated with each identifier.
* **Error Handling and Reporting:** Throughout the language processing activities, errors can occur, ranging from syntax mistakes to semantic inconsistencies. The system software is responsible for detecting, reporting, and sometimes recovering from these errors. Meaningful error messages are provided to guide programmers in identifying and rectifying the issues.

In summary, language processing activities in system software ensure that the code written by programmers is transformed into a format that the computer can understand and execute correctly. This involves various stages like lexical analysis, syntax analysis, semantic analysis, intermediate code generation, code optimization, code generation, symbol table management, and error handling. Each of these stages contributes to producing efficient and reliable machine-executable code from high-level programming languages.

**Symbol Table**

The symbol table is defined as the set of Name and Value pairs. Symbol Table is an important data structure created and maintained by the compiler in order to keep track of semantics of variables i.e. it stores information about the scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

It is used by various phases of the compiler as follows:-

**1. Lexical Analysis:** Creates new table entries in the table, for example like 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 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 address information of 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.

**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:

Lightbox

**Operations on Symbol Table :**

**Following operations can be performed on symbol table-**

1. Insertion of an item in the symbol table.

2. Deletion of any item from the symbol table.

3. 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.

* 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 –(Refer to Hash functions Kmod n, K mod 10, Collision Resoultion Techniques i.e. open hashing, closed hashing- linear and Quadratic Probing)**

* 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(log2 n) on average.

**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**

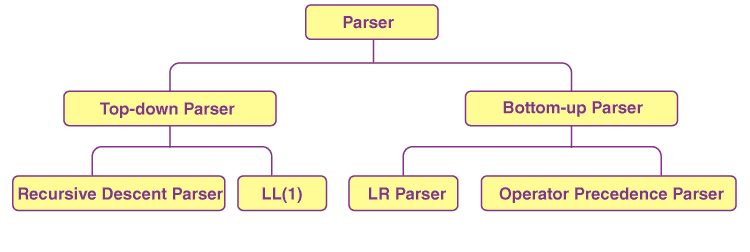
* Resolution of variable and function names: Symbol tables are used to identify the data types and memory locations of variables and functions as well as to resolve their names.
* Resolution of scope issues: To resolve naming conflicts and ascertain the range of variables and functions, symbol tables 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.

What is Parsing?

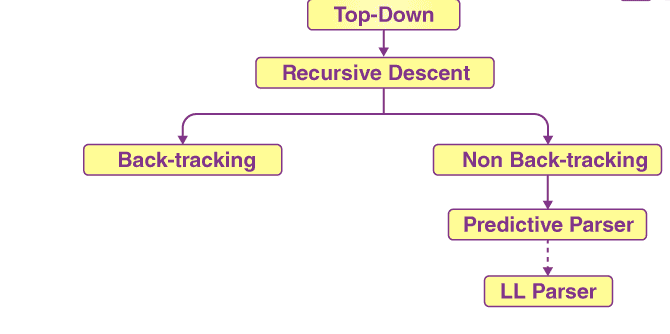
A parser takes input in the form of sequence of tokens (strings) and checks whether rt hey belong to a particular grammar (context free grammar) or not. The output is referred to as parse tree.

Types of Parsing:

* The Top-down Parsing
* The Bottom-up Parsing



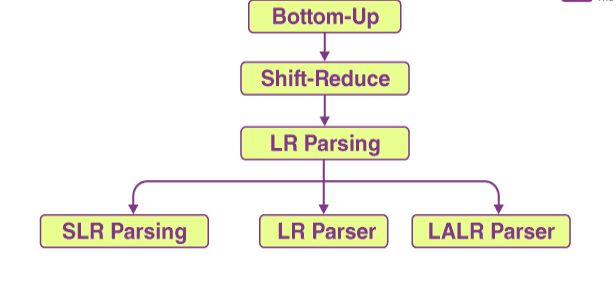
**Top-down Parsing:** When the parser generates a parse with top-down expansion to the first trace, the left-most derivation of input is called top-down parsing. The top-down parsing initiates with the start symbol and ends on the terminals. Such parsing is also known as predictive parsing.



**Recursive Descent Parsing:** Recursive descent parsing is a type of top-down parsing technique. This technique follows the process for every terminal and non-terminal entity. It reads the input from left to right and constructs the parse tree from right to left. As the technique works recursively, it is called recursive descent parsing.

**Back-tracking:** The parsing technique that starts from the initial pointer, the root node. If the derivation fails, then it restarts the process with different rules.

**Bottom-up Parsing:** The bottom-up parsing works just the reverse of the top-down parsing. It first traces the rightmost derivation of the input until it reaches the start symbol.



**Shift-Reduce Parsing:** Shift-reduce parsing works on two steps: Shift step and Reduce step.

**Shift step:** The shift step indicates the increment of the input pointer to the next input symbol that is shifted.

**Reduce Step:** When the parser has a complete grammar rule on the right-hand side and replaces it with RHS.

**LR Parsing:** LR parser is one of the most efficient syntax analysis techniques as it works with context-free grammar. In LR parsing L stands for the left to right tracing, and R stands for the right to left tracing.