1. Subprogram overloading

Subprogram overloading is the ability to define multiple subprograms with the same name but different parameter lists. This allows programmers to create subprograms that perform similar tasks but operate on different data types or with different numbers of parameters.

Example koode function overloading nte

2. **Message passing in concurrency control**

Message passing is a form of interprocess communication (IPC) used in concurrent programming languages to allow processes or threads to exchange information with each other. In message passing, processes communicate by sending and receiving messages, rather than by accessing shared memory or other shared resources.

Overall, message passing is an important topic in concurrent programming languages, and is widely used in both distributed and shared-memory concurrency. Message passing allows processes or threads to communicate with each other in a safe and structured way, without the need for shared memory or other shared resources. However, message passing can suffer from synchronization problems such as deadlocks and race conditions, and requires careful programming to ensure correctness and efficiency.

3.real world applications where prolog is used

1. Expert Systems: Prolog's declarative and logical nature makes it suitable for building expert systems. It has been used in areas such as medical diagnosis, natural language processing, and knowledge-based systems.
2. Natural Language Processing: Prolog's ability to process symbolic information and perform pattern matching has made it useful in natural language processing tasks like parsing, semantic analysis, and text understanding.
3. Rule-based Systems: Prolog's rule-based programming paradigm allows for the development of rule-based systems. These systems can be employed in areas such as business rules engines, decision support systems, and configuration systems.

4.Advantages and disadvantages of dynamic local variables in subprogram

Advantages of dynamic local variables in subprograms:

1. Flexibility:.
2. Memory Efficiency
3. Resource Utilization

Disadvantages

1. Increased Complexity

2. Runtime Overhead:.
3. Potential for Errors:
4. Limited Scope:

Part b

7A

Explain diff parameter passing models in programs

1. Call by value: In this method, the value of the parameter is passed to the subprogram. The subprogram cannot modify the original value of the parameter, but it can use it in its calculations.
2. Call by reference: In this method, the address (or pointer) of the parameter is passed to the subprogram. The subprogram can modify the original value of the parameter, as it has access to the memory location where the parameter is stored.
3. Call by name: In this method, the parameter is not evaluated until it is used in the subprogram. This can be useful when the evaluation of the parameter is expensive or when the parameter is used conditionally.
4. Call by need: In this method, the parameter is not evaluated until it is actually needed by the subprogram. This can be useful when the evaluation of the parameter is expensive or when the parameter is not always needed.

7B DIFFERENTIATE coroutines from conventional programs

Coroutines differ from conventional programs in several ways:

1. Control Flow: In conventional programs, control flow is primarily dictated by the program's structure and the order of execution of statements. Coroutines, on the other hand, have the ability to suspend their execution at specific points and resume from where they left off later. This allows for cooperative multitasking and more flexible control flow patterns.
2. Non-Preemptive Execution: Unlike conventional programs that can be preempted by the operating system or other external factors, coroutines have non-preemptive execution. They voluntarily yield control to other coroutines, allowing for explicit coordination and sharing of resources.
3. Cooperative Multitasking: Coroutines enable cooperative multitasking, where multiple tasks or coroutines work together by explicitly yielding control to each other. This allows for fine-grained

concurrency and avoids the complexities and overhead associated with preemptive multitasking.
4. State Preservation: Coroutines maintain their own execution state, including local variables and program counter, when they suspend execution. This means that when a coroutine is resumed, it continues execution from the point of suspension, preserving its internal state. In conventional programs, the state is typically managed by the program stack and is lost when a function returns.
5. Asynchronous Communication: Coroutines often involve asynchronous communication and synchronization mechanisms. They can communicate and exchange data with other coroutines using channels, message passing, or other synchronization primitives. Conventional programs typically rely on synchronous function calls and shared memory for communication.
6. Resumable Execution: Coroutines have the ability to be suspended and resumed multiple times during their execution, allowing for more flexible and fine-grained control over program flow. This is in contrast to conventional programs that follow a linear execution model without the ability to pause and resume at arbitrary points.

9A Exception handler and how exceptions are handled

Exception handling is a mechanism used in programming languages to handle errors and other exceptional situations that can occur during the execution of a program. Exception handling allows programmers to separate error-handling code from normal program logic, making code more readable and maintainable. Here are some key points related to exception handling:

1. Exception handling involves two parts: Exception handling involves two parts: throwing an exception, and catching the exception. When an exceptional situation occurs, a program can throw an exception, which interrupts normal program flow and transfers control to an exception handler. The exception handler can then catch the exception and take appropriate action, such as logging an error message, retrying an operation, or terminating the program.
2. Exceptions can be predefined or user-defined: Programming languages typically provide a set of predefined exceptions, such as divide-by-zero, file-not-found, and out-of-memory. Programmers can also define their own exceptions, which are often used to represent specific error conditions in a program.
3. Exception handling can be hierarchical: Exceptions can be organized into a hierarchy, with more general exceptions at the top and more specific exceptions at the bottom. This allows exception handlers to

catch exceptions at different levels of specificity, depending on their needs.
4. Exception handling can impact performance: Exception handling can have an impact on program performance, particularly if exceptions are thrown and caught frequently. However, the performance impact of exception handling is typically small compared to other factors, such as I/O or database access.
5. Exception handling can be used for resource management: Exception handling can be used to manage resources such as files, network connections, and memory. For example, a program can use exception handling to ensure that resources are always released, even if an exception occurs during their use.

9B how to overcome design issues in exception handling

To overcome design issues in exception handling, there are several best practices and techniques that can be applied:

1. Clearly Define Exception Handling Policies: Establish clear guidelines and policies for exception handling in your software development process. Define when and how exceptions should be used, and document the expected behavior in exceptional situations.
2. Use Specific and Meaningful Exception Types: Create custom exception types that are specific and meaningful for your application domain. This helps in better categorization and handling of exceptions, allowing for more precise error identification and recovery.
3. Use Exception Hierarchies: Organize exceptions in a hierarchy to provide a structured way of handling different types of exceptions. This allows for more granular handling based on the specific exception type or a broader category of exceptions.
4. Separate Exception Handling from Business Logic: Keep exception handling code separate from the core business logic of your application. This promotes code modularity, readability, and maintainability. It also allows for easier modification and updating of exception handling behavior without impacting the main logic.
5. Use Try-Catch-Finally Blocks Appropriately: Employ try-catch-finally blocks to handle exceptions effectively. Place the appropriate code inside the try block that might throw an exception, and handle specific exceptions in separate catch blocks. Finally blocks can be used to perform necessary cleanup or resource releasing operations.
6. Provide Adequate Error Messages and Logging: Include clear and meaningful error messages in your exception handling code. This helps in identifying the cause of the exception and assists in troubleshooting and debugging. Additionally, logging exceptions and associated information can provide valuable insights for diagnosing issues in production environments.
7. Use Exception Propagation Judiciously: Propagate exceptions up the call stack only when it makes sense and is appropriate for the

situation. Avoid excessive nesting of try-catch blocks and unnecessary rethrowing of exceptions.
8. Test Exception Handling Scenarios: Create comprehensive unit tests and integration tests that cover various exception scenarios. This helps ensure that exception handling code behaves as expected and handles exceptions appropriately.

By following these best practices, you can improve the design and implementation of exception handling in your software, making it more robust, maintainable, and resilient to errors.

11A Differentiate functional and imperative programming

FUNCTIONAL	IMPERATIVE
Simpler syntax	Complex syntax
Easier	Difficult to learn and use
Simpler semantics	Complex semantics
High productivity	Less productivity
Smaller programs	Large programs
Slower Uses specific functions like recursive Eg: Lisp	Faster Uses statements, loops Eg : java,c

11B logical programming:

Logical programming is a programming paradigm that is based on the principles of mathematical logic. In logical programming, programs are expressed in terms of logical statements and rules that describe relationships between objects in a domain of discourse. One of the most popular logical programming languages is Prolog, which stands for "Programming in Logic".

In logical programming, the program consists of a set of logical statements and rules that describe the relationships between objects in the domain of discourse. The program can then be queried to find objects that satisfy certain logical conditions.

- Logical programming is a declarative programming paradigm in which programs are expressed in terms of logical statements and rules.
- The most popular logical programming language is Prolog, which is based on first-order predicate logic.

- In Prolog, programs are defined as a set of facts and rules, which are used to represent relationships between objects in a domain of discourse.

SEE QN PAPER

Last q nans: √((6*6)+(8*8)) = √(36+64) = √100 = 10

Miss important enn paranje baki topics

- COROUTINES

- A coroutine is a generalization of a subprogram that allows for multiple entry and exit points. Coroutines are often used for cooperative multitasking, where multiple tasks can run concurrently within a single process.
- Coroutines have two operations: suspend and resume. When a coroutine suspends, it saves its state (including its program counter and local variables) and yields control back to the calling coroutine. When a coroutine resumes, it restores its state and continues executing from where it left off.
- Coroutines can be implemented using either stackful or stackless models. In stackful coroutines, each coroutine has its own call stack, while in stackless coroutines, a single call stack is shared among all coroutines. Stackful coroutines are more efficient but can be more difficult to implement, while stackless coroutines are simpler but may have more overhead.
- In addition to their use in multitasking, coroutines can be used for a variety of other purposes, such as implementing generators (which produce a sequence of values) and iterators (which traverse a data structure).
- Languages that support coroutines include Lua, Python, Ruby, and C#. Some languages, such as C++, have coroutines as an experimental or upcoming feature.
- When designing programs that use coroutines, it is important to carefully manage state and control flow in order to avoid issues such as race conditions and deadlocks.

-CLOSURE

1. A closure is a function object that has access to variables in its lexical scope, even after the scope has exited. This allows the function to "remember" values from previous calls and maintain state across multiple invocations.

2. Closures are often used to implement function factories (which create new functions with specific behavior) and callbacks (which allow functions to be passed as arguments to other functions).
3. Closures are created when a function is defined within another function and the inner function references variables from the outer function. The inner function retains a reference to the environment in which it was created, including the values of any variables that were in scope at that time.
4. When the inner function is called, it has access to the variables in its closure. This allows the function to use and modify the values of those variables, even if they are no longer in scope.
5. Closures are supported in many programming languages, including JavaScript, Python, Ruby, and Scala. Some languages, such as Java, support closures through the use of anonymous inner classes.
6. When designing programs that use closures, it is important to be aware of potential issues such as memory leaks (when closures hold references to objects that are no longer needed) and unexpected behavior due to shared state.

-OOPS CONCEPT DESIGN ISSUES

1. Class design: A key design issue in OOP is the design of classes, which are the blueprints for creating objects. Good class design includes factors such as proper encapsulation, choosing appropriate access modifiers, and ensuring that each class has a single responsibility.
2. Inheritance: Inheritance is a mechanism for creating new classes based on existing ones, and is a key feature of OOP. Good design practices for inheritance include avoiding deep inheritance hierarchies and favoring composition over inheritance where possible.
3. Polymorphism: Polymorphism is the ability of objects to take on multiple forms, and is another key feature of OOP. Good design practices for polymorphism include ensuring that all objects that share a common interface can be used interchangeably, and avoiding overloading methods in ways that can lead to confusion or errors.
4. Interface design: Interfaces are contracts that define the behavior of objects, and are an important design issue in OOP. Good interface design includes factors such as defining clear and concise methods, providing appropriate documentation, and avoiding exposing implementation details to users.
5. Coupling and cohesion: Coupling refers to the degree to which different parts of a program depend on each other, while cohesion refers to the degree to which a single module or class has a single, well-defined responsibility. Good design practices for OOP include minimizing coupling between classes and ensuring that each class has a high degree of cohesion.

6. Design patterns: Design patterns are reusable solutions to common programming problems, and are an important design issue in OOP. Good design practices include using well-established design patterns when appropriate, and avoiding overuse of patterns that can lead to code complexity and maintenance issues.

-SEMAPHORES AND MONITORS

Concurrency control is the management of concurrent access to shared resources in a program. In multi-threaded programs, it is essential to coordinate the access to shared resources in order to avoid race conditions and other synchronization problems. Semaphores and monitors are two common mechanisms used to implement concurrency control.

1. Semaphores: A semaphore is a synchronization object used to control access to shared resources in a program. Semaphores can be used to prevent race conditions by ensuring that only one thread at a time can access a shared resource. Semaphores come in two types: binary and counting. A binary semaphore is a simple on/off switch, while a counting semaphore allows multiple threads to access a shared resource at the same time, up to a certain limit.
2. Monitors: A monitor is a higher-level synchronization mechanism that combines a mutex (a mutual exclusion lock) with a condition variable. A monitor allows multiple threads to access a shared resource in a structured way, by providing methods for entering and exiting the monitor, as well as for waiting and signaling on condition variables. Monitors are typically used to manage access to complex shared data structures, such as databases or communication channels.
3. Deadlocks: Deadlocks can occur in concurrent programs when two or more threads are blocked waiting for each other to release a resource. Deadlocks can be difficult to detect and resolve, and can lead to program crashes or hangs. To avoid deadlocks, it is important to design programs that are deadlock-free, or to use mechanisms such as timeouts or priority inversion avoidance.
4. Race conditions: Race conditions can occur when two or more threads access a shared resource at the same time, and the order of the accesses affects the program's behavior. Race conditions can be difficult to detect and reproduce, and can lead to subtle bugs or security vulnerabilities. To avoid race conditions, it is important to use appropriate synchronization mechanisms, such as semaphores or monitors, to coordinate access to shared resources.

Overall, concurrency control is an important topic in programming languages, particularly in multi-threaded programs. Semaphores and monitors are two common mechanisms used to implement concurrency control, and are essential for managing access to shared resources in a structured and safe way.

-SEARCHING STRATEGIES IN PROLOG

In Prolog, there are two main searching strategies used to solve problems: depth-first search (DFS) and breadth-first search (BFS).

Depth-first search explores a graph by following each path to its deepest point before backtracking. This means that DFS follows a path until it reaches a dead end or a goal state, and then backtracks to the previous node to explore other paths. DFS can be implemented using Prolog's backtracking mechanism and recursion.

Breadth-first search, on the other hand, explores a graph by systematically searching all the nodes at a given depth before moving on to nodes at the next depth. BFS uses a queue data structure to store nodes that have been visited but not explored yet.

-MERITS AND DEMERITS OF INHERITANCE

Merits:

- Reusability: Inheritance allows developers to reuse existing code by creating new classes based on existing ones. This can save time and effort in the development process.
- Polymorphism: Inheritance allows for polymorphism, which means that objects of different classes can be treated as if they were of the same class. This can lead to more flexible and modular code.
- Abstraction: Inheritance allows developers to create abstract classes that define a common interface for a group of related classes. This can make the code more understandable and easier to maintain.
- Extensibility: Inheritance allows developers to extend the functionality of existing classes by adding new methods or attributes to them.

Demerits:

- Tight coupling: Inheritance can lead to tight coupling between classes, which means that changes to one class can have unintended effects on other classes. This can make the code harder to maintain and modify.
- Inheritance hierarchy: Inheritance can create complex hierarchies of classes, which can be difficult to understand and navigate.
- Fragile base class problem: Changes to a base class can have unintended effects on derived classes, leading to bugs and errors in the code.

- Code duplication: Inheritance can lead to code duplication if multiple classes inherit from the same base class but need to make different changes to it. This can make the code harder to maintain and modify.