**Question Bank -UNIT -1**

**Subject: Principle of Programming Languages Subject Code CIE- 320**

**Class: B. Tech CSE 6th Faculty Name: Dr. Archana Sharma/Ms. Shweta Rai**

|  |  |  |
| --- | --- | --- |
| **Q1** | **Describe is the syntax for a for loop in C++?**  In C++, a for loop can be written as follows:  for (initialization; condition; increment/decrement) {  // code to be executed  } | **L2** |
| Q2 | **Discuss that how do you define a class in Java?**  A: In Java, a class can be defined using the "class" keyword followed by the class name and curly braces for the class body.  For example:  public class MyClass {  // class members and methods  } | L2 |
| Q3 | **Examine the necessary attributes to compute the area of a triangle given its base and height.**  The necessary attributes to compute the area of a triangle given its base and height are:  1. Base: The length of the base of the triangle, which is a straight line segment that forms the bottom of the triangle.  2. Height: The perpendicular distance from the base to the opposite vertex (or apex) of the triangle. The height is always measured at a right angle to the base.  These two attributes, the base and the height, are sufficient to calculate the area of a triangle using the formula:  Area = 0.5 \* base \* height | L3 |
| Q4. | **Construct an attribute grammar to compute the factorial of a given number**:  Factorial ::= Expr(fac)  Expr(e) ::= Expr1(e1) \* Expr2(e2) {e = e1 \* e2}  Expr1(e) ::= Num(n) {e = n} | '(' Expr(e) ')' { }  Expr2(e) ::= Expr1(e1) {e = e1}  Num(n) ::= [0-9]+ {n = stoi($)} | L3 |
| Q5 | **Differentiate between static and dynamic scoping work.**  Static scoping and dynamic scoping are two different approaches that programming languages use to determine the visibility and lifetime of variables.  **Static Scoping:**  Static scoping, also known as lexical scoping, is a compile-time concept.  In static scoping, the scope of a variable is determined by its position in the source code.  The visibility of a variable is determined based on its location in the program's source code, typically defined by blocks (e.g., functions or procedures).  Variables are bound to their respective scopes during the compile-time and remain fixed throughout the execution of the program.  When a variable is referenced, the compiler determines its scope by searching for the nearest enclosing scope in which the variable is declared, and uses the value of that variable.  Static scoping allows for better control over variable visibility and can often help in detecting potential errors at compile-time.  **Dynamic Scoping:**  Dynamic scoping is a runtime concept.  In dynamic scoping, the scope of a variable is determined by the current state of the program's execution stack.  The visibility of a variable is determined by the current execution context or the call stack during runtime.  When a variable is referenced, the interpreter or runtime system looks for the most recent declaration of that variable in the program's execution stack, starting from the current context and moving upwards through the call stack until it finds a matching declaration, and uses the value of that variable.  Dynamic scoping can lead to unexpected behavior as the value of a variable can change depending on the sequence of function calls.  It is less common than static scoping and is generally considered less predictable and less safe.  In summary, static scoping is determined at compile-time and uses the structure of the program to determine variable visibility, while dynamic scoping is determined at runtime and uses the call stack to determine variable visibility. | L2 |
| Q6. | **Compare the L value and R-Value > Justify How is the concept of L value and R value relevant to different programming languages?**  The concepts of L value and R value are relevant in programming languages as they define the rules for assigning values to variables and expressions.  L value (left value): It refers to an expression or a variable that can be assigned a value. L values are typically locations in memory where a value can be stored. They represent the object or memory location to which an assignment can be made. In simpler terms, an L value represents something that can stand on the left side of an assignment statement.  R value (right value): It refers to an expression or a value that can be used for assigning to an L value. R values are typically constants, literals, or the result of an expression. They represent the values that can be assigned to an L value. R values can stand on the right side of an assignment statement.  Relevance to programming languages:  C and C++: In C and C++, L values and R values have distinct significance. For example, an L value can be on the left side of an assignment statement, whereas an R value can be on the right side of an assignment statement. L values are required for modification, while R values represent values that can be assigned. Understanding L values and R values is crucial for understanding pointer operations, references, and memory management in these languages.  Java: The concept of L value and R value is less explicitly defined in Java. Java is a strictly pass-by-value language, where variables hold values, not references to locations in memory. In Java, the distinction between L values and R values is less relevant, as all variables are treated as values.  Python: Python does not have explicit notions of L values and R values. In Python, the distinction between L values and R values is blurred, as variables hold references to objects. Assigning a value to a variable is similar to creating a name or reference to an object. Python variables can point to any valid object, making the concept of L values and R values less crucial.  In summary, while the concept of L value and R value is explicitly relevant in languages like C and C++, it may be less significant in languages like Java and Python, where variables and assignments are handled differently. | L4 |
| Q7. | **Describe the role of an L value in assignment statements?**  The role of an L value in assignment statements is to specify the location in memory where a value should be stored. In other words, it represents the variable or memory location that will receive the assigned value. An L value refers to an identifier (variable name) that can be assigned a value. | L2 |
| Q8. | **Discuss the restrictions on the usage of L values and R values in programming?**  There are certain restrictions on the usage of l-values and r-values in programming.  1. L-values can appear on both sides of an assignment operator (=), as they represent objects that can be assigned new values. However, r-values can only appear on the right-hand side of an assignment operator.  Example:  int x = 5; // x is an l-value and can be assigned a new value  int y = x; // x is an r-value in this case as it is used to initialize y  2. L-values can be taken the address of, whereas r-values cannot. Taking the address of an object allows you to access and modify its value indirectly.  Example:  int x = 5;  int\* ptr = &x; // valid as &x takes the address of x (l-value)  3. L-values can be used as arguments in functions that expect references, while r-values cannot be bound to non-const references. This is because r-values are temporary values whereas l-values persist.  Example:  void foo(int& x) {  // do something with x  }  int a = 5;  foo(a); // valid as a is an l-value  foo(5); // invalid as 5 is an r-value  These are some of the restrictions, but the specific rules may vary depending on the programming language you are using. | L2 |
| Q9. | **What are the different storage allocation strategies used in programming?**  There are several storage allocation strategies used in programming, which determine how and when memory is allocated and deallocated for different variables and data structures. Some of the common allocation strategies are:  1. Static Allocation: In this strategy, memory is allocated during compile-time and remains fixed throughout the program execution. Variables declared with a static allocation are typically stored in the data section of the program's memory and have a fixed size.  2. Stack Allocation: The stack allocation strategy is commonly used for local variables and function calls. Memory is allocated from the stack during runtime when a function is called, and deallocated when the function returns. It follows the Last-In-First-Out (LIFO) principle, meaning the most recently allocated memory is the first to be deallocated.  3. Heap Allocation: The heap allocation strategy is used for dynamically allocating memory during runtime. It allows flexible and dynamic memory allocation and deallocation. Memory allocated on the heap must be manually managed by the programmer, requiring explicit allocation and deallocation calls (e.g., malloc, free in C) to avoid memory leaks or invalid accesses.  4. Dynamic Array Allocation: In this strategy, memory for arrays is allocated on the heap or stack depending on the array's scope and requirements. Dynamic arrays allow resizing during runtime, whereas static arrays have a fixed size determined at compile-time.  5. Garbage Collection: Garbage collection is a memory management technique used in higher-level programming languages like Java and C#. It automatically deallocates memory for objects that are no longer in use, freeing the programmer from managing memory manually. Garbage collection typically utilizes algorithms to identify and reclaim unused memory periodically.  6. Object Pools: Object pooling is a technique where a fixed number of pre-initialized objects are created and stored in a pool. When needed, objects are taken from the pool and returned after use. Object pools reduce memory allocation overhead and can optimize resource usage in situations where creating and destroying objects frequently is cost-intensive.  These are some commonly used storage allocation strategies, and the choice of strategy depends on the programming language, application requirements, and performance considerations. |  |
| Q10. | **Compare the different translation models in programming?**  There are various translation models used in programming to convert code from one language to another. These models differ based on their approach, complexity, and functionality. Some of the common translation models are:  1. Literal Translation: This model directly translates each line of code from the source language to the target language. It maps equivalent statements and syntax without any additional optimizations or transformations. It is a straightforward approach, but it may not produce the most efficient or idiomatic code in the target language.  2. Source-to-Source Translation: Also known as transpilation or transcompilation, this model translates the source code to a semantically equivalent representation in another language. It involves parsing the source code, building an abstract syntax tree (AST), and then generating target code based on this AST. Source-to-source translation allows for language-specific optimizations and transformations, producing more efficient code in the target language.  3. Intermediate Representation (IR) Transformation: In this model, the source code is translated into an intermediate representation (IR) that is language-agnostic. The IR captures the essence and behavior of the original code independent of the source and target languages. Once the IR is generated and validated, it can be transformed or optimized before generating the translated target code. IR transformations enable more advanced optimizations and analysis techniques.  4. Virtual Machine Translation: This model involves translation to a virtual machine (VM) bytecode or intermediate language, which can then be executed by a runtime environment specific to the target language. The source code is translated into the VM language, and the runtime environment handles the execution. This approach allows for portability across different platforms and languages that share the same VM.  5. Just-In-Time (JIT) Compilation: JIT compilation combines interpretation and translation. The source code is initially interpreted while profiling runtime behavior. As the code is executed, hot and frequently executed portions are identified, and those segments are dynamically translated into native machine code for faster execution. JIT compilation can provide a balance between performance and flexibility, as it optimizes code during runtime.  These different translation models offer various trade-offs in terms of speed, efficiency, portability, and flexibility. The choice of model depends on the specific requirements, constraints, and goals of the translation process. | L2 |
| Q11. | **Compare and contrast between the special purpose and general-purpose programming languages.**  Special Purpose vs General Purpose Programming Languages:  Special Purpose Languages: These are designed for specific tasks or domains. They offer specialized features tailored for particular applications. Examples include SQL for database querying, MATLAB for numerical computing, and VHDL for hardware description.  General Purpose Languages: These are versatile languages designed to be used for a wide range of applications. They provide broad features and capabilities. Examples include Python, Java, C++, and JavaScript. | L2 |
| Q12. | **Describe the concept of attribute grammar and provide a syntax-directed definition for a desktop calculator.**  Attribute Grammar and Syntax-Directed Definition for a Desktop Calculator:  Attribute Grammar: An attribute grammar is a formalism for describing the behavior of context-free grammars, typically used in compilers. It associates attributes with the grammar's production rules.  Syntax-Directed Definition for a Desktop Calculator:  Consider a syntax-directed definition for a basic arithmetic calculator:  Define attributes:  value: Holds the value of the expression.  Production rules:  <expr> ::= <expr> + <term>:  value(expr) = value(expr) + value(term)  <expr> ::= <term>:  value(expr) = value(term)  <term> ::= <term> \* <factor>:  value(term) = value(term) \* value(factor)  <term> ::= <factor>:  value(term) = value(factor)  <factor> ::= <number>:  value(factor) = value(number)  <number> ::= <digit>:  value(number) = value(digit)  <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9:  value(digit) = corresponding\_integer\_value  Use these rules to compute the value of the expression 5 \* (3 + 7) - 2. | L2 |
| Q13. | **Discuss the two way selection (if, if-else, nested if-else, cascaded if else) in C language with syntax.**  Two-way Selection in C:  if (condition) {  // code to execute if condition is true  } else {  // code to execute if condition is false  } | L2 |
| Q14. | **Discuss any potential implications or restrictions on the usage of L-values and R-values in programming languages. Identify the L-value(s) and R-value(s) in the assignment statement y = x;**  L-values and R-values:  L-value: Represents an object that occupies some identifiable location in memory (e.g., a variable). It can appear on the left-hand side of an assignment.  R-value: Represents a value rather than an object stored in memory. It can appear on the right-hand side of an assignment.  In the statement y = x; x is the R-value, and y is the L-value. | L2 |
| Q15. | **Compare mixed mode and assignment statement? How it can be written in Ada and Java?**  Mixed Mode and Assignment Statement:  Mixed Mode: Refers to performing operations involving operands of different types.  Assignment Statement: Assigns a value to a variable.  In Ada: x := 5; (Ada doesn't support mixed-mode arithmetic directly)  In Java: int x = 5; (Java also doesn't support mixed-mode arithmetic directly) | L4 |
| Q16. | **Discuss type checking? Differentiate between static and dynamic type checking and give their relative advantages.**  Static Type Checking:  Definition: Static type checking is performed at compile time, where the types of variables, expressions, and operations are checked before the program is executed.  Advantages:  Early Error Detection: Static type checking catches type-related errors before the program is run, reducing the likelihood of runtime errors.  Improved Code Quality: It enforces stricter typing rules, leading to more robust and maintainable code.  Performance Optimization: Compiler optimizations based on type information can improve the performance of the compiled code.  Example Languages: Java, C, C++, Swift.  Dynamic Type Checking:  Definition: Dynamic type checking is performed at runtime, where the types of values are checked during program execution as operations are performed.  Advantages:  Flexibility: Dynamic typing allows for more flexibility in programming, as variables can hold values of any type without explicit type declarations.  Rapid Development: It can lead to faster development cycles since developers don't need to spend time specifying types explicitly.  Ease of Prototyping: Dynamic typing is well-suited for prototyping and scripting tasks where rapid development is prioritized over type safety.  Example Languages: Python, JavaScript, Ruby.  Relative Advantages:  Static Type Checking:  Provides early detection of type-related errors, leading to more reliable and maintainable code.  Helps in understanding the program's structure and behavior at compile time.  Can enable better compiler optimizations.  Dynamic Type Checking:  Offers flexibility and rapid development, especially in dynamically changing environments.  Simplifies code writing by allowing variables to change types dynamically.  Suitable for scripting and prototyping tasks where quick development is essential. | L2 |
| Q17. | **Given the following code snippet:**  **x = 5**  **y = 10**  **z = 0**  **if x \* y > 30 || z == 0:**  **z = x + y**  **else:**  **z = x – y**  **Examine the final value of z after executing this code snippet.**  **Final Value of z:**  x = 5  y = 10  z = 0  if x \* y > 30 || z == 0:  z = x + y, since x \* y > 30 is true  Final value of z is 5 + 10 = 15. | L3 |
| Q18. | **Discuss the concept of statement-level control structures in programming languages. Provide examples of different types of statement-level control structures and discuss their significance in program flow control.**  Statement-level control structures in programming languages determine the flow of execution within a program by allowing for conditional execution and iteration. They provide mechanisms for making decisions based on certain conditions and for repeating a set of statements multiple times. Here are some common types of statement-level control structures along with their significance in program flow control:  Conditional Statements (if, if-else, nested if-else):  if statement: Executes a block of code only if a specified condition is true.  if condition:  # code to execute if condition is true  **if-else statement:** Executes one block of code if a condition is true and another block if the condition is false.  if condition:  # code to execute if condition is true  else:  # code to execute if condition is false  **nested if-else statement:** Allows for multiple levels of conditionals.  if condition1:  if condition2:  # code to execute if both condition1 and condition2 are true  else:  # code to execute if condition1 is true and condition2 is false  else:  # code to execute if condition1 is false  Statement-level control structures in programming languages determine the flow of execution within a program by allowing for conditional execution and iteration. They provide mechanisms for making decisions based on certain conditions and for repeating a set of statements multiple times. Here are some common types of statement-level control structures along with their significance in program flow control:  Conditional Statements (if, if-else, nested if-else):  if statement: Executes a block of code only if a specified condition is true.  python  Copy code  if condition:  # code to execute if condition is true  if-else statement: Executes one block of code if a condition is true and another block if the condition is false.  python  Copy code  if condition:  # code to execute if condition is true  else:  # code to execute if condition is false  nested if-else statement: Allows for multiple levels of conditionals.  python  Copy code  if condition1:  if condition2:  # code to execute if both condition1 and condition2 are true  else:  # code to execute if condition1 is true and condition2 is false  else:  # code to execute if condition1 is false  Significance: Conditional statements allow programs to make decisions based on different conditions, enabling different code paths to be executed depending on the state of the program or its inputs.  Switch Statement (or Case Statement):  Executes one of many possible blocks of code, depending on the value of an expression.  switch (expression) {  case value1:  // code to execute if expression equals value1  break;  case value2:  // code to execute if expression equals value2  break;  // additional cases...  default:  // code to execute if expression doesn't match any case  }  Significance: Switch statements provide a concise way to handle multiple possible outcomes based on the value of a single expression, improving code readability and maintainability.  Looping Statements (for, while, do-while):  for loop: Executes a block of code repeatedly until a specified condition is false.  for (initialization; condition; increment/decrement) {  // code to execute in each iteration  }  **while loop:** Executes a block of code as long as a specified condition is true.  while (condition) {  // code to execute while condition is true  }  do-while loop: Similar to a while loop but guarantees that the block of code is executed at least once before the condition is checked.  while True:  # code to execute at least once  if condition:  break  Looping statements enable repetitive execution of code, allowing tasks to be performed iteratively, such as processing elements in a list, iterating through a range of values, or waiting for certain conditions to be met.  These statement-level control structures are fundamental building blocks in programming, providing the necessary flexibility to create complex algorithms and control the flow of execution within programs. They allow developers to write more expressive and efficient code by enabling conditional branching and iterative processing. | **L2** |
| Q19. | Consider the following BNF grammar for a simple arithmetic expression language:  <expr> ::= <term> | <expr> + <term> | <expr> - <term>  <term> ::= <factor> | <term> \* <factor> | <term> / <factor>  <factor> ::= <number> | ( <expr> )  <number> ::= <digit> | <digit> <number>  <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  Using the given grammar, Construct the parse the following arithmetic expression: 5 \* (3 + 7) – 2  To parse the given arithmetic expression using the provided BNF grammar, we need to break down the expression according to the grammar rules and apply them recursively. Let's go through the steps:  Given expression: 5 \* (3 + 7) - 2  Start with the highest-level rule <expr>:  <expr> can be either <term> or <expr> + <term> or <expr> - <term>.  Break down the expression:  Since the expression starts with a number (5), it matches the rule <term>.  <term> can be further broken down into <factor>.  <factor> can be a <number>, and since 5 is a number, it matches <number>.  Continue parsing:  The next character after 5 is \*, indicating multiplication.  So, we continue with the <term> rule.  <term> can be <term> \* <factor>.  We already have <term> as 5, and <factor> is (3 + 7).  Parse the expression inside the parentheses:  (3 + 7) matches the <factor> rule, which is <expr> inside parentheses.  So, we start parsing <expr>: 3 + 7.  Parse the expression inside the parentheses:  <expr> can be <term>, and since 3 is a number, it matches <term>.  <term> is <factor>, and since 3 is a number, it matches <number>.  Continue parsing:  The next character after 3 is +, indicating addition.  So, we continue with the <term> rule.  <term> can be <term> \* <factor>.  We already have <term> as 3, and <factor> is 7.  Parse the expression inside the parentheses:  7 matches the <factor> rule, which is <number>.  So, 7 is a number.  Finish parsing the expression inside the parentheses:  We have completed parsing the expression inside the parentheses, which evaluates to 10.  Continue parsing the outer expression:  We now have 5 \* 10.  Perform the multiplication: 5 \* 10 = 50.  Finish parsing the expression:  After multiplication, we have 50 - 2.  Perform the subtraction: 50 - 2 = 48.  So, the given arithmetic expression 5 \* (3 + 7) - 2 evaluates to 48. | L3 |
| Q20. | **Discuss the scope and lifetime of variables. Illustrate when they would coincide and when they don’t.**  The scope of a variable refers to the portion of the program in which the variable can be accessed or used. The lifetime of a variable refers to the duration during which the variable exists in the memory.  In many programming languages, such as C++ and Java, the scope of a variable is typically defined by the block of code in which it is declared. For example, a variable declared inside a function is only accessible within that function, making its scope limited to that function. Variables declared outside of any function, known as global variables, have a larger scope and can be accessed from anywhere in the program.  The lifetime of a variable is determined by where and how it is declared. Local variables have a lifetime limited to the block of code in which they are declared. Once the block of code is executed and terminated, the local variable is destroyed and its memory is released. Global variables, on the other hand, have a lifetime that extends for the entire duration of the program.  Variables can coincide in scope and lifetime when they are declared in the same block of code, such as within a function. In this case, the variables have the same scope (the function) and lifetime (the duration of the function execution).  Variables may not coincide in scope and lifetime when they are declared in different blocks of code such as one being a global variable and the other being a local variable in a function. In this case, the global variable has a larger scope and longer lifetime compared to the local variable.  Understanding the scope and lifetime of variables is important for writing clean and efficient code, as it helps prevent naming conflicts and memory leaks. | L2 |