MODEL QUESTION PAPER ANSWER KEY

1. Differentiate between readability and writability.

The language evaluation criteria are:

Readability: the ease with which programs can be read and understood.

Writability: the ease with which a language can be used to create programs.

Reliability: conformance to specifications (i.e., performs to its specifications).

Cost: the ultimate total cost.

We will look at them in details:

a) Readability - It is the ability to understand the program, and then modify it.

Readability has strong relation with the language of choice and the application domains. Readability involves:

i. Simplicity

• If it is simple, it is easy to learn.

• Programmers usually use the features they are familiar with, which are not necessarily the ones that the readers of the program are familiar with.

• Most often a large number of features will do the same thing, causing unnecessary confusion. That is, multiplicity of features does not help.

Example: i = i + 1; i+ = 1; i + +; + + i;

• Operator overloading gives an operator multiple meanings, which may be confusing when done incorrectly. For example, in some languages, + is used for integer addition, floating point addition, decimal addition and string concatenation

• Assembly languages are simple, but are too low-level (or not “sophisticated” enough) to express our ideas.

ii. Orthogonality

• Features can be combined in a systematic way to form new features.

• Orthogonality means independent, i.e., if you can do one thing with a feature, there is no reason that you cannot do the same to the other. In a sense the features are symmetric.

• If a language is more orthogonal, it has fewer exceptions, and may become simpler, but may have unnecessary complexity since the number of legal constructs would be very large.

• VAX and IBM addition example.

- Orthogonality requires every possible combination of primitives to be legal. Example: 32bit addition from IBM/VAX assembly languages: IBM

A reg, mem; adds mem into reg

AR reg1, reg2; adds reg2 into reg1

VAX

addl op1, op2 ; op1 and op2 are any reg/mem

The VAX instruction set exhibits orthogonality

• Simplicity is the result of a combination of a small number of features

through Orthogonality.

iii. Control Statements

• Structured programming proclaims blocked control structures with single entry and exit, and discourages the use of goto statements.

• Goto statements should be used with caution. They should precede their destination, and never jump to distant labels.

• Limited use of goto statements is still unavoidable in certain languages, like FORTRAN or BASIC.

iv. Data Types and Structures

• Sufficient data type and data structure support not only improve readability, but also expressibility.

• Adequate facilities for defining data types and structures aids readability.

• Primitive/intrinsic data types should be adequate, too.

v. Syntax Considerations

• Identifier form - should not be too restrictive on length. The length and valid combination of characters.

• Special words - Words such as while, if, end, class, etc., have special meaning within a program. How to express compound statement? Can special words be used as identifiers?

• Form and meaning - How a statement appears should clear indicate whatit actually means.

b) Writability - Does the language make it easy to write what you have in mind?

Readability first, then writability. How easily can the language be used to create programs for a particular domain? Consider Visual Basic (VB) and C for – A graphical game program, – An embedded controller for automotive brakes.

i. Simplicity and Orthogonality

• The combination of a small number of features (simplicity) and the consistent way to combine them to form new features (orthogonality) is crucial.

ii. Support for Abstraction

• Abstraction hides the details (in implementation) of a construct and only provides a clear interface of how this construct should be used. It specifies what a construct works, not how it works.

• Procedure or Process abstraction isolate what a process does from how the process does it. Procedural or process abstraction is a specification that:

- Describes effects on outputs for given inputs

- what it does, not how it does it

- ignores implementation

- Treats procedure or function as a “black box”

- Can be applied in any language. Classic example: sorting, that is: We just want to call a sort routine when we need to sort something. We don’t want to clutter up code by implementing a sort algorithm every time.

• Data abstraction separate what an object should behave given certain events from how it implements the actions.

iii. Expressivity

• The ability to handle data in meaningful, natural ways.

• The language should have convenient way to express computation,

e.g., goto may be sufficient, but while loop is better.

• Abstract Data Types:

 Extend procedural abstraction to data. Example: type float.

 Extends imperative notion of type by: Providing encapsulation of data/functions. Separation of interface from implementation.

2. Define binding and binding time.

Ans set (a) Binding and Binding time.

➢ A binding is an association between two things, such as a name and the thing it names.

➢ Binding time is the time at which a binding is created or, more generally, the time at which any implementation decision is made.

➢ There are many different times at which decisions may be bound:

1. **Language design time**: In most languages, the control flow constructs, the set of fundamental (primitive) types, the available constructors for creating

complex types, and many other aspects of language semantics are chosen when the language is designed.

2. **Language implementation time:** Most language manuals leave a variety of issues to the discretion of the language implementor. Typical examples include the precision (number of bits) of the fundamental types, the coupling of I/O to the operating system's notion of files, the organization and maximum sizes of stack and heap, and the handling of run-time exceptions such as arithmetic overflow.

3. **Program writing time:** Programmers choose algorithms, data structures, and names.

4. **Compile time**: Compilers choose the mapping of high-level constructs to machine code, including the layout of statically defined data in memory**.**

5. **Link time:** Since most compilers support separate compilation—compiling different modules of a program at different times—and depend on the availability of a library of standard subroutines, a program is usually not complete until the various modules are joined together by a linker. The linker chooses the overall layout of the modules with respect to one another, and resolves intermodule references. When a name in one module refers to an object in another module, the binding between the two is not finalized until link time.

6. Load time: Load time refers to the point at which the operating system loads the program into memory so that it can run.

7**. Run time:** Run time is actually a very broad term that covers the entire span from the beginning to the end of execution. Bindings of values to variable occur at run time, as do a host of other decisions that vary from language to language.

Two types of binding

1. Static

2. Dynamic

The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time, respectively. It is difficult to overstate the importance of binding times in the design and implementation of programming languages. In general, early binding times are associated with greater efficiency.

Later binding times are associated with greater flexibility. Compiled languages tend to have early binding times. Interpreted languages tend to have later binding times.

Ans set (b)

Binding: An association of a name with an object. The operation of associating two things, like a name and the entity it represents. Binding is associate an attribute with an entity.

Examples of attributes are name, type, value.

Binding occurs at various times in the life of a program

The compiler performs a process called binding when an object is assigned to an object variable.

Binding time is the the moment when the binding is performed (compilation, execution, etc).

The early binding (static binding) refers to compile time binding

late binding (dynamic binding) refers to runtime binding

The Concept of Binding

The l-value of a variable is its address.

The r-value of a variable is its value.

A binding is an association, such as between an attribute and an entity, or between an operation and a symbol.

• A binding is static if it first occurs before run time and remains unchanged throughout program execution.

• A binding is dynamic if it first occurs during execution or can change during execution of the program.

Possible Binding Time: Binding Time is the time at which a binding is created

Language design time

Language implementation time

program writing time

compile time

link time

load time

Runtime

Language design time : (bind operator symbols to operations. * to mul) program structure, possible types , control flow constructs are chosen **Language implementation time**: Coupling of I/O to OS, arithmetic overflow, stack size, type equality ,handling of run time exceptions

Ex)A data type such as int in C is bound to a range of possible values

program writing time: Programmers choose algorithms, data structures and names compile time:bind a variable to a particular data type at compile time.

link time: Library of standard subroutines joined together by a linker.

load time: Refers to the point at which the OS loads the program into memory so that it can run. Virtual address are chosen at link time and physical addresses change at run time.bind a variable to a memory cell (ex. C static variables)

Runtime :refers to the entire span from the beginning to the end of execution. Virtual functions, values to variables, many more. bind a nonstatic local variable to a memory cell.

static and dynamic binding:

The terms static and dynamic are generally used to refer to things bound before run time and at run time, respectively

A binding is static if it first occurs before run time and remains unchanged throughout program execution

A binding is dynamic if it first occurs during execution or can change during execution of the program.

Compiler-based language implementations tend to be more efficient than interpreter-based implementations because they make earlier decisions.

3. What are the advantages of user-defined enumeration types?

An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers. In Java, for example, the primitive ordinal types are integer, char, and boolean.

There are two user-defined ordinal types that have been supported by programming languages:

enumeration and subrange

Enumeration Types

An enumeration type is one in which all of the possible values, which are named constants, are provided, or enumerated, in the definition.

Enumeration types provide a way of defining and grouping collections of named constants, which are called enumeration constants.

Examples of primitive ordinal types in Java

– integer

– char

– boolean

C# example:

enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

The design issues for enumeration types are as follows:

• Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant in the program checked?

• Are enumeration values coerced to integer?

• Are any other types coerced to an enumeration type?

Evaluation of Enumerated Type

Aid to readability, e.g., no need to code a color as a number

• Aid to reliability, e.g., compiler can check:

– operations (don't allow colors to be added)

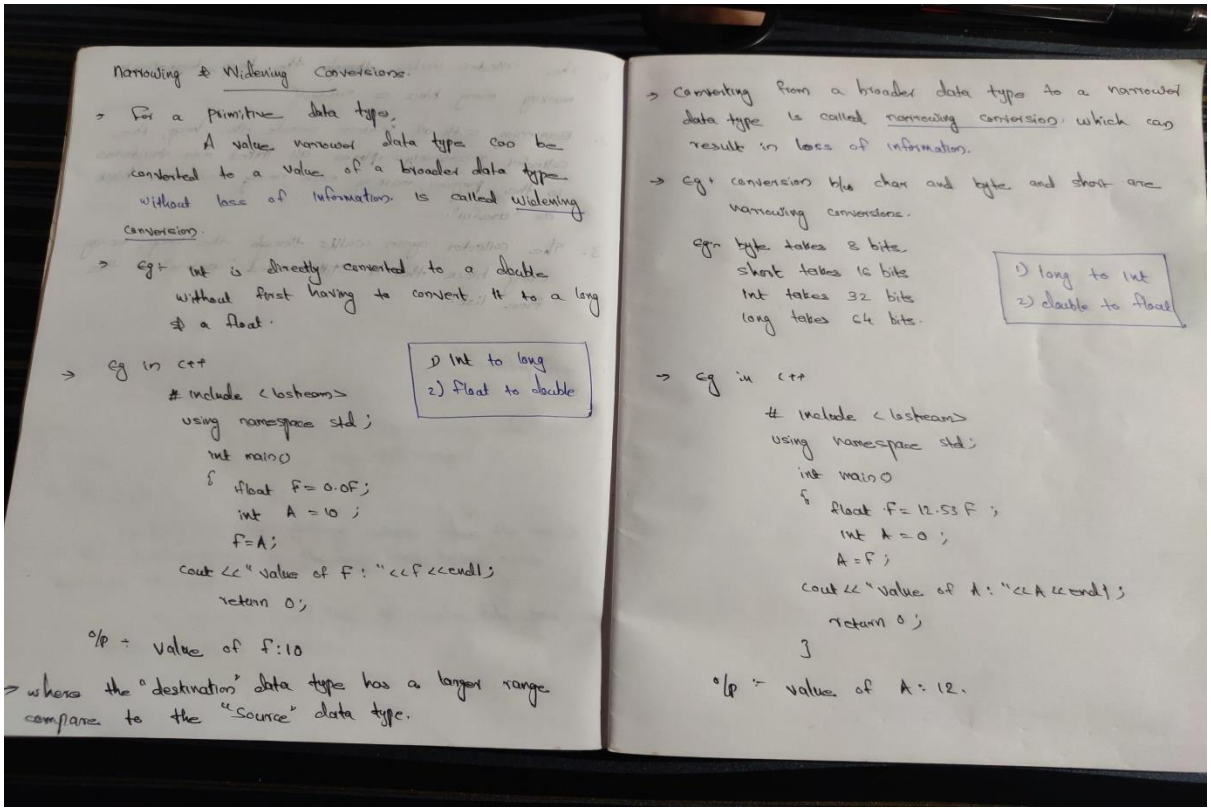
– No enumeration variable can be assigned a value outside its defined range

– Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types.

Implementation of User-Defined Ordinal Types.

Enumeration types are implemented as integers.

4. Define narrowing and widening conversions.



5. Why for statement in C language is more flexible than that of older languages?

a) Because we can use literally anything as its initialization or condition or update expression. We can even omit them and still exit out of the loop. Not only that you can use more than one initialization and/or update expression for a for loop. Also compare to the for or equivalent statemets in order languages, the for in C's condition can be altered even inside the loop. So in C its more flexible, even though it causes confusion and is hard to maintain.

6. What are the advantages and disadvantages of dynamic local variables in subprograms?

A) There are several advantages of stack-dynamic local variables, the primary one being the flexibility they provide to the subprogram. It is essential that recursive subprograms have stack-dynamic local variables. Another advantage of stack- dynamic locals is that the storage for local variables in an active subprogram can be shared with the local variables in all inactive subprograms.

The main disadvantages of stack-dynamic local variables are the following:

First, there is the cost of the time required to allocate, initialize (when necessary), and deallocate such variables for each call to the subprogram.

Second, accesses to stack-dynamic local variables must be indirect, whereas accesses to static variables can be direct. This indirectness is required because the place in the stack where a particular local variable will reside can be determined only during execution.

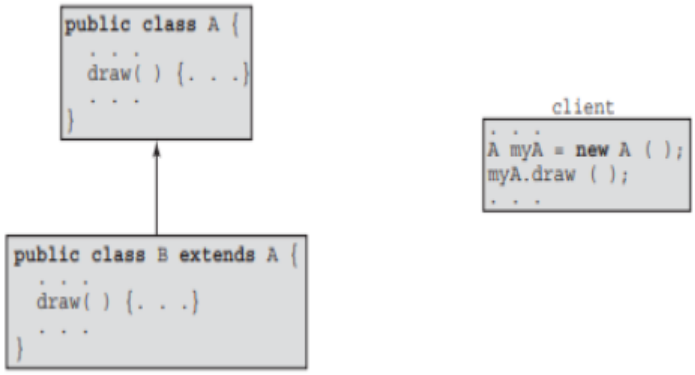
Finally, when all local variables are stack dynamic, subprograms cannot be history sensitive; that is, they cannot retain data values of local variables between calls.

7. Illustrate the concept of dynamic method binding with an example.

The third characteristic (after abstract data types and inheritance) of object oriented programming languages is a kind of polymorphism provided by the dynamic binding of messages to method definitions. This is sometimes called dynamic dispatch.

Consider the following situation:

There is a base class, A, that defines a method draw that draws some figure associated with the base class. A second class, B, is defined as a subclass of A. Objects of this new class also need a draw method that is like that provided by A, but a bit different because the subclass objects are slightly different. So, the subclass overrides the inherited draw method. If a client of A and B has a variable that is a reference to class A's objects, that reference also could point at class B's objects, making it a polymorphic reference If the method draw, which is defined in both classes, is called through the polymorphic reference, the run-time system must determine, during execution, which method should be called, A's or B's (by determining which type object is currently referenced by the reference).



Polymorphism is a natural part of any object-oriented language that is statically typed.

In a sense, polymorphism makes a statically typed language a little bit dynamically typed, where the little bit is in some bindings of method calls to methods. The type of a polymorphic variable is indeed dynamic.

One purpose of dynamic binding is to allow software systems to be more easily extended during both development and maintenance.

In some cases, the design of an inheritance hierarchy results in one or more classes that are so high in the hierarchy that an instantiation of them would not make sense.

Shot note

A polymorphic variable can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants

• When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method will be dynamic

• Allows software systems to be more easily extended during both development and maintenance

An abstract method is one that does not include a definition (it only defines a protocol)

• An abstract class is one that includes at least one virtual method

• An abstract class cannot be instantiated

8. Is it mandatory to use constructors in object-oriented languages? Justify your answer.

A) No. Users do not need to write constructors for every class. A constructor can be declared using any of the access modifiers. It is mandatory to have a constructor with the right access modifier. However, the compiler supplies default if an access modifier is not defined in the class and a constructor is not declared.

9. What are the applications of logic programming languages?

Logic programming can be used in any domain where a large amount of data must be analyzed to make decisions. However, it is most commonly applied to a few subjects. Following are some places where logic programming is most likely to be found.

Artificial Intelligence/Machine Learning: This is one of the main applications of logic programming. It is especially relevant because it provides a structured method of defining domain-specific knowledge. AI systems use their facts and rules to analyze new queries and statements.

Natural Language Processing (NLP): NLP handles interactions between people and computers. It relies upon a system of rules to interpret and understand speech or text. NLP systems translate their insights back into a more data-friendly format. NLP systems can also generate a relevant response to user requests and feedback.

Database Management: Logic programming can determine the best place in a database to store new data. It can also analyze the contents of a database and retrieve the most useful and relevant results for a query. Logic programming is frequently used with large freeform NoSQL databases. These databases do not use tables to organize and structure data and must be analyzed using other methods.

Predictive Analysis: Logic programs can sort through a large amount of data, analyze results and make predictions. This is especially useful in areas

such as climate forecasting, the monitoring of deep space objects, and predicting equipment failures.

Logic programming is also used in fault diagnosis, pattern matching, and mathematical proofs**.**

10. Explain the working of let and let-rec constructs in Scheme.

LET

• General form: (LET ((name_1 expression_1) (name_2 expression_2) ...

(name_n expression_n)) body)

• Evaluate all expressions, then bind the values to the names; evaluate the body

• Each binding of a has as its region.

(let ((x 2) (y 3))

(* x y)) = 6

(let ((x 3) (y (+ x 1)))

(+ x y)) = Error : unbound symbol x Eg: (let ((a 3) (b 4)

(square (lambda (x) (* x x)))

(plus +))

(sqrt (plus (square a) (square b)))) ⇒ 5.0

Eg: (let ((x 2) (y 3))

(let ((x 7) (z (+ x y)))

(* z x))

)

= 35

Let*

• Let* is similar to let, but the bindings are performed sequentially from left to

right, and the region of a binding indicated by (<variable> <init>) is that part of

the let* expression to the

right of the binding. Thus the second binding is done in an environment in

which the first binding is visible, and so on.

(let ((x 2) (y 3))

(let* ((x 7) (z (+ x y)))

(* z x)))

=70

11. (a) Explain different criteria used for evaluating languages.

The language evaluation criteria are:

Readability: the ease with which programs can be read and understood.

Writability: the ease with which a language can be used to create programs.

Reliability: conformance to specifications (i.e., performs to its specifications).

Cost: the ultimate total cost.

We will look at them in details:

a) Readability - It is the ability to understand the program, and then modify it.

Readability has strong relation with the language of choice and the application

domains. Readability involves:

i. Simplicity

• If it is simple, it is easy to learn.

• Programmers usually use the features they are familiar with, which are

not necessarily the ones that the readers of the program are familiar with.

• Most often a large number of features will do the same thing, causing

unnecessary confusion. That is, multiplicity of features does not help.

Example: i = i + 1; i+ = 1; i + +; + + i;

• Operator overloading gives an operator multiple meanings, which may be confusing when done incorrectly. For example, in some languages, + is used for integer addition, floating point addition, decimal addition and string concatenation

• Assembly languages are simple, but are too low-level (or not “sophisticated” enough) to express our ideas.

ii. Orthogonality

• Features can be combined in a systematic way to form new features.

• Orthogonality means independent, i.e., if you can do one thing with a feature, there is no reason that you cannot do the same to the other. In a sense the features are symmetric.

• If a language is more orthogonal, it has fewer exceptions, and may become simpler, but may have unnecessary complexity since the number of legal constructs would be very large.

• VAX and IBM addition example.

- Orthogonality requires every possible combination of primitives to be legal. Example: 32bit addition from IBM/VAX assembly languages:

IBM

A reg, mem; adds mem into reg

AR reg1, reg2; adds reg2 into reg1

VAX

addl op1, op2 ; op1 and op2 are any reg/mem

The VAX instruction set exhibits orthogonality

• Simplicity is the result of a combination of a small number of features through Orthogonality.

iii. Control Statements

• Structured programming proclaims blocked control structures with single entry and exit, and discourages the use of goto statements.

• Goto statements should be used with caution. They should precede their destination, and never jump to distant labels.

• Limited use of goto statements is still unavoidable in certain language like FORTRAN r BASIC.

iv. Data Types and Structures

• Sufficient data type and data structure support not only improve readability, but also expressibility.

• Adequate facilities for defining data types and structures aids readability.

• Primitive/intrinsic data types should be adequate, too.

v. Syntax Considerations

• Identifier form - should not be too restrictive on length. The length and valid combination of characters.

• Special words - Words such as while, if, end, class, etc., have special meaning within a program. How to express compound statement? Can special words be used as identifiers?

• Form and meaning - How a statement appears should clear indicate what it actually means.

b) Writability - Does the language make it easy to write what you have in mind?

Readability first, then writability. How easily can the language be used to create programs for a particular domain? Consider Visual Basic (VB) and C for – A graphical game program, – An embedded controller for automotive brakes.

i. Simplicity and Orthogonality

• The combination of a small number of features (simplicity) and the consistent way to combine them to form new features (orthogonality) is crucial.

ii. Support for Abstraction

• Abstraction hides the details (in implementation) of a construct and only provides a clear interface of how this construct should be used.

It specifies what a construct works, not how it works.

• Procedure or Process abstraction isolate what a process does from how the process does it. Procedural or process abstraction is a specification that:

- Describes effects on outputs for given inputs

- what it does, not how it does it

- ignores implementation

- Treats procedure or function as a “black box”

- Can be applied in any language. Classic example: sorting, that is:

We just want to call a sort routine when we need to sort something.

We don’t want to clutter up code by implementing a sort algorithm every time.

• Data abstraction separate what an object should behave given certain events from how it implements the actions.

iii. Expressivity

• The ability to handle data in meaningful, natural ways.

• The language should have convenient way to express computation, e.g., goto may be sufficient, but while loop is better.

• Abstract Data Types:

Extend procedural abstraction to data. Example: type float.

Extends imperative notion of type by:

Providing encapsulation of data/functions.

Separation of interface from implementation.

c) Reliability

Reliability is the property of performing to specifications under all conditions.

Many design considerations contribute to reliability. The question here is “Will the program crash easily”?

i. Type Checking

• Is the process of verifying and enforcing the constraints of types, and it can occur either at compile time (i.e. statically) or at runtime (i.e. dynamically). That is, checks errors either at runtime or compile time.

• A language is statically-typed if the type of a variable is known at compile time instead of at runtime. Common examples of statically-typed languages include Ada, C, C++, C#, Java, Fortran, Haskell, ML, Pascal, and Scala.

• Dynamic type checking is the process of verifying the type safety of a program at runtime. Common dynamically-typed languages include Groovy, JavaScript, Lisp, Lua, Objective-C, PHP, Prolog, Python, Ruby, Smalltalk and Tcl.

• Runtime checks are significantly more expensive.

• Early detection is less expensive than correcting released code - But forcing early detection of type errors reduces writability and expressiveness.

- Many scripting languages freely cast types at runtime which can result in bizarre and difficult to detect errors later.

• Compile-time type checking is cheaper, but for many dynamic execution languages, run-time checking is necessary.

• Type checking includes arguments in function calls, assignment, index range in certain languages (e.g., PASCAL).

ii. Exception Handling

Refers to built-in mechanisms to intercept runtime errors, handle them and continue normal execution

Possible in any language, but some languages have better facilities than others.

iii. Aliasing

To have more than one method to access the same memory.

Two or more distinct names that refer to the same memory location or object.

Widely regarded as a fruitful source of error.

Difficult to detect and almost impossible to prevent.

iv. Readability and Writability

A reliable program is written with the best method, the best method is implemented only when we understand what exactly we want to do.

A language that does not support natural ways of expressing an algorithm will require the use of “unnatural” approaches, and hence reduced reliability.

It is possible to write unreadable code in any language.

d) Cost Cost to train the programmers Simplicity and orthogonality.

Cost to write programs Writability

Cost to compile the program Quick-and-dirty or highly optimized.

Cost to execute the program Most likely determined by what language you use.

Cost to build the compiler and runtime support system

The complexity to build compiler should be considered.

Cost to recover from problems

Cost to maintain the program Readability, Most of the software costs

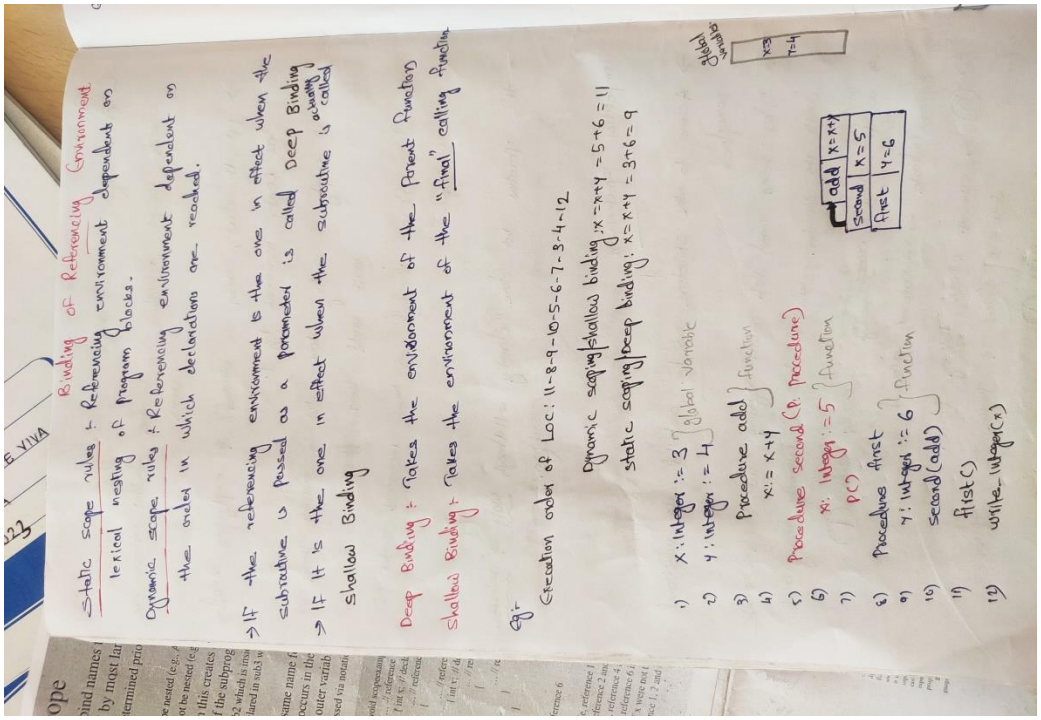
are in maintenance, not developments.

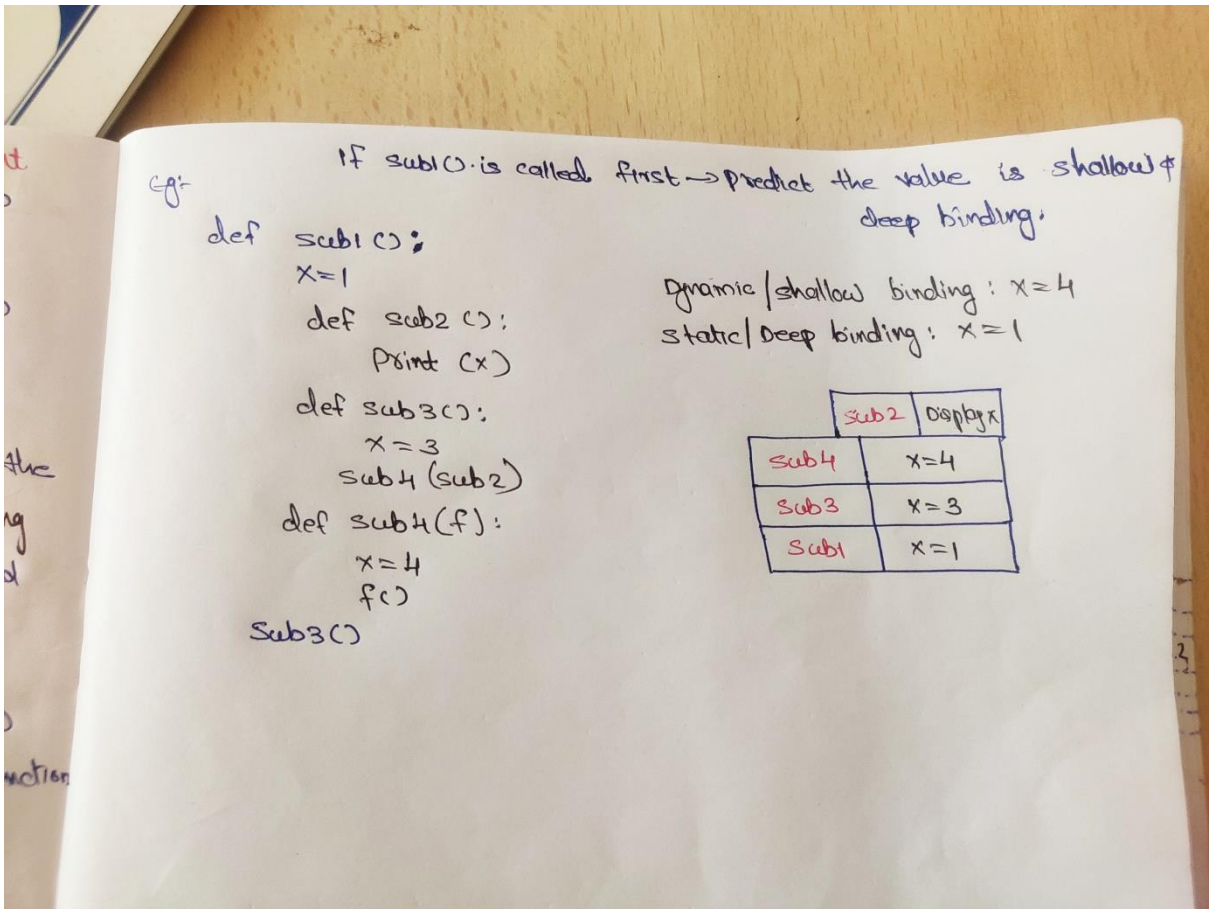
(b) Consider the following pseudocode:

```
x : integer := 3
y : integer := 4
procedure add
x := x + y
procedure second(P : procedure)
x : integer := 5
P()
procedure first
y : integer := 6
second(add)
first()
write integer(x)
```

(a) What does this program print if the language uses static scoping? Give reasons.

(b) What does it print if the language uses dynamic scoping? Give reasons.





12.(a) With respect to storage binding, explain the meanings, purposes, advantages and disadvantages of four categories of scalar variables.

Object Storage Management

➢ An object has to be stored in memory during its lifetime.

➢ Storage Allocation mechanisms

i. Static

ii. Stack

iii. Heap

➢ Allocation - getting a cell from some pool of available cells.

➢ Deallocation - putting a cell back into the pool.

➢ The lifetime of a variable is the time during which it is bound to a particular memory cell. So the lifetime of a var begins when it is bound to a specific cell and ends when it is unbound from that cell.

➢ Categories of variables by lifetimes:

1. static

2. stack-dynamic,

3. explicit heap-dynamic

4. implicit heap-dynamic

Static variables

➢ Bound to memory cells before execution begins and remains bound to the same memory cell throughout execution.

➢ E.g. all FORTRAN 77 variables, C static variables.

➢ Advantages:

Efficiency: (direct addressing): All addressing of static vars can be direct. No run-time overhead is incurred for allocating and deallocating vars.

History-sensitive: vars retain their values between separate executions of the subprogram.

➢ Disadvantage:

o Storage cannot be shared among variables.

o Eg: if two large arrays are used by two subprograms, which are never active at the same time, they cannot share the same storage for their arrays.

Stack-dynamic variables

➢ Storage bindings are created for variables when their declaration statements are elaborated, but whose types are statically bound.

➢ Elaboration of such a declaration refers to the storage allocation and binding process indicated by the declaration, which takes place when execution reaches the code to which the declaration is attached.

➢ Eg: The variable declarations that appear at the beginning of a Java method are elaborated when the method is invoked and the variables defined by those declarations are deallocated when the method completes its execution.

➢ Stack-dynamic variables are allocated from the run-time stack.

➢ If scalar, all attributes except address are statically bound.

➢ Eg: • Local variables in C subprograms and Java methods.

➢ Advantages:

o Allows recursion: each ac tive copy of the recursive subprogram has its own version of the local variables.

o In the absence of re cursion it conserves storage because all subprograms share the same memory space for their locals.

➢ Disadvantages:

o Overhead of allocation and deallocation.

o Subprograms cannot be history sensitive.

o Inefficient references (indirect addressing) is required because the place in the stack where a particular var will reside can only be determined during execution.

➢ In Java, C++, and C#, variables defined in methods are by default stack dynamic.

Explicit Heap-dynamic variables

➢ Nameless memory cells that are allocated and deallocated by explicit directives "run-time instructions", specified by the programmer, which take effect during execution.

➢ These vars, which are allocated from and deallocated to the heap, can only be referenced through pointers or reference variables.

➢ The heap is a collection of storage cells whose organization is highly disorganized because of the unpredictability of its use.

➢ Eg. dynamic objects in C++ (via new and delete)

```
int *intnode;
…
intnode = new int; // allocates an int cell
… delete intnode; // deallocates the cell to which
                  // intnode points
```

➢ An explicit heap-dynamic variable of int type is created by the new operator.

➢ This operator can be referenced through the pointer, intnode.

➢ The var is deallocated by the delete operator.

➢ Java, all data except the primitive scalars are objects.

➢ Java objects are explicitly heap-dynamic and are accessed through reference variables.

➢ Java uses implicit garbage collection.

➢ Explicit heap-dynamic vars are used for dynamic structures, such as linked lists and trees that need to grow and shrink during execution.

➢ Advantage

o Provides for dynamic storage management.

➢ Disadvantage

o Inefficient "Cost of allocation and deallocation" and unreliable "difficulty of using pointer and reference variables correctly".

Implicit Heap-dynamic variables

➢ Bound to heap storage only when they are assigned value. Allocation and deallocation caused by assignment statements.

➢ All their attributes are bound every time they are assigned. E.g. all variables in APL; all strings and arrays in Perl and JavaScript.

➢ Advantage

o Flexibility allowing generic code to be written.

➢ Disadvantages

o Inefficient, because all attributes are dynamic "run-time." Loss of error detection by the compiler

(b) What is meant by referencing environment of a statement? Show the (7)

referencing environment at the indicated program points (1), (2), (3) & (4) for the following program segment. Assume that the programming language is statically scoped.

```
program example;
var a, b : integer;
procedure sub1;
```

```
var x, y: integer;
 begin { sub1 }
                    ..... (1)
 end { sub1 }
procedure sub2;
var x : integer;
            .....
 procedure sub3;
var x: integer;
begin { sub3 }
                    ..... (2)
 end { sub3 }
begin { sub2 }
                    ..... (3)
 end { sub2}
begin {example}
                    ..... (4)
end {example }
```

A) The referencing environments of the indicated program points are as follows:

Point Referencing Environment

1	X and Y of Sub1, A & B of Example
2	X of Sub3, (X of Sub2 is hidden), A and B of Example
3	X of Sub2, A and B of Example
4	A and B of Example

➢ Eg, dynamic-scoped language

Consider the following program; assume that the only function calls are the following: main calls sub2, which calls sub1

```
void sub1( )
{
int a, b;
                        … 1
} /* end of sub1 */
void sub2( )
{
int b, c;
                         … 2
sub1;
} /* end of sub2 */
void main ( )
{
int c, d;
                … 3
sub2( );
} /* end of main */
```

The referencing environments of the indicated program points are as follows:

Point Referencing Environment

1 a and b of sub1, c of sub2, d of main

2 b and c of sub2, d of main

3 c and d of main