

B.Tech.

Fourth Semester Examination

Programming Languages (CSE-204F)

Short Answer Type Questions

Q. 1. (a) Define data object.

Ans. A data object represents a container for data values, a place where data values may be stored and later retrieved. A data object is characterized by a set of attributes, the most important of which is its data type. The attributes determine the number and type of values that the data object may contain and also determine the logical organisation of those values.

Q. 1. (b) Write a short note on need of translator.

Ans. The general term translator denotes any language processor that accepts programs in some source language (which may be high or low level) as input and produces functionally equivalent programs in another object language (which may also be high or low level) as output.

Q. 1. (c) What is information hiding?

Ans. Information hiding is the term used for a central principle in the design of programmer-defined abstraction. Each such program component should hide as much information as possible from the users of the component. Thus, the language provided square-root function is a successful abstract operation because it hides the details of the number representation and the square-root computation algorithm from the user.

Q. 1. (d) Define the term type definitions.

Ans. In defining a complete new abstract data type, some mechanism is required for definition of a class of data objects. In languages such as C, Pascal and Ada, this mechanism is termed as type definition but a type definition does not define a complete abstract data type because it does not include definition of the operations on data of that type.

Q. 1. (e) What is heap storage management?

Ans. Heap storage management is a block of storage within which pieces are allocated and freed in some relatively unstructured manner. In heap storage management, storage allocation, recovery, compaction and reuse may be severe. There is no single heap storage management technique, but rather a collection of techniques for handling various aspects of managing this memory.

Q. 1. (f) What are procedural languages?

Ans. Procedural languages consists of a series of procedures (or subprograms or functions or subroutines) that execute when called. Each procedure consists of a sequence of statements, where each statement manipulates data that may either be local to the procedure, a parameter passed in from the calling procedure or defined globally.

Q. 1. (g) Define Recursive Subprograms.

Ans. Recursion, in the form of recursive subprograms calls, is one of the most important sequence-control structures in programming. Many algorithms are most naturally represented using recursion. In LISP, where list structures are the primary data structure available, recursion is the primary control mechanism for repeating of sequences of statements, replacing the iteration of most other languages.

Q. 1. (h) Write a short note on dynamic scope.

Ans. The dynamic scope of an association for an identifier, is the set of subprogram activations in which

the association is visible during execution. The dynamic scope of an association always includes the subprogram activation in which that association is created as part of local environment. It may also be visible as a non-local association in other subprogram activation.

Q. 1. (i) Define the concept of block structure.

Ans. The concept of block structure as found in block structured languages such as Pascal, PL/I and Ada deserves special mention. Block structured languages have a characteristic program structure and associated set of static scope rules. The concepts originated in the language ALGOL 60, one of the most important early languages and because of their elegance and effect on implementation efficiency, they have been adopted in other languages.

Q. 1. (j) Define character string.

Ans. A character string is a data object composed of a sequence of characters. A character string data type is important in most languages, owing in part to the use of character representations of data for input and output.

Q. 2. What are the characteristics of a good programming language?

Ans. There are various factors, why the programmers prefer one language over another. Some of the important reasons for a good programming language are given as follows :

(i) Clarity, Simplicity and Unity : A programming language provides both a conceptual framework for thinking about algorithm and a means of expressing these algorithms. The language should be an aid to the programmer long before the actual coding stage. It should provide a clear, simple and unified set of concepts that can be used as primitives in developing algorithm.

(ii) Orthogonality : The term orthogonality refers to the attribute of being able to combine various features of a language in all possible combinations, with every combination being meaningful.

(iii) Support for Abstraction : Even with the most natural programming language for an application, there is always a substantial gap remaining between the abstract data structure and operations that characterize the solution to a problem and the particular primitive data structure and operations built into a language.

(iv) Ease of Program Verification : The reusability of programs written in a language is always a central concern. There are many techniques for verifying that a program correctly performs its required function. A program may be proven correct by a formal verification method, it may be proven correct by desk checking, it may be tested by executing it with test input data and checking the output results against the specifications etc.

(v) Programming Environment : The technical structure of a programming language is only one aspect affecting its utility. The presence of an appropriate programming environment may make a technically weak language easier to work with than a stronger language that has little external support. A long list of factors might be included as part of the programming environment.

(vi) Portability of Programs : One important criterion for many programming projects is that of the transportability of the resulting programs from the computer on which they are developed to other computer system. A language that is widely available and whose definition is independent of the features of a particular machine forms a useful base for the production of transportable programs.

Q. 2. (b) What is general syntactic criteria of a programming language?

Ans. The primary purpose of a syntax is to provide a notation for communication between the programmer and the programming language processor. The choice of particular syntactic structures, however is constrained only slightly by the necessity to communicate particular items of information.

There are many secondary criteria, but they may be roughly categorized under the general goals of making programs easy to read, easy to write, easy to translate and unambiguous. Some of the ways that

language syntactic structure may be designed to satisfy these criteria are given as follows :

(i) Readability : A program is readable if the underlying structure of algorithm and data represented by the program is apparent from an inspection of the program text. A readable program is often said to be self-documenting. It is understandable without any separate documentation.

(ii) Writeability : The syntactic feature that make a program easy to write are often in conflict with those features that make it easy to read. Writeability is enhanced by use of concise and regular syntactic structures, while for readability a variety of more "verbose" constructs are helpful. C unfortunately has the attribute of providing for very concise programs that are hard to read, although it does have a full complement of useful features.

(iii) Ease of Verifiability : Relates to readability and writeability is the concept of program correctness or program verification. After many years of experience, we understand that while understanding each programming language statement is relatively easy, the overall process of creating correct program is extremely difficult. Therefore we need techniques that enable the program to be mathematically proven correct.

(iv) Ease of Translation : A conflicting goal is that of making programs easy to translate into executable form. Readability and writeability are criteria directed to the needs of the human programmer. Ease of translation relates to the needs of the translator that processes the written program.

(v) Lack of Ambiguity : Ambiguity is a central problem in every language design. A language definition ideally provides a unique meaning for every syntactic construct that a programmer may write. An ambiguous construction allows two or more different interpretations.

Q. 3. (a) Explain implementation of elementary data types.

Ans. The implementation of an elementary data type consists of a storage representation for data objects and values of that type, and a set of algorithms or procedures that define the operations of the type in terms of manipulations of the storage representation.

Storage Representation : Storage for elementary data type is strongly influenced by the underlying computer that will execute the program. For example, the storage representation for integer or real values is almost always the integer or floating point binary representation for numbers used in underlying hardware. The reason for this choice is simple. If the hardware storage representation are used, then the basic operations on data of that type may be implemented using the hardware provided operation. If the hardware storage representations are not used, then the operations must be software simulated, and the same operation will execute much less efficiently.

Implementation of Operations : Each operation defined for data object of a given type may be implemented in one of three main ways :

(i) Directly as a Hardware Operation : For example, if integers are stored, using the hardware representation for integers, then addition and subtraction may be implemented using the arithmetic operations built into the hardware.

(ii) As a Procedure and Function Subprogram : For example, a square root operation is usually not provided directly as a hardware operation. It might be implemented as a square root subprogram that calculates the square root of its argument. If data objects are not represented using a hardware-defined representation, then all operations must be software simulated, usually in the form of subprograms provided in a subprogram library.

(iii) As an In-Line Code Sequence : As in-line code sequence is also a software implementation of the operation, but instead of using a very short subprogram, the operations in the sub-program are copied into the program at the point where the subprogram would otherwise have been invoked. For example, the absolute value function on number, defined by :

$\text{abs}(x) = \text{if } x < 0 \text{ then } -x \text{ else } x$

is usually implemented as an in-line code sequence :

- (a) Fetch value of x from the memory.
- (b) If $x > 0$, skip the next instruction.
- (c) Set $x = -x$.
- (d) Store new value of x in memory.

Where each line is implemented by a single hardware operation.

Q. 3. (b) Explain the concept of assignment and initialization.

Ans. Most of the operations for the common elementary data types, in particular numbers, enumerations, booleans & characters, take one or two argument data objects of the type, perform a relatively simple arithmetic, relational or other operation, and produce a result data object, which may be of the same or different type. The operation of assignment, however, is somewhat more subtle and deserves special mention.

Assignment is the basic operation flux checking the binding of a value to a data object. This change, however is a side effect of the operation. In some language such as C & LISP, assignment also 'return a value' which is a data object containing a copy of the value assigned. There factors become clear when we try to write a specification flux assignment. In Pascal the specification for assignment of integer would be :

$\text{assignment} (:=) : \text{integer}_1 \times \text{integer}_2 \rightarrow \text{void}$

With the action : Set the values contained in data object integer_1 to be a copy of the value contained in data object integer_2 and return no explicit result. (The change to integer_1 is an implicit result or side effect). In C, the specification is :

$\text{assignment} (=) : \text{integer}_1 \times \text{integer}_2 \rightarrow \text{integer}_3$

With the Action : Set the values contained in data object integer_1 to be a copy of the value contained in data object integer_2 and also create and return a new data object integer_3 , containing a copy of the new value of integer_2 .

Initialization : An uninitialized variable, as more generally an uninitialized data object, is a data object that has been created but not yet assigned a value. Creation of a data object ordinarily involves only allocation of a block of storage. Without any further action, the block of storage retains whatever bit pattern it happened to contain when the allocation was made. An explicit assignment is ordinarily required to bind a data object to valid value. In some languages (e.g., Pascal) initialization must be done explicitly with assignment statements. In the languages (e.g., APL) initial value for each data object must be specified when the object is created, the assignment of initial values is handled implicitly without use of the assignment operation by the programmer.

Q. 3. (c) Explain the concept of Boolean & characters.

Ans. Most languages provide a data type for representing true and false, usually called a Boolean or logical data type.

The Boolean data type consists of data objects having one of two values, true or false. In Pascal and Ada, the Boolean data type is considered simply a language defined enumeration :

$\text{type Boolean} = (\text{false}, \text{true});$

Which both defines the name true and false for the values of the type and defines the ordering $\text{false} < \text{true}$.

The most common operation on Boolean type include assignment as well as :

$\text{and} : \text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean} \text{ (conjunction)}$

$\text{or} : \text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean} \text{ (inclusive disjunction)}$

not : Boolean \rightarrow Boolean (negative or complement)

Other Boolean operations such as equivalence, exclusive or implication, nand (not-and) and nor (not-or) are sometimes included.

The storage representation of a Boolean data object is a single bit of storage, provides no descriptor designating the data type is needed. Because single bits may not be separately addressable in memory.

Characters : Most data are input and output in character form. Conversion during input and output to other data types is usually provided, but processing of some data directly in character form is also important. Sequence of characters (character strings) are often processed as a unit. Provision for character-string data may be provided either directly through a character-string data type (as in ML and Prolog) or through a character data type, with a character string considered as a linear array of characters (as in C, Pascal and Ada).

Specification : A character data type provides data objects that have a single character as their value. The set of possible character values is usually taken to be a language-defined enumeration corresponding to the standard character sets supported by the underlying hardware and operating system, such as the ASCII character set.

Q. 4. (a) Write the specification of structured data type.

Ans. The major attributes for specifying structured data type are given as follows :

(i) **Number of Components :** A data structure may be of fixed size if the no. of components is invariant during its lifetime, as the variable size if the number of components changes dynamically. Variable size data structure types usually define operations that insert and delete components from structure. Array and records are common examples of fixed size data structure types; stacks, list, sets, tables and files are example of variable-size types.

(ii) **Type of Each Component :** A data structure is homogeneous if all its components are of the same type. It is heterogeneous if the components are of different types. Array, character strings, sets and files are usually homogeneous; records and lists are usually heterogeneous.

(iii) **Name to be Used for Selecting Components :** A data structure type needs a selection mechanism for identifying individual components of the data structure. For an array, the name of an individual component may be an integer subscript or sequence of subscripts; for a table, the name may be a programmer-defined identifier. Some data structure types such as stacks and files allow access to only a particular component at any time, but operations are provided to change the component that is currently accessible.

(iv) **Maximum Number of Components :** For a variable size data structure such as a character string as stack, a maximum size for the structure in terms of number of components may be specified.

(v) **Organisation of the Components :** The most common organisation is a simple linear sequence of components. Vectors (one-dimensional array), records, character strings, stacks, list and files are data structure with this organisation. Array, record and list types, however, also usually are extended to multidimensional forms : multi-dimensional array, records whose components are record and lists whose components are lists. Thus, extended forms may be treated as separate types or simply as the basic sequential type in which the components are themselves data structure of similar type. For example, A two dimensional array (matrix) may be considered as a separate type (as in FORTRAN's A (i, j)), as a "vector of vectors" (as in C's A[i] [j]), a vector in which the components (the rows or columns) are themselves vectors.

Q. 4. (b) Discuss declaration and type checking for data structure.

Ans. The basic concepts and concerns surrounding declarations and type checking for data structure are similar to those discussed for elementary data objects, however, structures are ordinarily more complex because there are more attributes to specify.

For example, the Pascal declaration at the beginning of a subprogram P;

A : array [1.....10, -5.5] of real;

Specifies the following attributes of array A :

- (i) Data type is an array.
- (ii) Number of dimensions is two.
- (iii) Subscripts naming the row are the integer from 1 to 10.
- (iv) Subscripts naming the columns are the integer from -5 to 5.
- (v) Number of components is 110 (10 rows × 11 columns)

Data type of each component is real.

Declaration of these attributes allow a sequential storage representation for A and the appropriate accessing formula for selecting and component A [i, j] of A to be determined at compile time.

Type checking is somewhat more complex data structure because component selection operations must be taken into account. There are two main properties :

(i) **Existence of Selected Component** : The arguments to a selection operation may be of right types but the component designated may not exist in the data structure. For example, a subscripting operation that selects a component from an array may receive a subscript that is "out of bounds" for the particular array i.e., that produces an invalid P-value for the array component. This is not a type checking problem in itself, provided that the selection operation fails "gracefully" by noting the error and raising an exception, e.g., a "subscript range error."

(ii) **Type of Selected Component** : A selection sequence may define a complex path through a data structure to the desired component. For example, the C :

A [2] [3]-> item

Selects the contents of component named "item" in the record reached via the pointer contained in the component named "link" of the record that is the component in row 2 and column 3 of array A. To perform static type checking, it must be possible to determine at compile time the type of the component selected by any valid composite selector of this sort. Static type checking guarantees only that if the component does exist, then it is of the right type.

Q. 5. (a) What are Records? Define specification and syntax of records.

Ans. A data structure composed of a fixed number of components of different types is usually termed as records.

Specification & Syntax : Both records and vectors are forms of fixed length linear data structure but records differ in two ways :

- (i) The components of records may be heterogeneous of mixed data types, rather than homogeneous.
- (ii) The components of records are named with symbolic names (identifiers) rather than indexed with subscripts. The C syntax for a record declaration (struct in C) is fairly typical.

```
struct_Employee type
{
    int ID;
    int age;
    float SALARY;
    char Dept;
} Employee;
```


The declaration defines a record of type Employee Type consistency of four components of types integer, integer, real and character, with component names ID, Age, SALARY and Dept respectively. Employee is declared to be a variable of type Employee Type.

To select a component of the record, one write in C;

Employee ID

Employee SALARY

.....

The attributes of a record are seen in above declaration :

- (i) The number of components.
- (ii) The data type of each component.
- (iii) The selector used to name each component.

The component of a record often called fields and the component names than all the field names. Records are sometimes called structures in C.

Component selection is the basic operation on a record, as in the selection Employee SALARY. This operations corresponds to the subscripting operation for arrays, but with one crucial difference. The "subscript" here is always a literal component name ; it is never a computed value.

Operations on entire records are usually few. Most commonly assignment of records of identical structure is provided. e.g.,

struct Employee Type INPUTREC;

.....

Employee = INPUT REC

Where INPUT REC has the same attributes as Employee. The correspondence of component names between records is also made the basis for assignment in COBOL and PL/I e.g., in the COBOL.

Q. 5. (b) Discuss the evaluation of the data type concept.

Ans. Since actual computers ordinarily include no provision for defining and enforcing data type restrictions (e.g., the hardware cannot tell the difference between a bit string representing a real number, a bit string representing a character, or a bit string representing an integer), higher level languages provides a set of basic data types, such as real, integer and character string. Type checking is provided to ensure that operations such as + or × are not applied to data of the wrong type. The early notion of data type defines a type as a set of values that a variable might take on. Data types are associated directly with individual variables so that each declaration names a variable and define it types. If a program uses several arrays, each containing, 20 real numbers, each array is declared separately and the entire array description is repeated for each.

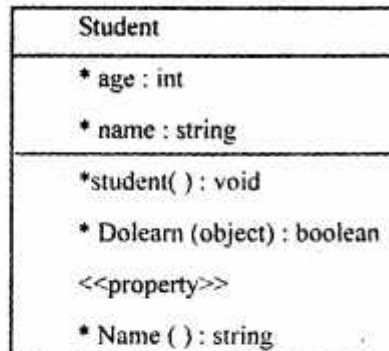
Pascal around 1970, extended the concept to a general type definition applicable to a set of variables. A type definition defines the structure of a data object of the defined type, a declaration requires only the variable name and the name of the type to be given.

During the early 1970's the concept of data types was extended beyond just a set of data objects to also include the set of operations that manipulate those data objects. For primitive types such as real and integer, a language provide a facility to declare variable of that type and a set of operations on real and integers that represent the only way that real and integers can be manipulated by the programmer. Thus, the storage representation of real and integers is effectively encapsulated; e.g., it is hidden from the programmer.

The programmer may use real and integer data objects without knowing, or caring, exactly how they are represented in storage. All the programmer sees is the name of the type and the list of operations available for manipulating data objects of that type.

Q. 5. (c) Define encapsulation & information hiding.

Ans. The encapsulation is the inclusion within a program object of all the resources need for the object to function basically, the methods and data. In OOP the encapsulation is mainly by creating classes, the classes expose public methods and properties. The class is kind of a container or capsule or cell, which encapsulate the set of methods, attribute and properties to provide its indented functionalities or other classes. In that sense, encapsulation also allows a class to change its internal implementation without hurting the overall functioning of the system. That idea of encapsulation is to hide how a class does it but to allow requesting what to do.



In order to modularize/define the functionality of a one class, the class can use function/properties exposed by another class in many different ways. There are several other ways that are encapsulation can be used, as an example we can take the usage of an interface. The interface can be used to hide the information of an implemented class.

Q. 6. (a) Define implicit and explicit sequence control.

Ans. Sequence control structures may be conveniently categorized in three groups :

- (i) Structure used in expressions (and thus within statements, since expressions form the basic building block for statements), such as precedence rules and parentheses;
- (ii) Structures used between statements or groups of statements, such as conditional and iteration statements and
- (iii) Structures used between subprograms, such as subprogram calls and coroutines.

This division is necessarily somewhat imprecise. For example, some languages such as LISP and APL, have no statements, only expressions, yet versions of the usual statement sequence-control mechanisms are used.

Sequence, control structures may be either implicit or explicit. Implicit (or default) sequence control structures are those defined by the language to be in effect unless modified by the programmer through some explicit structure. For example, most languages defines the physical sequence of statements in a program as controlling the sequence in which statements are executed unless modified by an explicit sequence-control statement, within expression there is also commonly a language-defined hierarchy of operations that controls the order of execution of the operations in the expression when parentheses are absent.

Explicit sequence-control structure are those that the programmer may optionally use to modify the implicit sequence of operations defined by the language as e.g., by using parentheses within expressions or goto statements and statements labels.

Q. 6. (b) What are local data and local referencing environments?

Ans. The local environment of a subprogram Q consists of the various identifiers declared in the head of subprogram Q (but not Q itself). Variable names, formal parameters name and subprogram names are the concern here. The subprogram names of interest are the name of subprograms that are defined locally within Q (i.e., subprograms whose definition are nested within Q).

For local environment, static and dynamic scope rules are easily made consistent. The static scope rule specifies that a reference to an identifier X in the body of subprogram Q is related to the local declaration of X in the head of subprogram Q. The dynamic scope rule specifies that a reference to X during execution of Q refers to the association for X in the current activation of Q. To implement the static scope rule, the compiler simply maintains a table of the local declarations for identifiers that appear in the head of Q and while compiling the body of Q, it refers to this table first whenever the declaration of an identifier is required. Implementation of the dynamic scope rule may be done in two ways, and each gives a different semantics to local references.

Consider subprograms P, Q and R with a local variable X declared in Q in figure. Subprogram P calls Q which in turn calls R, which later returns controls to Q, which completes its execution and returns control to P.

```
procedure R;
.....
end;
procedure Q;
var X : integer := 30;    --initial value of X is 30
begin
    write (X);            -- print value of X
    R;                    --Call subprogram R
    X := X + 1;            --increment value of X
    write (X)              --print value of X again
end;
procedure P;
.....
    Q;                    --Call subprogram Q.
.....
end;
```

Q. 7. (a) Write the concepts of names & referencing environments.

Ans. There are basically any two ways that a data object can be made available as an operand for an operation.

(i) Direct Transmission : A data object computed at one point as the result of an operation may be directly transmitted to another operation as an operand as for example : the result of the multiplication $2 \times Z$ is transmitted directly to the addition operation as an operand in the statement $X := Y + 2 \times Z$. In this case, the data object is allocated storage temporarily during its life time and may never be given a name.

Referring Through a Named Data Object : A data object may be given a name, when it is created, and the name may then be used to designate it as an operand of an operation. Alternatively the data object may be made a component of another data object that has a name so that the name of a larger data object may be used together with a selection operation to designate the data object as an operand.

Direct transmission is used for data control within expressions, but most data control outside of expressions involves the use of names and the referencing of names. The problem of the meaning of names forms the central concerns in data control.

Programs Functions that May be Named : Each language differs, but some general categories seen in many languages are :

- (i) Variable names
- (ii) Formal parameter name
- (iii) Subprogram names
- (iv) Name for defined types
- (v) Names for defined constants
- (vi) Statement labels
- (vii) Exception names
- (viii) Name for primitive operations e.g., +, *, SQRT
- (ix) Name for literal constants e.g., 17, 3, 25.

Q. 7. (b) Write and discuss sequencing out arithmetic expressions.

Ans. Consider the formula for computing roots of the quadratic equation :

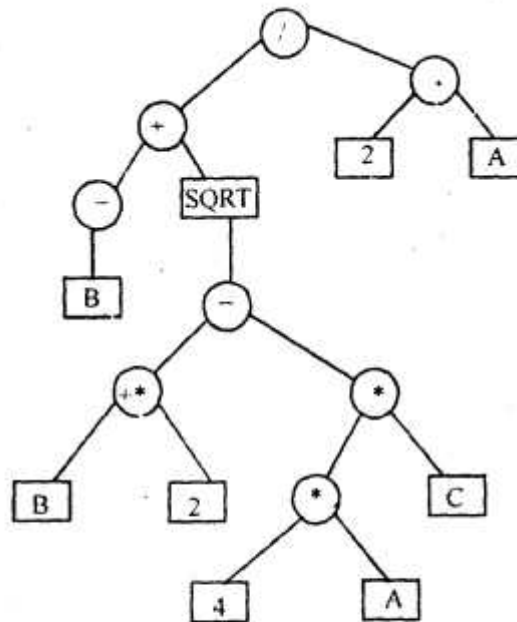
$$r_{out} = \frac{-\beta \pm \sqrt{\beta^2 - 4 \times A \times C}}{2 \times A}$$

This apparently simple formula actually involves at least 15 separate operations. Coded in a typical assembly or machine language it would require at least 15 instructions one probably for more. Moreover, the programmer would have to provide storage for and keep track of each of the several intermediate results generated and would also have to worry about optimization. Can the two references to the value of B and of A be combined, in what order should the operation be performed to minimize temporary storage and make best use of the hardware, and so on? In a high level language such as FORTRAN, however, the formula for one of the roots can be coded as a single expression almost directly :

$$\text{Root} = (-B + \text{SQRT}(B^2 - 4 * A * C)) / (2 * A)$$

The notation is compact and natural, and the language processor rather than the programmer concerns itself with temporary storage and optimization. It seems fair to say that the availability of expressions in high level languages is one of their major advantages over machine and assembly language. Expressions are a powerful and natural device for representing sequences of operations, yet they raise new problems? Take the FORTRAN expression for the quadratic formula.

The representation of quadratic formula are given as follow :



[Tree representation of quadratic form]

Q. 7. (c) Define static and dynamic scope.

Ans.

```

Procedure Big is
X : Integer;
procedure sub 1 is
begin -- of sub 1
..... X .....
end; -- of sub 1
procedure sub 2 is
X : Integer;
begin -- of sub 2
...
end; -- of sub 2
begin -- of Big
...
end; -- of Big.
    
```

The scope of a variable is the range of statements in which the variable is visible. A variable is visible in a statement if it can be referenced in that statement. There are of two types of scope-static & dynamic scope.

Static Scope : The method of binding names to non-local variables, called static scoping, static scoping

is so named because the scope of a variable can be statically determined; that is, prior to execution. There are two categories of static-scoped languages : those in which sub-programs can be nested, which creates nested static scopes, and those in which subprograms cannot be nested. In this later, category, static scopes are also created by subprograms but nested scopes are created only by nested class definitions and blocks.

Ada, Java-Script and PHP allow nested sub-programs, but the C-based language do not. A static-scoped languages finds a reference to a variable, the attributes of the variable can be determined by finding the statement in which it is declared.

Dynamic Scope : The scope of variables in APL, SNOBOL4 and the early versions of LISP is dynamic. Perl and COMMON LISP also allow variables to be declared to have dynamic scope, although these languages also use static scoping. Dynamic scoping is based on the calling sequence of sub-programs, not on their spatial relationship to each other. Thus, the scope can be determined only at run time.

Consider the procedure : Assume that dynamic scoping rules apply to non-local references. The meaning of the identifier X referenced in sub I in dynamic—it cannot be determined at compile time. It may reference the variable from either declaration of X, depending on the calling sequence.

Q. 8. (a) What are major run-time elements requiring storage?

Ans. The programmer tends to view storage management largely in terms of storage of data and translated programs. However run-time storage management encompasses many other areas. Some such as return points for subprograms have been touched on, other have not yet been mentioned explicitly.

The major program and data elements requiring storage during program execution are given as follows :

(i) Code Segments for Translated User Programs : A major block of storage in any system must be allocated to store the code segments representing the translated form of user programs.

(ii) System Run-Time Programs : Another substantial block of storage during execution must be allocated to system programs that support the execution of the user programs. These may range from simple library routines, such as sine, cosine or print-string functions, to software interpreters or translators present during execution.

(iii) User-Defined Data Structure and Constants : Space for user data must be allocated for all data structure declared in or created by user programs including constants.

(iv) Subprogram Return Points : Subprograms have the property that they may be involved from different-points in a program. Therefore, internally generated sequence control information, such as subprogram return points, coroutine resume points or event notices for scheduled subprograms, must be allocated storage.

(v) Referencing Environments : Storage of referencing environments (identifier, associations) during execution may require substantial space.

(vi) Input Output Buffers : Ordinarily input and output operations work through buffers, which serve as temporary storage areas where data are stored between the time of the actual physical transfer of the data to or from external storage and the program-initiated input and output operations. Often hundreds of memory locations must be reserved for these buffers during execution.

(vii) Miscellaneous System Data : In almost every language implementation, storage is required for various system data : tables, status information for input-output and various miscellaneous pieces of state information.

Q. 8. (b) Explain the concept of static-storage management

Ans. The simplest form of allocation is static allocation, allocation during translation that remains fixed throughout execution. Ordinarily storage for the code segments of user and system programs is allocated statically, as is storage for I/O buffers and various miscellaneous system data. Static allocation requires no run

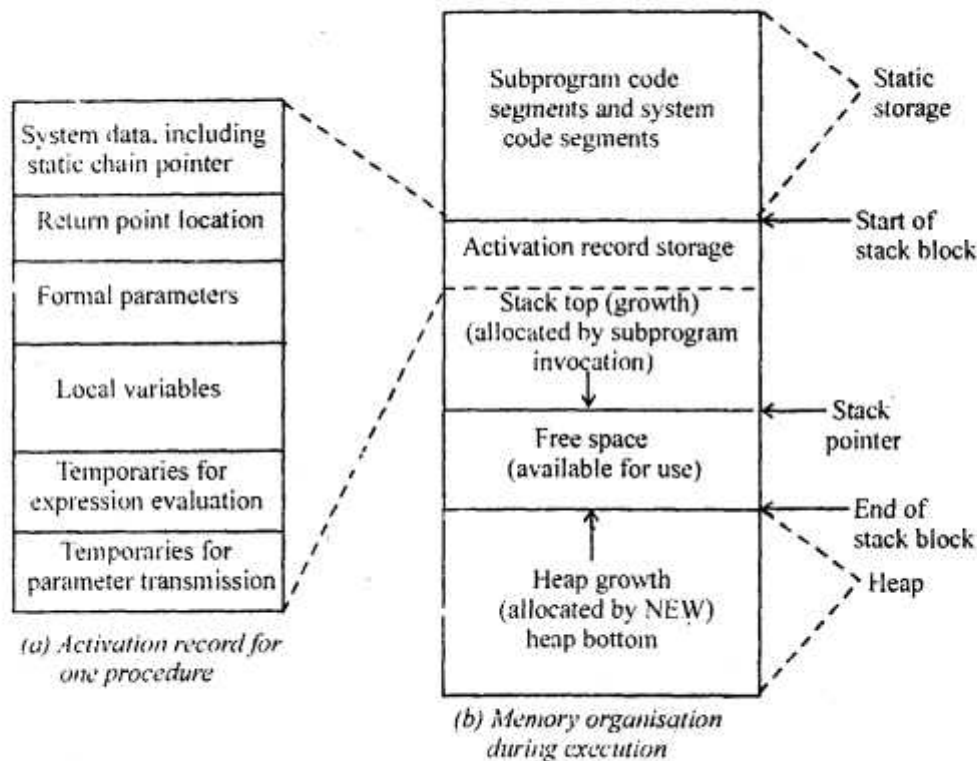
time storage management software and of course, there is no concern for recovery and reuse.

In the usual FORTRAN implementation, all storage is allocated statically. Each subprogram is compiled separately, with the compiler setting up the code segment (including an activation record) containing the compiled program, its data areas, temporaries, return point location and miscellaneous items of system data. The loader allocates space in memory for these compiled blocks at load time, as well as space for system run-time routines. During program execution no storage management need take place.

Static storage allocation is efficient because no time or space is expended for storage management during execution. The translator can generate the direct 1-value addresses for all data items. However, it is incompatible with recursive subprogram calls, with data structure whose size is dependent on computed or input data, and with many other desirable language feature. For many programs, static allocation is quite satisfactory. Two of the most widely used programming languages, FORTRAN and COBOL, are designed for static storage allocation and languages like C, which have dynamic storage, also permit static data to be created for efficient execution.

Q. 9. (a) Define Stack-based storage management?

Ans. The simplest run time storage management technique is the static. Free storage at the start of execution is set up as a sequential block in memory. As storage is allocated, it is taken from sequential locations in this stack block beginning at one end. Storage must be freed in the reverse order of allocation so that a block of storage being freed is always at the top of the stack.



A single stack pointer is all that is needed to control storage management. The stack pointer always points to the top of the stack, the next available word of free storage in the stack block. All storage in use lies in the

stack below the location pointed to by the stack pointer. All free storage lies above the pointer. When a block of K locations is freed, the pointer is moved back K locations. Compaction occurs automatically as part of freeing storage. Freeing a block of storage automatically recovers the freed storage and makes it available for reuse.

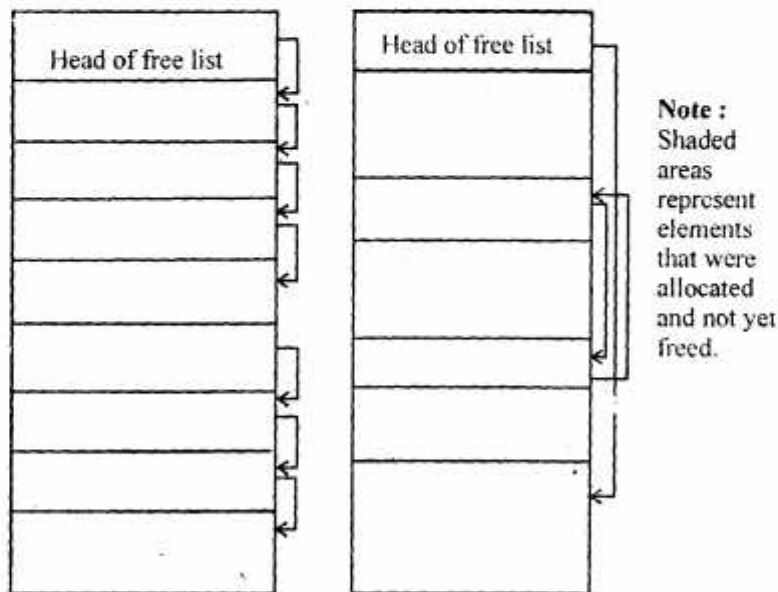
Most Pascal implementations are built around a single central stack of activation records for subprograms together with a statically allocated area containing system program and subprogram code segments. The structure of a typical activation record for a Pascal subprogram is shown in figure (a) :

The activation record contains all the variables items of information associated with a given subprogram activation. Figure (b) shows a typical memory organisation during Pascal execution.

Q. 9. (b) Explain heap storage management concepts in detail.

Ans. A heap is a block of storage within which pieces are allocated and freed in some relatively unstructured manner. Here the problems of storage allocation, recovery, compaction and reuse may be severe. There is no single heap storage management technique, but rather a collection of techniques for handling various aspects of managing this memory.

The need for heap storage arise when a language permits storage to be allocated and freed at arbitrary points during program execution, as when a language allows creation, destruction or extension of programmer data structures at arbitrary program points.



(a) Initial free-space list

(b) Free space after execution

Assume that each fixed-size element that is allocated from the heap and later recovered occupies N words of memory. Typically N might be 1 or 2. Assuming the heap occupies a contiguous block of memory, we conceptually divide the heap block into a sequence of K elements, each N words long, such that $K \times N$ is the size of heap. Whenever an element is needed, one of these is allocated from the heap. Whenever an element is freed, it must be one of these original heap elements.

Figure (a) illustrates such an initial free space list as well as the list after allocation and freeing of a number of elements.

Q. 9. (c) What is the difference between procedural languages and non-procedural languages?

Ans. Procedural Language : Procedural programming can sometimes be used as a synonym for imperative programming (specifying the steps the programs must take to reach the desired state). Programmers writing in such language must develop a proper order of actions in order to solve the problem, based on a knowledge of data processing and programming.

Non-Procedural Languages : A computer language that does not require writing traditional programming logic. Also known as a "declarative language, users concentrate on defining the input and output rather than the program steps required in a procedural language.

The following data base example show procedural and non-procedural ways a list of file. Procedural and non-procedural languages are also considered third and fourth generation languages :

Procedural (3GL)

```
USE FILE X
DO WHILE .NOT.EOF
    ?NAME, AMOUN DUE
    SKIP
END DO
```

Non-Procedural (4GL)

```
USE FILE X
LIST NAME, AMOUN DUE.
```

Example :

Procedural Languages : C++, COBOL or Visual Basic etc.

Non-Procedural Language : Query languages, report writers, interactive database programs, spreadsheets etc.