SYLLABUS

Principles of Programming Languages (CS515PE)

Unit - I

Preliminary Concepts : Reasons for studying concepts of programming languages, programming domains, language evaluation criteria, influences on language design, language categories, language design trade-offs, implementation methods, programming environments.

Syntax and Semantics : General problem of describing syntax and semantics, formal methods of describing syntax, attribute grammars, describing the meanings of programs. **(Chapter - 1)**

Unit - II

Names, Bindings, and Scopes : Introduction, names, variables, concept of binding, scope, scope and lifetime, referencing environments, named constants.

Data types : Introduction, primitive data types, character string types, user defined ordinal types, array, associative arrays, record, union, tuple types, list types, pointer and reference types, type checking, strong typing, type equivalence.

Expressions and Statements : Arithmetic expressions, overloaded operators, type conversions, relational and boolean expressions, short-circuit evaluation, assignment statements, mixed-mode assignment.

Control Structures : Introduction, selection statements, iterative statements, unconditional branching, guarded commands. **(Chapter - 2)**

Unit - III

Subprograms and Blocks : Fundamentals of subprograms, design issues for subprograms, local referencing environments, parameter passing methods, parameters that are subprograms, calling subprograms indirectly, overloaded subprograms, generic subprograms, design issues for functions, user defined overloaded operators, closures, co routines.

Implementing subprograms : General semantics of calls and returns, implementing simple subprograms, implementing subprograms with stack-dynamic local variables, nested subprograms, blocks, implementing dynamic scoping.

Abstract Data types : The concept of abstraction, introductions to data abstraction, design issues, language examples, parameterized ADT, encapsulation constructs, naming encapsulations. **(Chapter - 3)**

Unit - IV

Concurrency : Introduction, introduction to subprogram level concurrency, semaphores, monitors, message passing, Java threads, concurrency in function languages, statement level concurrency.

Exception Handling and Event Handling : Introduction, exception handling in Ada, C++, Java, introduction to event handling, event handling with Java and C#. **(Chapter - 4)**

Unit - V

Functional Programming Languages : Introduction, mathematical functions, fundamentals of functional programming language, LISP, support for functional programming in primarily imperative languages, comparison of functional and imperative languages.

Logic Programming Language : Introduction, an overview of logic programming, basic elements of prolog, applications of logic programming.

Scripting Language : Pragmatics, Key Concepts, Case Study : Python - Values and Types, Variables, Storage and Control, Bindings and Scope, Procedural Abstraction, Data Abstraction, Separate Compilation, Module Library. **(Chapter - 5)**

UNIT - I

1 Preliminary Concepts, Syntax and Semantics

Part I : Preliminary Concepts

1.1 Reasons for Studying Concepts of Programming Languages

Q.1 What is need for study of programming languages ? Explain.

☞ [JNTU : Part B, May-18, Marks 5]

Ans. : Following are the reasons for studying the programming language concepts -

1) Increased capacity to express ideas :

- The study of different programming languages makes the programmer with variety of programming constructs.
- In this study, programmer can understand various data structures, control structures abstractions and their ability.
- Then they can use suitable programming constructs to express the ideas they want to implement.

2) Improved background for choosing appropriate language :

- By the knowledge of various programming languages, one can be aware of its variety of features.
- Hence while developing a software application programmer can make appropriate selection of the language whose features are most applicable.
- For example – if application requirement is use of GUI then the programmer will choose Visual basic as a programming language.

3) Increased ability to learn a new language :

- Due to the knowledge of programming constructs and implementation techniques, programmer can learn new language easily.

4) Better understanding of significance of implementation :

- By learning and understanding different languages, programmer can know the reasons behind the language design.
- This knowledge leads to use the language more intelligently.

5) Better use of languages that are already known :

- By studying the concepts of programming languages, programmers can learn about previously unknown and unused parts of the languages they already use and begin to use those features.

6) Overall advancement of computing :

- By understanding variety of features of programming languages such as arrays, dynamic arrays, strings, dynamic memory allocation, generic programming - the programmer can use these features for advance level implementation and computing.

1.2 Programming Domain

Q.2 Specify different areas of computer applications and their associated languages.

Ans. : 1) Scientific applications :

- The scientific applications require large numbers of floating point computations.
- It typically makes use of arrays.
- The programming language used is - Fortran.

2) Business applications :

- The business applications are characterized by facilities of producing variety of reports.
- It requires use decimal numbers and characters data.
- It also needs an ability to specify decimal arithmetic operations.
- The programming language preferred for building the business applications is - COBOL.

3) Artificial intelligence :

- The artificial intelligence is an area in which the computations are symbolic rather than numeric. Symbolic computation means that symbols, consisting of names rather than numbers.
- Symbolic computation is more conveniently done with linked lists of data rather than arrays.
- This kind of programming sometimes requires more flexibility than other programming domains
- The programming language used in this area of applications is - LISP.

4) Systems programming :

- System programming is a system software which consists of collection of operating system and programming support tools.
- The system software is used continuously, hence it needs to be efficient.
- It should also have a support for interfacing with external devices to be written. (For example - some support programs for plug and play devices).
- The general programming languages such as C, C++ are usually preferred for system programming.

5) Web software :

- The World Wide Web software is used to represent some dynamic web contents, representation of text, audio or video data, and contents containing some computations and so on.
- It need a support for variety of languages from markup languages (For example - HTML), scripting (For example - PHP), to general-purpose (For example - Java).

1.3 Language Evaluation Criteria

Q.3 Explain the criteria of success for a good programming language.

☞ [JNTU : Part B, March-17, Marks 5]

Ans. : There are four commonly used evaluation criteria

1) Evaluation Criteria : Readability : It is an ability of the language because of which the program can be easily read and understood. This criterion is directly related to following characteristics of the language -

i) Simplicity :

a) The overall **simplicity** of the program which can be achieved by having **manageable set of features and constructs.**

b) Feature multiplicity : That means existence of multiple ways of doing the same operation.

c) Operator overloading : If operator overloading is used appropriately then the program becomes readable otherwise it becomes less readable.

ii) Orthogonality : A set of features/constructs is said to be orthogonal if some features can be combined to build the control and data structures of the language.

iii) Data types : The total number of data types used in the programming language.

iv) Syntax design : Three types of syntax affect readability : identifier forms, special words and form and meaning.

2) Writability : It is an ability of language which makes it easy to create the program. It is directly related to following characteristics -

i) Simplicity and Orthogonality : That means there should be small number of primitive constructs(simplicity) and consistent set of rules for combining them (orthogonality) to make the language writable.

ii) Support for Abstraction : This is a characteristic of language to define and use complicated operations in such a way that the implementation details can be ignored.

iii) **Expressivity :** This is a characteristic in which programmer is allowed to use powerful operators to express the idea in less number of lines of codes.

3) **Reliability :** It is a conformance to specification. That means this is a criteria which allows to perform specification under all conditions. Following are the characteristics that are directly related to reliability of the language.

i) **Type checking :** The type errors need to tested.

ii) **Aliasing :** This refers to having two or more names referring to same memory cell. This is something that affects the reliability of the program.

iii) **Readability and writability :** The program becomes more reliable if both the readability and writability is achieved.

4) **Cost :** The total cost of software system. It depends upon following factors

i) Training to the programmer

ii) Software creation

iii) Compilation an execution

iv) Poor reliability

v) Maintenance

5) **Some other criteria :**

i) **Portability :** This is a criterion that allows to move program from one implementation to other.

ii) **Generality :** It is a applicability to wide range of applications.

iii) **Well definiteness :** The completeness of a language.

Q.4 Describe in your own words, the concept of orthogonality in programming language design.

☞ **[JNTU : Part B, March-17, Marks 5]**

Ans. : Refer Q.3

Q.5 Describe the important factors influencing the writability of a language.

☞ **[JNTU : Part B, Dec.-17, Marks 5]**

Ans. : Refer Q.3

1.4 Influences on Language Design

Q.6 What are the different factors that influence the evolution of programming languages ?

☞ **[JNTU : Part B, Marks 5]**

Ans. : Following are the factors that influence the evolution of programming languages -

i) Computer architecture :

- The most commonly used computer architecture is Von Neumann.
- According to this architecture data and programs are store in memory.
- Memory is separate from CPU.
- Instructions and data move from memory to CPU.
- There is a strong use of imperative language.

ii) Programming methodology :

- In 1950s and early 1960s simple applications with limited machine efficiency were built.
- In later 1960s the programs with better readability and better control structures were developed.
- Then the structured programming approach is being adopted.
- In 1980s there is a strong use of object oriented programming methodology.

1.5 Language Categories

Q.7 Describe programming paradigms.

☞ **[JNTU : Part B, May-18, Marks 5]**

Ans. : Definition : Programming paradigm can be defined as a method of problem solving and an approach to programming language design.

Various types of programming paradigm are -

1. Imperative or procedural programming :

- The imperative programming is also called as procedural programming language.
- A program consists of sequence of statements. After execution of each statement the values are stored in the memory.
- The central features of this language are variables, assignment statements, and iterations.
- Examples of imperative programming are - C, Pascal, Ada, Fortran and so on.

2. Object oriented programming :

- In this language everything is modeled as object. Hence is the name.
- This language has a modular programming approach in which data and functions are bound in one entity called class.
- This programming paradigm has gained a great popularity in recent years because of its characteristic features such as data abstraction, encapsulation, inheritance, and polymorphism and so on.
- Examples of object oriented programming languages are - Smalltalk, C++, Java,

3. Functional programming :

- In this paradigm the computations are expressed in the form of functions.
- Functional programming paradigms treat values as single entities. Unlike variables, values are never modified. Instead, values are transformed into new values.
- Computations of functional languages are performed largely through applying functions to values, i.e., (+ 10 20). This expression is interpreted as 10 + 20. The result 30 will be returned.
- For building more complex functions the previously developed functions are used. Thus program development proceeds by developing simple function development to complex function development.
- Examples of function programming are LISP, Haskell, ML.

4. Logic programming :

- In this paradigm we express computation in terms of mathematical logic only. It is also called as rule based programming approach.
- Rules are specified in no special order.
- It support for declarative programming approach. In this approach how the computations take place is explained.
- The logic paradigm focuses on predicate logic, in which the basic concept is a relation.
- Example of Logic programming is Prolog.

Q.8 List out language categories.

☞ [JNTU : Part A, Dec.-16, Marks 3]

Ans. : Language categories are :

1) Procedural or imperative programming
2) Object oriented programming
3) Logic programming
4) Functional programming.

Q.9 Explain the merits and demerits of imperative programming.

Ans. : Merits :

1. Simple to implement.
2. These languages have low memory utilization.

Demerits :

1. Large complex problems can not be implemented using this category of language.
2. Parallel programming is not possible in this language.
3. This language is less productive and at low level compared to other programming languages.

Q.10 Explain the merits and demerits of functional programming.

Ans. : Merits :

1. Due to use of functions the programs are easy to understand.
2. The functions are reusable
3. It is possible to develop and maintain large programs consisting of large number of functions.

Demerits :

1. Functional programming is less efficient than the other languages.
2. They consume large amount of time and memory for execution
3. Purely functional programming is not a good option for commercial software development.

Q.11 Explain the merits and demerits of logic programming.

Ans. : Merits :

1. This programming paradigm is reliable.
2. The program can be quickly developed using this approach as it makes use of true/false statements rather than objects.
3. It is best suitable for the problems in which knowledge base can be established.

Demerits :

1. Execution of the program is very slow
2. True/false statements can not solve most of the problems
3. It can solve only limited set of problems efficiently.

Q.12 Give the relative advantages of object oriented programming paradigm.

☞ [JNTU : Part A, March-17, Marks 2]

Ans. : 1) Re-usability : The class feature allows the code to be reused instead of writing it over and over again.

2) Data Redundancy : If a user wants a similar functionality in multiple classes, he/she can go ahead by writing common class definitions for the similar functionalities and inherit them.

3) Security : With the use of data hiding and abstraction mechanism, security for sensitive information can be maintained.

4) Code Maintenance : It is easy to maintain and modify existing code as new objects can be created with small differences to existing ones.

5) Objects can be created on real world entities.

Q.13 What are the fundamental features of imperative languages ?

☞ [JNTU : Part A, March-17, Marks 3]

Ans. : Refer Q.7.

Q.14 The levels of acceptance of any language depend upon the language description. Comment.

☞ [JNTU : Part B, Nov.-15, Marks 5]

Ans. : Refer Q.3.

1.6 Language Design Trade-offs

Q. 15 What are design trade-offs ? Explain.

☞ [JNTU : Part A, Marks 3]

Ans. : There are certain design criterion that are conflicting each other. That means increase of one generally decreases other. These are -

1) Reliability Vs. Cost of Execution : If program is reliable then it is probably slower to execute.

2) Readability Vs. Writability : If program is readable then it is probably not writable or vice versa.

3) Flexibility Vs. Safety : If program is flexible then probably it is not safe or vice versa

1.7 Implementing Methods

Q.16 What are the three general methods of implementing a programming language ?

☞ [JNTU : Part B, Nov.-15, Marks 5]

Ans. : The three methods of language implementation are -

1) Compilation : This is a process in which the high level programs are translated into machine language.

2) Pure interpretation : In this process, programs are interpreted line by line to target program.

3) Hybrid implementation system : In this system the programs are translated using both interpreter and compiler.

Q.17 List the principal phases of compilation.

☞ [JNTU : Part A, Dec.-17, Marks 2]

Ans. : 1) Lexical analysis

2) Syntax analysis

3) Intermediate code generation and semantic analysis

4) Code generation

5) Code optimization(optional phase)

Q.18 Explain in detail different stages in language translation.

Ans. : • The translation is divided into two major parts - **Analysis and synthesis**. The task of compilation is carried out in different phases. Refer Fig. Q. 18.1 These phases actually represent basic structure of compiler.

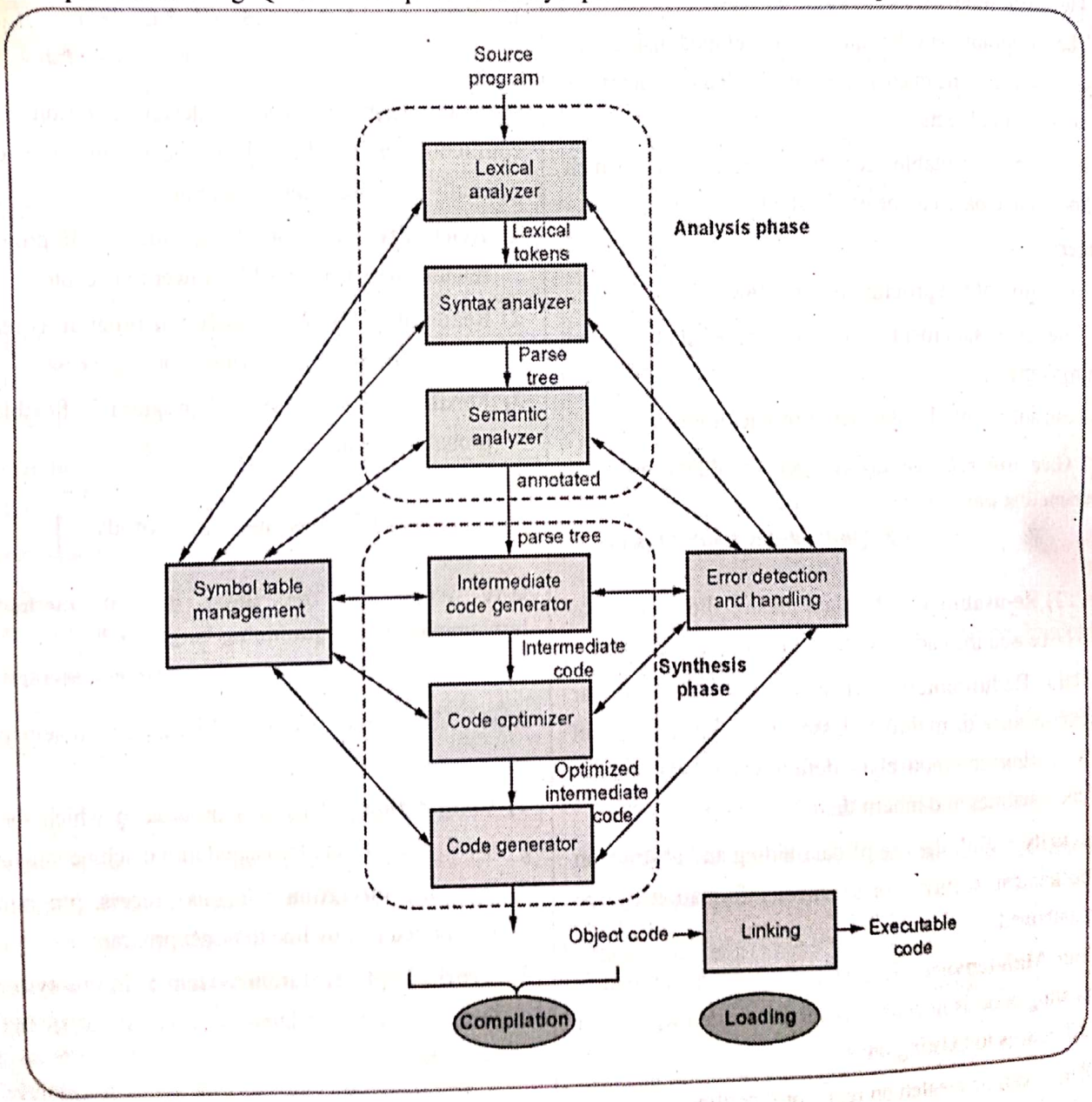


Fig. Q. 18.1

1. Lexical analysis :

- This is an initial phase of compilation in which each statement in the source program is scanned or read line by line and then it is broken into stream of strings called **tokens**.
- The lexical analysis phase is also called as scanning because during this phase entire source program is read or scanned.
- The lexical analyser identifies the type of each lexeme and store its information in the symbol table.

2. Syntax analysis :

- The syntax analysis is a phase which takes the stream of tokens and build a hierarchical structure for checking the syntax of the source program.
- This is the second phase in the process of compilation.
- It is also called as **parsing** as the parse tree is constructed in this phase.

3. Semantic analysis :

- The semantic analysis is a phase in which the meaning of the source program is analysed.
- This is a phase in which various tasks such as symbol table maintenance, error detection and recovery, and insertion of implicit information.

4. Code optimization :

- The semantic analyser produces the code called intermediate code. For example - consider the statement

$$a = b + c - d$$

- The intermediate code can be generated as follows -

$$t1 = b + c$$
$$t2 = t1 - d$$
$$a = t2$$

- The code can be generated from above steps as follows -

1. Load register with b
2. Add c to register
3. Store register in temporary variable t1
4. Load register with t1
5. Subtract d to register
6. Store register in t2
7. Load register with t2
8. Store register in a

Note that instructions 3, 4, 6 and 7 are redundant because all data can be kept in register before storing it to variable a. This is one kind of optimization that can be induced during the translation of the program.

5. Code generation : The output code produced during this phase can be directly executable or there may be another translation step required to follow. These steps can be assembly, linking and loading.

6. Linking and Loading : This is an optional final phase. The task of loader is to perform the relocation of an object code.

- The link editor links the object modules and prepares a single module from several files of relocatable object modules to resolve the mutual references.
- These files may be library files and these library files may be referred by any program.
- **Symbol table management :** The symbol table is a data structure in which the information about identifiers, arrays names, subprogram name, formal parameters and so on is stored. The lexical analyser enters the initial information as it reads the source program the semantic analyser enters the information as it process the programming statements.
- **Error detection :** Errors can be detected during the lexical analysis, syntax analysis and semantic analysis phase. The semantic analyser not only detects the errors but produce appropriate error messages.

Q.19 Explain language processing using translation.

☞ **[JNTU : Part B, Nov.-15, Marks 5]**

Ans. : • The process of translation is performed in following steps -

1. The program modules are separately translated into relocatable machine code. This translator is called as **compiler**.
2. These translated modules are linked together in a relocatable unit. This task is carried out by **linker** or **linkage editor**.
3. Finally the complete program is loaded into the memory as an executable machine code. This task is carried out by **loader**.

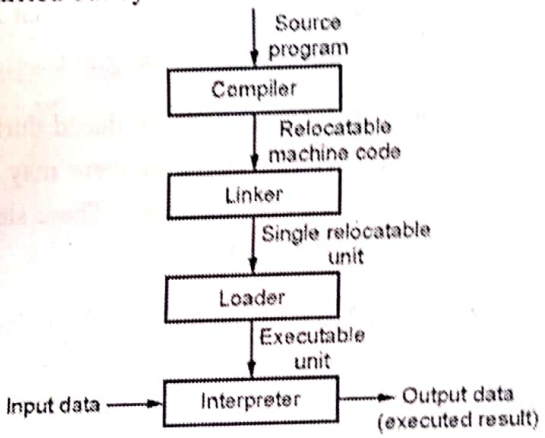


Fig. Q. 19.1 : Language processing using translation

The modern translation technique uses the combined two technique - translation and then interpretation. That means the source code is translated into **intermediate code** and then the intermediate code is interpreted.

Q.20 A programming language can be compiled or interpreted. Give relative advantages and disadvantages of compilation and interpretation.

☞ **[JNTU : Part A, May-18, Marks 3]**

OR Differentiate between compiler and interpreter.

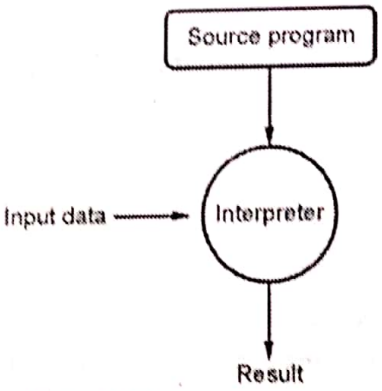
Ans. :

Sr. No.	Interpreter	Compiler
1.	**Demerit :** The source program gets interpreted every time it is to be executed, and every time the source program is analyzed. Hence interpretation is less efficient than Compiler.	**Merit :** In the process of compilation the program is analyzed only once and then the code is generated. Hence compiler is efficient than interpreter.
2.	The interpreters do not produce object code.	The compilers produce object code.
3.	**Merit :** The interpreters can be made portal because they do not produce object code.	**Demerit :** The compilers has to be present on the host machine when particular program needs to be compiled.
4.	**Merit :** Interpreters are simpler and give us improved debugging environment.	**Demerit :** The compiler is a complex program and it requires large amount of memory.

Q.21 What is pure interpretation ?

☞ **[JNTU : Part A, Marks 3]**

Ans. : In this process, programs are interpreted line by line to target program with no translation.



Advantages : Refer Q.23.

Disadvantages : Refer Q.22.

Q.22 What advantages and disadvantages of interpretation.

Ans. : Advantages :

1. Modification of user program can be easily made and implemented as execution proceeds.
2. Type of object that denotes a variable may change dynamically.
3. Debugging a program and finding errors is simplified task for a program used for interpretation.
4. The interpreter for the language makes it machine independent.

Disadvantages :

1. The execution of the program is slower.
2. Memory consumption is more.

Q.23 What are advantages of using Pure interpretation ?

Ans. : A pure interpreter reads the source text of a program, analyzes it, and executes it immediately. Following are the advantages of pure interpretation :

1) Pure interpreter are **simpler** and gives **improved debugging environment**.
2) The pure interpreter is **portable** as they do not produce the object code. Portable means the target code produced by interpreter can be executed on the desired computer architecture to get the final object code.

Q.24 Explain hybrid implementation System in detail.

Ans. : • Some languages have the implementation systems that are actually the combination of compilers and pure interpreters.

- Using compilers the high level programs are translated into intermediate language. These intermediate languages are easy to interpret. The interpreter interprets the intermediate language to form the machine code.
- Examples - Perl is implemented using with hybrid implementation system. Initially implementation of Java were all hybrid. The intermediate form of Java code is called as **byte code.** This byte code provides the facility of execution of the code on any platform. Thus portability to any machine(Platform independence) can be achieved using byte code.
- The Just-In-Time (JIT) compiler is a component of the Java Runtime Environment that improves the performance of Java applications at run time. Java programs consists of classes, which contain platform neutral bytecode that can be interpreted by a Java Virtual Machine(JVM) on many different computer architectures. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation.

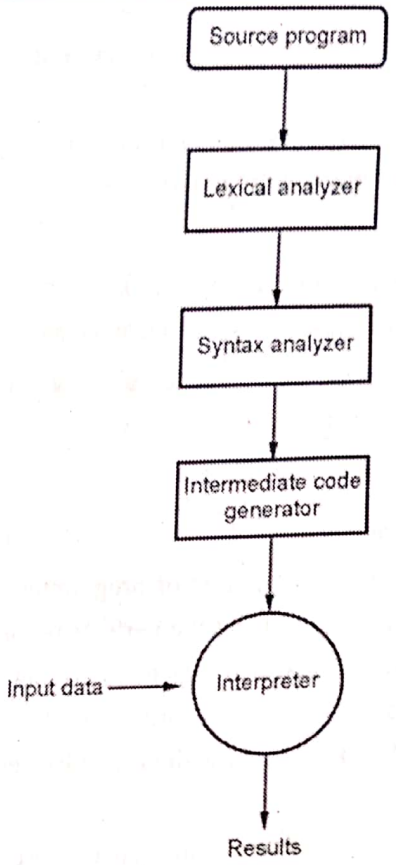


Fig. Q. 24.1 Hybrid implementation system

Q.25 Explain the concept of preprocessor in brief.

Ans. : • **Definition :** Preprocessor is a program that processes a input source program just before the it is compiled.

- Proprocessor instructions are embedded in the program.
- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included.
- For example : The C preprocessor expands #include or #define in the following manner -
  - **#include "mytestfile.h"**

    causes the preprocessor to copy the contents of mytestfile.h into the program at the position of the #include.
  - **#define SIZE 5**

    In the program we can declare an array of size 5 in following manner -

    int a[SIZE];

1.8 Programming Environments

Q.26 What is programming environment ? Explain the effect of programming environment on language design.

Ans. : Programming environment is an environment in which the programs can be created and tested.

The programming environment influence on language design in two major areas namely separate compilation and testing and debugging.

1. Separate compilation :

- There are varying number of programmers working on design, code and test the parts of programs. This require the language to structured so that individual subprograms or other parts can be separately compiled and executed without requirement of other part.
- Separate compilation is difficult because compiling one subprogram may need some information which is present in some other sub program or sometimes shared data is also used in the program due to which separate compilation is not possible. To solve this problem, some languages have adopted certain features that aid separate compilation. These features can be
  1. Use of keyword extern
  2. Use of scoping rules
  3. In object oriented programming, the feature like inheritance can be used to use the shared data.

2. Testing and Debugging :

Many languages contain some features that help in testing and debugging: For example -

1. **Breakpoint feature :** By this feature, the programmer can set the breakpoints in the program. When the breakpoint is reached during the execution of the program, the execution of the program is interrupted and control is given to the programmer. The programmer can then inspect the program for testing and debugging.
2. **Execution trace feature :** By this feature particular variable or statement can be tagged for tracing during the execution of the program.
3. **Assertions :** The assertions is a conditional expression which is inserted as a separate statement in a program. For example following is a Java Code fragment in which the assertion is used

```
System.out.print("Enter your age: ");
int age = reader.nextInt();
Assertion.assert(age<18, "You are not allowed
to vote");
```

Q.27 Explain the concept of environment framework in programming environment.

Ans. :

- Environment framework is nothing but the collection of infrastructure services that can be used for program development purpose. These services are data repository, Graphical user interface, security and communication services.
- **Borland JBuilder** is a programming environment that provides integrated compiler, editor and debugger for Java development.
- The modern programming environment makes use of **Visual Studio .NET** for developing the applications using C#, Visual Basic, .NET, Jscript, J# and so on.
- **Unix** is an old programming environment and has a strong support for various powerful tools for software production and maintenance. The UNIX GUI runs on the top of basic UNIX kernel. Typical examples of such GUI are GNOME and KDE.

1.9 Evolution of Major Programming Languages

Q.28 Explain evolution of major programming languages.

Ans. : • FORTRAN I :

- It is designed for new IBM 704 , which had index registers and floating point hardware.

- From this language onwards the idea of compilation came into existence.
- The first implemented version of Fortran has following features -
  - Names could have up to six characters.
  - Post-test counting loop such as DO was present.
  - It has formatted I/O
  - It allows user defined subprograms.
  - There were no data typing statements

- **FORTRAN II :**
  - It was distributed in 1958
  - Independent compilation was supported for FORTRAN II
- **FORTRAN IV,77,90 :**
  - It is evolved during 1960-62
  - It has explicit type declarations
  - There was logical selection statement
  - Subprogram names could be parameters
  - It was standardized as FROTRAN 66 in 1966
  - In FORTRAN 77 the character sting handling features was introduced. IF-THEN-ELSE statement were introduced.
  - Most significant changes took place in FORTRAN 90 by allowing use of modules, dynamic arrays, pointers, recursion,CASE statements, parameter type checking.
- **Functional Programming : LISP :**
  - It is a list processing language.
  - This language is used in the research of AI.
  - The syntax is based on lambda calculus.
- **ALGOL 60 :**
  - It supports the block structure.
  - It has two parameter passing methods.
  - It supports the subprogram recursion.
- **COBOL :**
  - The first macro facility in a high level language.
  - It has a support for hierarchical data structures.
  - It contains nested selection statements.
  - Long names for variables up to 30 characters were allowed with hyphens.
  - It has separate data division.
- **BASIC :**
  - It is easy to use and learn.
  - The current popular version of BASIC is visual basic.
- **PL/I :**
  - It was designed by IBM and SHARE.
  - Characterized by dynamic typing and dynamic storage allocation
  - Variables are untyped i.e. A variable acquires a type when it is assigned a value
  - Storage is allocated to a variable when it is assigned a value.
- **PASCAL :**
  - It is designed initially for teaching structured programming.
  - It was small, simple and easy to learn.
- **C :**
  - It is developed in 1972 for system programming.
  - It has powerful set of operators.
  - Though designed as a systems language, it has been used in many application areas
- **PROLOG :**
  - It is based on formal logic.
  - It is a Non-procedural language.
  - It is considered to be intelligent database system that uses an inferencing process to infer the truth of given queries.
  - It is comparatively inefficient.
- **ADA :**
  - For design of this language required huge design effort, involving hundreds of people, much money, and about eight years.
  - By allowing use of packages, ADA has a support for data abstraction.
  - It support the exception handling mechanism.
  - It allows the use of generic units and concurrency mechanism.
  - It contains more flexible libraries.

- **C++ :**
  - It is developed in Bell labs in 1980.
  - It support for both **procedural** and **object oriented fatures**.
- **JAVA :**
  - Developed by Sun Microsystems in 1990.
  - It is based on C++
  - It supports only **OOP**.
  - It contains references but no pointers.
  - It has a facility and support for applets and concurrency.
  - It has libraries for applets, GUIs, database access.
  - It is widely used for web programming.
- **Scripting Languages for WEB :**
  - Various scripting languages such as Perl, PHP, Python, JavaScript, Ruby and so on are used as a scripting languages for web programming.
- **.NET Language: C# :**
  - It is based on C++ , Java, and Delphi.
  - It includes pointers, delegates, properties, enumeration types, a limited kind of dynamic typing, and anonymous types.
  - Is evolving rapidly.
- **Markup/Programming Hybrid Languages :**
  - XSLT : eXtensible Stylesheet Language Transformation (XSTL) transforms XML documents for display.
  - Java Server Pages : a collection of technologies to support dynamic Web documents

Part II : Syntax and Semantics

1.10 General Problem Describing Syntax

Q.29 Explain the following terminologies –

i) Sentence, ii) Language, iii) Lexeme, iv) Token.

Ans. : i) Sentence : Sentence is a string of characters over some alphabet.

ii) Language : Language is a set of sentences.

iii) Lexeme : Lexeme is a lowest level syntactic unit of language. For example – index, count

iv) Token : The token is a category of lexemes. For example – identifier, keywords.

Consider following statement

a = b + 10

Lexeme	Token
a	identifier
=	assign_operator
b	identifier
+	addition_operator
10	int_literal

Q.30 Define syntax and Semantics.

☞ **[JNTU : Part A, Nov.-15, Dec.-16, Marks 2]**

Ans. : Syntax : It is the form or structure of the expressions, statements, and program units. The grammar of a language is called syntax.

Semantics : The meaning of the expressions, statements, and program units is called semantic.

Q.31 Differentiate between syntax and semantics.

☞ **[JNTU : Part B, March-16, Marks 5]**

Ans. :

Syntax	Semantics
The **form or structure** of the expressions, statements, and program units is called **syntax.**	The **meaning** of the expression, statements and programming units is called **semantics.**
It is handled at **compile time.**	It is handled at **run time.**
It is given by **context free grammar**	It is given by **syntax directed definition.**
The syntactic **errors are easy to locate.**	The semantic **errors are difficult to locate.**

Q.32 Define grammars, derivation and a parse tree.

☞ **[JNTU : Part B, Dec.-16, Marks 3]**

Ans. : Grammar : The formal language-generation mechanism that are used to describe the syntax of programming language is called grammar. There are two methods of describing the syntax - Backus-Naur Form(BNF) and Context Free Grammar(CFG).

Derivation : Refer Q.34.

Parse Tree : Refer Q.34.

Q.33 Discuss about language recognizers and language generators.

☞ **[JNTU : Part B, Dec.-16, Marks 5]**

Ans. : For defining a language formally there are two ways - i) Recognition and ii) Generation.

i) Language Recognizers :

- A recognition device reads input strings of the language and decides whether the input strings belong to the language.
- For example - consider an input set Σ = {a,b} and language L consists of only even length strings, the language recognizer will indicate whether the string 'abbb' belongs to L and 'abb' does not belong to L.

ii) Language Generators :

- A language generator is a device that can be used to generate the sentences of a language.
- One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator.

Q.34 Define Derivation and Parse Tree.

☞ **[Part A, Nov.-15, Marks 3]**

Ans. : Derivation : Derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols).

A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded. The derivation continues until the sentential form contains no non terminals.

A rightmost derivation is one in which the rightmost nonterminal in each sentential form is the one that is expanded. The derivation continues until the sentential form contains no non terminals.

For example : Consider following grammar -

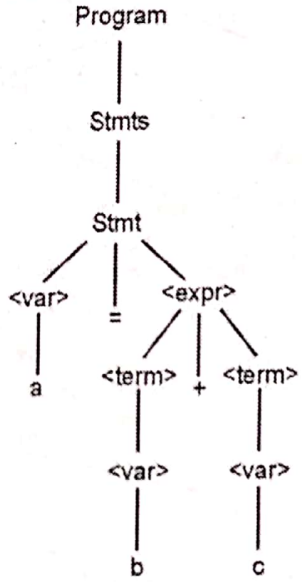
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const

The derivation of a statement a=b+c is as follows –

<program> => <stmts> => <stmt>
=> <var> = <expr>
=> a = <expr>
=> a = <term> + <term>
=> a = <var> + <term>
=> a = <var> + <var>
=> a = <b> + <term>
=> a = <b> + <c>

Parse Tree : Hierarchical structures of the language are called parse trees. For example -

Parse tree for a=b+c is as follows -



Q.35 Explain the parse tree for the sum and average program by using grammar.

☞ **[JNTU : Part B, March-16, Marks 5]**

Ans. : Step 1 : We will write the grammar rules before building the parse tree for the sum and average program.

program → stmt_list $$
stmt_list → stmt stmt_list |ε
stmt → id := expr| read id write | expr
expr → term term_tail
term_tail → add_op term term_tail |ε

term → factor factor_tail

factor tail → mult_op factor factor_tail |ε

factor → (expr)| id |number

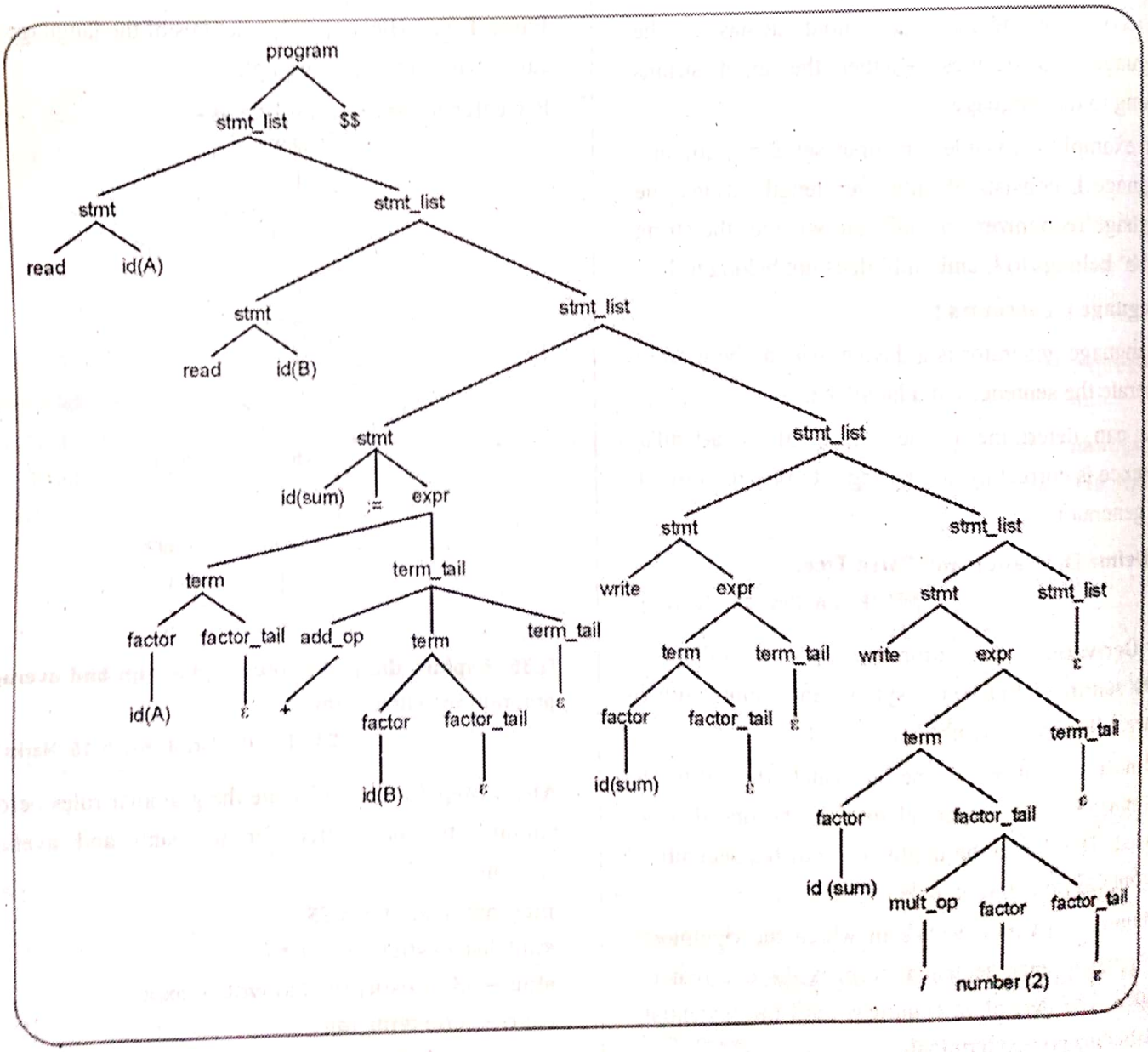
add_op → +| -

mult_op → *| /

Step 2 : Now we will use sample programming statements that compute the sum and average of two numbers.

```
read A
read B
sum := A + B
write sum      ← Calculating the sum
write sum/2 ← Calculating the average
```

Step 3 : The parse tree can be built as follows -



Q.36 Construct the parse tree for the simple statement.

A := B * (A + C).

☞ [JNTU : Part B, Dec.-17, Marks 5]

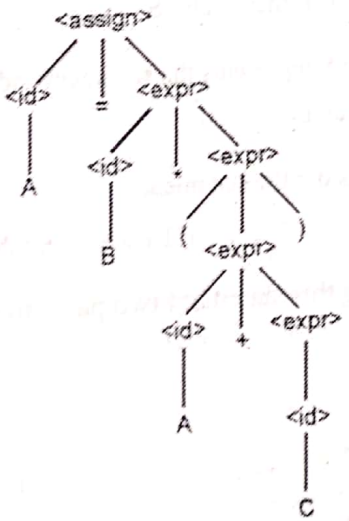
Ans. : Step 1 : For the assignment statements the grammar is as given below -

<assign> → <id> = <expr>
<id> → A|B|C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

Step 2 : The statement A = B * (A + C) can be derived using **Leftmost Derivation** as follows -

<assign> ⇒ <id> = <expr>
⇒ A = <id> * <expr>
⇒ A = B * (<expr>)
⇒ A = B * (<id>+<expr>)
⇒ A = B * (A + <expr>)
⇒ A = B * (A + <id>)
⇒ **A = B * (A + C)**

Step 3 : The parse tree is as given below -



1.11 Formal Methods of Describing Syntax

Q.37 What are the different components of the context free grammar used for programming language construction ? ☞ [JNTU : Part B, Marks 5]

Ans. : Context free grammar or simply grammar is a collection of four things -

1. The set of tokens or **terminals.**
2. The set of **non-terminals** which are actually variables representing constructs in a program.
3. The set of rules called **productions.** Each production has non-terminal at its left side, followed by the symbol := and then followed by set of terminals and non terminals as its right side.
4. The non terminal chosen as a **starting nonterminal** represents the main construct of the language.

For example - Refer Q. 38.

Q.38 Explain how to represent the arithmetic expression using context free Grammar.

☞ [JNTU : Part B, Marks 5]

Ans. : For representing the arithmetic expressions the grammar can be created. While deriving the grammar for expression the associativity and precedence is taken into consideration. The grammar for expression can be given as follows -

E::= E+T|E-T|T
T::= T*F|T/F|F
F::= (E)|id

The parse tree for id1+id2*id3-id4 is as follows –

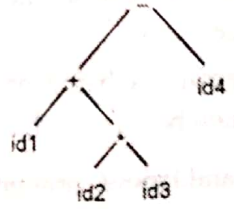


Fig. Q. 38.1 Parse tree

Q.39 Explain the Backus Naur Form.

☞ [JNTU : Part B, Marks 5]

Ans. : • Backus Naur form is a representation of context free grammar in which particular notations are used.

The non terminals in BNF are enclosed within special symbols < and >

- The empty string is written as <empty>
- The terminals appear as it is. Sometimes they can be denotes with quotes.
- The **productions** can be denoted using the symbol ::= which means "can be". The symbol "|" means OR can be used to denote the alternative definitions at the right hand side of the production.
- **Example -**

Consider following BNF rules

```
<stmt> ::=<type><list>;
<type> ::= int|float
          <list>  ::= <list>,id
          <list>  ::= id
```

Here *stmt, type* and *list* are **nonterminal** symbols. The *int, float, id* and ; are **terminal** symbols. The <stmt> is a **starting nonterminal.**

The alternatives are separated by |

Q.40 What are three reasons why syntax analyzers are based on grammars ?

☞ [JNTU : Part A, Marks 3]

Ans. : Three reasons are -

1) Using BNF (this is one form of grammar) descriptions of the syntax of programs are clear and concise.
2) BNF rules can be used as the direct basis for the syntax analyzer.
3) The implementations based on BNF are relatively easy to maintain because of their modularity.

Q.41 What is ambiguous grammar ? Explain with suitable example.

☞ [JNTU : Part B, Marks 5]

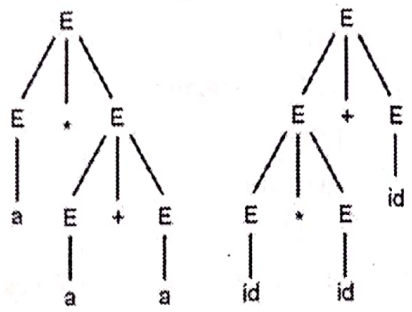
Ans. : If the grammar generates more than one parse trees for representing the same string then the grammar is said to be **static ambiguous** or **ambiguous.**

For example -

Consider the following ambiguous grammar -

E ::= E * E | E + E | a

The string **a*a+a** can be represented by following parse trees



Parse tree 1 Parse tree 2

Fig. Q. 41.1

Q.42 Explain the dangling else ambiguity.

☞ [JNTU : Part B, Marks 5]

Ans. : The dangling else problem is a famous static ambiguous or ambiguous problem. The production rules for this type of problem are as follows -

S ::= if E then S

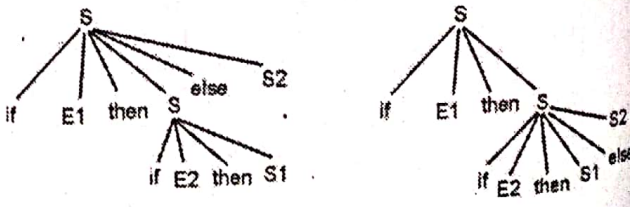
S ::= if E then S else S

Here S represents the statement and E represents the expressions.

Consider the statement

if E1 then if E2 then S1 else S2

For deriving this statement two parse trees can be drawn as follows –



Parse tree 1 Parse tree 2

Fig. Q. 42.1 Dangling - Else

Q.43 Describe the lists using BNF notation.

Ans. : Syntactic lists are described using recursion

<id_list> → identifier

|identifier, > <id_list>

The above rule is recursive because the LHS symbol appears at the RHS.

Q.44 What is EBNF ? Explain in detail.

☞ [JNTU : Part B, Marks 5]

Ans. :

- The EBNF stands for Extended BNF.
- The extensions do not enhance the descriptive power of BNF; they only increase its readability and writability.
- There are three extensions are commonly used for denoting the EBNF.
  - **The optional parts are placed in brackets [ ]**

For example -

BNF	EBNF
<if_stmt> → if <expr><statement> |if <expr><statement> else <statement>	<if_stmt> → if <expr><statement> [else <statement>]

  - **Put the repetitions in in braces ({})**

For example -

BNF	EBNF
<id_list> → identifier |identifier, > <id_list>	<id_list> → <identifier> {, <identifier>}

  - **Put multiple-choice options of RHSs in parentheses and separate them with vertical bars**

For example -

BNF	EBNF
<term> → <term> * <factor> |<term> / <factor> |<term> % <factor>	<term> → <term> (* | / | %) <factor>

Q.45 Explain with an example how operator associativity can be incorporated in grammars.

[Part B, Nov.-15, Marks 5]

Ans. : • The order of evaluation of expression is determined by precedence and associativity.

- When expression contains more than one operator, the order in which operators are evaluated depends upon their precedence levels. A higher precedence operator is evaluated before a lower precedence level operator. For example * has higher precedence than + operator.
- If the precedence levels of operators are the same, then the order of evaluation depends on associativity.
- **Left associative :** An operator is said to be left associative if subexpression containing multiple occurrences of the operator are grouped from left to right. The arithmetic operators +,-,* and / are left associative. For example - Consider the statement A=B+C+D can be represented by following parse tree

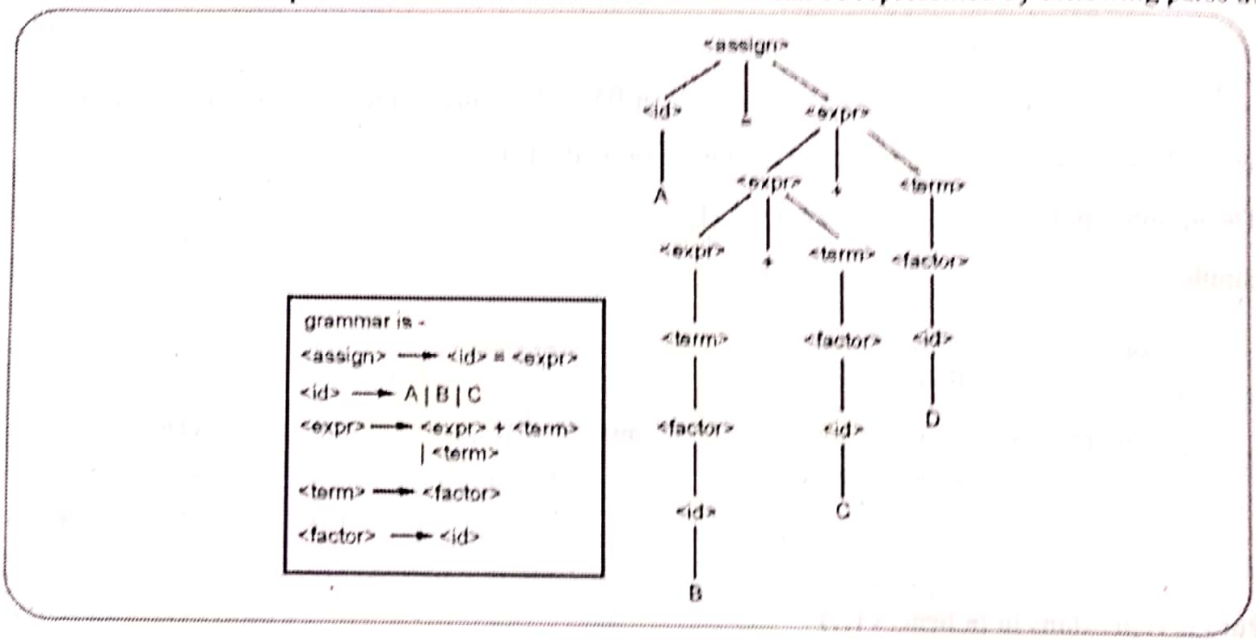


Fig. Q. 45.1 Parse Tree

- **Right associative :** An operator is said to be right associative if subexpression containing multiple occurrences of the operator are grouped from right to left. The exponentiation operator is right associative. For example - 2^2^3 = 2^(2^3) = 2^8 = 256.

1.12 Attribute Grammar

Q.46 Explain the attribute grammar and also write the attribute grammar for simple assignment statements.

[JNTU : Part B, March-16, Dec.-17, Marks 5]

Ans. : • Attribute grammar context free grammar that carry some semantic information on the parse tree node by means of attributes. Formal definition of attribute grammar is

- Associated with each grammar symbol X is a set of attributes A(X).
  - The set A(X) consists of two disjoint set S(X) and I(X), call synthesized and inherited attributes.
  - Synthesized attributes are used to pass semantic information up a parse tree, while inherited attributes pass semantic information down and across tree.
- Let X0 → X1 ... Xn be a rule
  - Functions of the form S(X0) = f(A(X1), . , A(Xn)) define synthesized attributes
  - Functions of the form I(Xj) = f(A(X0), ... , A(Xn)), for i <= j <= n, define inherited attributes

- **Attribute grammar for simple assignment statements.**

Syntax Rule	Semantic Rule
<assign> → <var> = <expr>	<expr>.exp_type ← <var>.actual_type
<expr> →<var>[2]+<var>[3]	<expr>.actual_type ← if (<var>[2].actual_type = int and (<var>[2].actual_type = int) then int else real end if **Predicate :** <expr>.actual_type == <expr>.exp_type
<expr> →<var>	<expr>.actual_type ← <var>.actual_type **Predicate :** <expr>.actual_type == <expr>.exp_type
<var>→ A | B | C	<var>.actual_type ← look-up (<var>.string) The look up function will look up the variable in the symbol table and returns the variable's type.

Q.47 Distinguish between ambiguous grammar and attribute grammar with an example.

☞ **[JNTU : Part B, Dec.-17, Marks 5]**

Ans. : Refer Q.41 and Q.46.

Q.48 Define ambiguity. Write short notes on attributed grammar.

☞ **[JNTU : Part B, May-18, Marks 5]**

Ans. : Ambiguity : A grammar is ambiguous if and only if it generates a sentential form that has two or more distinct parse trees.

Attributed Grammar : Refer Q.46.

1.13 Describing the Meaning of Programs

Q.49 What are the reasons for a methodology and notation for describing the semantics ?

☞ **[JNTU : Part B, Marks 5]**

Ans. : Several needs for a methodology and notation for semantics are -

1) Programmers need to know what statements mean
2) Compiler writers must know exactly what language constructs do
3) Correctness proofs would be possible
4) Compiler generators would be possible
5) Designers could detect ambiguities and inconsistencies.

Q.50 What are three methods semantic description ?

☞ [JNTU : Part A, Marks 3]

Ans. :

1) **Operational Semantics :** It is the method of describing the meaning of a statement or program by **specifying the effects** of running it on a machine.

2) **Denotational Semantics :** It is the method of describing the meaning of the statement based on the **recursive function theory.** In this method a mathematical object is defined for each language entity.

3) **Axiomatic Semantics :** It is a method of describing the meaning of the statement based on formal logic. The main purpose of this type of semantic method is **formal program verification.**

Q.51 In what fundamental way do operational semantic and denotational differ ?

☞ [JNTU : Part B, Nov.-15, May-18, Marks 5]

Ans. :

Operational Semantic	Denotational Semantic
The state changes are **defined by coded algorithms.**	In denotational semantics, the state changes are **defined by rigorous mathematical functions.**
Operational semantics **depends on programming languages of lower levels,** not mathematics. The statements of one programming language are described in terms of the statements of a lower-level programming language.	It **requires mathematical foundation.**
Operational semantics uses the state of a machine.	Denotational semantics uses the **state of the program** to describe meaning.
There is step-by-step computational processing of program.	There is no step-by-step computational processing of program.

Example of Operational Semantic :

For example - Consider following C statement for the for loop construct.

```
for(expr1;expr2;expr3)
{
    ... body of the for loop
}
```

Operational Semantic :

```
expr1;
loop: if expr==0 goto stop
  ...
expr3;
goto loop
stop: ...
```

Example of Denotational Semantic : Refer Q.53.

Q.52 Explain the features of denotational semantics.

☞ [JNTU : Part A, Dec.-17, Marks 3]

Ans. : The Features of denotational semantics are -

1) It is based on recursive function theory
2) It is most abstract semantic description method.
3) The mathematical object for each language entity is defined.
4) In this technique a function needs to be defined that maps instances of the language entities onto instances of the corresponding mathematical objects.
5) Can be used to prove the correctness of programs.

Q.53 Explain briefly about the denotational semantics with example.

☞ [JNTU : Part B, May-18, Marks 5]

Ans. : It is the method of describing the meaning of the statement based on the **recursive function theory.** In this method a mathematical object is defined for each language entity.

For example -

Step 1 : Consider following grammar rule –

```
<binary_num> → '0'
|'1'
|<binary_num>'0'
|<binary_num>'1'
```

Step 2 : Now we will use **Map_obj** function that maps the syntactic objects. This function is defined as -

Map_obj('0') = 0

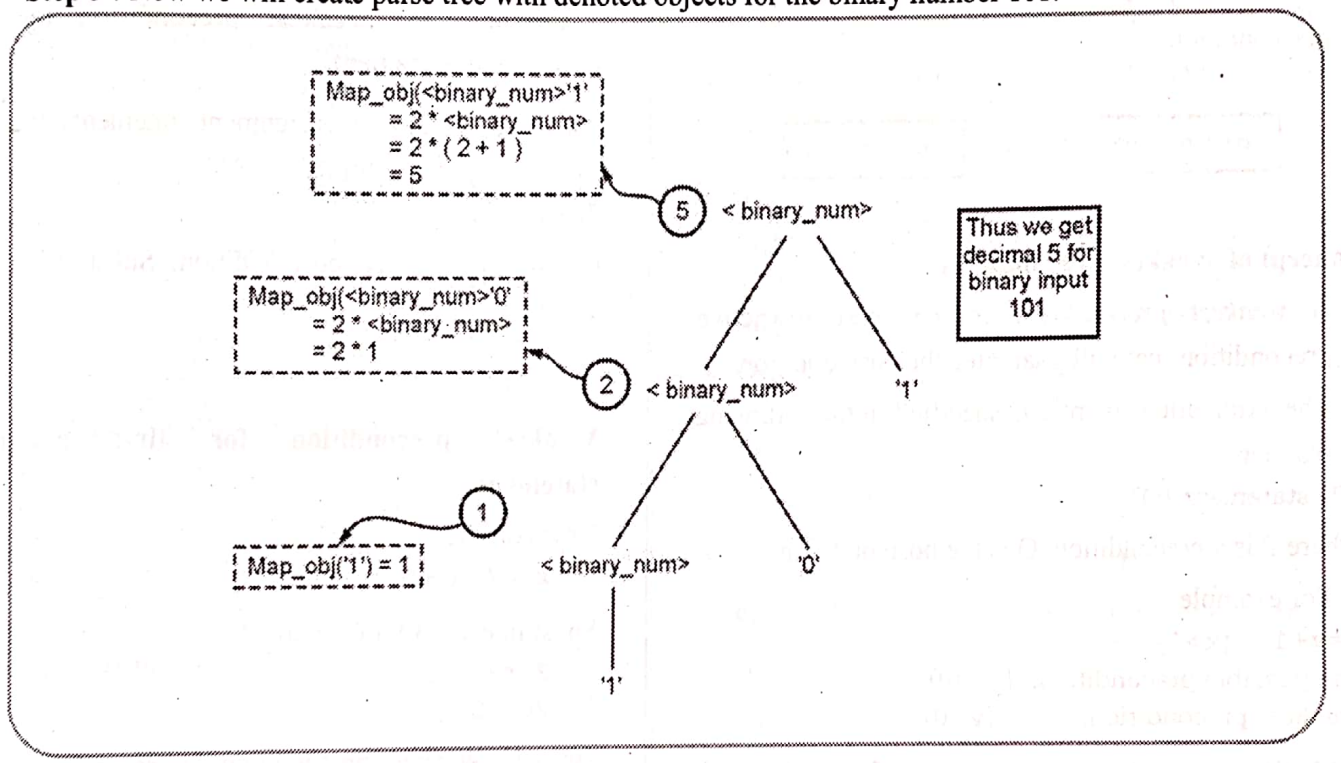
Map_obj('1') = 1

Map_obj(<binary_num>'0') = 2* Map_obj(<binary_num>)

Map_obj(<binary_num>'1') = 2* Map_obj(<binary_num>)+1

This **Map_obj** basically converts each binary number to its equivalent decimal number.

Step 3 : Now we will create parse tree with denoted objects for the binary number 101.



Q.54 Describe the basic concept of axiomatic semantics.

☞ **[JNTU : Part B, Dec.-16, Marks 5]**

OR Explain about the preconditions and postconditions of a given statement mean in axiomatic semantics.

☞ **[JNTU : Part B, Dec.-17, Marks 5]**

Ans. :

- The axiomatic semantic is based on formal logic or predicate calculus.
- The main purpose of this type of semantic method is - **formal program verification**
- Axioms or inference rules are defined for each statement type in the language to allow transformations of logic expressions into more formal logic expressions.
- An Axiom is a logical statement that is assumed to be true.
- An Inference Rule is a method of inferring the truth of one assertion on the basis of the values of other assertions.

$$\frac{S_1, S_2, S_3, ..., S_n}{S}$$

- The **rule** states that if S_1, S_2, ... and S_n are true, then the truth of S can be inferred. The top part of an inference rule is call its **antecedent;** the bottom part is called it **consequent.**

Concept of Assertion :

- The logic expressions are called **assertions.**
- An assertion before the statement or command is called precondition. This condition states the relationships and constraints among variables that are true at that point in execution.
- An assertion following a statement is called postcondition.

Concept of Weakest precondition :

- A *weakest precondition* is the least restrictive precondition that will guarantee the postcondition.
- The axiomatic semantic is specified in the following manner

{P} statement {Q}

Where P is a precondition, Q is the postcondition.

- For example -

x=y+1 {x>1}
one possible precondition: {y>10}
weakest precondition: {y>0}

Q.55 Give an axiomatic semantics for assignment statement.

Ans. : The precondition and postcondition of an assignment statement together define precisely its meaning.

Let x = E be a general assignment statement and N be its postcondition.

Then, its precondition, M, is defined by the axiom

$M = N_x \rightarrow E$

That means M is computed as N with all instances of x are replaced by expression E. For example – Consider

a=b*4+2 {a<10}

The waeakest precondition prepared by using following condition

b*4+2 <10
b*4<8
b<2

Thus the weakest precondition and the postcondition is {b<2}.

Q.56 x:=2*y+1

y:=x-3

{y<0}

Compute the weakest precondition for given assignment statement.

Ans. : For the given assignment statements {y < 0} is associated postcondition.

Let y : = x – 3 ...(1)

but y < 0 is given postcondition. Substituting it for equation 1, we get

x – 3<0
x<3

Weakest precondition for first assignment statement.

Now consider

x :=2 * y + 1 ... (2)

Substitute x < 3 for equation 2,

2*y + 1<3
2y < 2

{y < 1} Weakest precondition for second assignment statement.

Q.57 b = (c + 10) / 3 {b > 6}.

Compute the weakest precondition for given assignment statement and post condition.

Ans. :

(c+10)/3>6
(c+10)>18
c>8

Q.58 a = 2 * (b - 1) - 1 {a > 0}

Compute the weakest precondition for given assignment statement and post condition.

Ans. :

2*(b-1)-1>0
2*(b-1)>1
(b-1)>1/2
b>3/2

Q.59 Give an axiomatic semantic for sequence with suitable example.

Ans. : Let, S_1 and S_2 be the two subsequent program statements. If S_1 and S_2 has following preconditions and postconditions

$\{P_1\}\ S_1\ \{P_2\}$

$\{P_2\}\ S_2\ \{P_3\}$

The inference rule for such statements can be given as follows -

$$\frac{\{P_1\}S_1\{P_2\}, \{P_2\}S_2\{P_3\}}{\{P_1\}S_1,S_2,\{P_3\}}$$

For example : Consider following sequence of statements -

```
y=2*x+2;
x=y+2;   {x<10}
```

The weakest precondition can be computed as

```
y+2<10
y<8
2*x+2<8
2*x<6
x<3
```

∴ The weakest precondition for the first assignment is {x<3}.

Q.60 Compute the weakest precondition for the following sequences of assignment statements and post condition.

a = 2 * b + 1;

b = a – 3

{b < 0}

Ans. :

Consider b = a–3

a–3<0

a<3 ...(1)

Hence the weakest precondition is **{a<3}**

Now consider, a = 2*b+1

Hence, by considering equation (1), we get -

```
2*b+1<3
2*b<2
```

Hence the weakest precondition is **{b<1}.**

Q.61 Give axiomatic semantic for selection statement. [JNTU : Part B, Marks 5]

Ans. :

- The general form of selection statement is
  If B then S_1 else S_2
- The inference rule for selection statement is

$$\frac{\{B \text{ and } P\}S_1\{Q\}, \{\text{not } B \text{ and } P\}S_2\{Q\}}{\{P\}\text{if B then } S_1 \text{ else } S_2\ \{Q\}}$$

- **Example** - consider following selection statement

```
If (x > 0) then
    y = y – 1;
else
    y = y + 1;
{y > 0}
```

- **Step 1 :** We can use the axiom for assignment on the then clause as

$y = y - 1\ \{y > 0\}$

$y - 1 > 0$

$\{y>1\}$

This produces the precondition **{y>1}**

- **Step 2 :** Now we apply the same axiom to the else clause

$y = y + 1\ \{y > 0\}$

$y+1>0$

This produce precondition **{y > – 1}**

- **Step 3 :** From above two steps we get {y>1} and {y> – 1}. But {y>1} =>{y> – 1}.

Hence{y>1} is the precondition for the above given selection statement.

Q.62 Compute the weakest precondition for the following selection construct and its postcondition :

a. if (a == b)

b = 2 * a + 1

else

b = 2 * a;

{b > 1}

Ans. : Step 1 : We can use the axiom for assignment on the statement when if condition is true

b=2*a+1 with {b>1}

2*a+1>1

2*a>0

a>0

Step 2 : Now we will consider the axiom for assignment on the statement when else part condition is true

b=2*a

2*a>1

a>1/2

If a>1/2 then it implies a>0, Hence the weakest precondition is **{a>1/2}**

Q.63 Give the axiomatic semantic for Logical Pretest Loop.

Ans. :

- The logical pretest loop is normally denoted using the **while** statement.
- The corresponding step in the axiomatic semantics of a while loop is finding an assertion called a loop invariant, which is crucial to finding the weakest precondition.
- A loop invariant is a property of a program loop that is true before and after each iteration.
- The inference rule for computing the precondition for a while loop is as follows :

$$\frac{\{I \text{ and } B\}S\{I\}}{\{I\} \text{ while B do S end } \{I \text{ and not } B\}}$$

- Where I is loop invariant, B is a condition specified in While clause and S is statement to be execute when B is true. Note that the post condition for while loop is when loop invariant is true but the condition specified in while clause is not true, hence we write the postcondition as {I and not B}.

Concept of Loop invariant in while loops more in detail -

A loop invariant I must meet the following conditions :

1. The precondition denoted by P such that P => I (the loop invariant must be true initially, in order to run the loop)
2. {I} B {I} (evaluation of the Boolean must not change the validity of I)
3. {I and B} S {I} (I is not changed by executing the body of the loop)
4. (I and (not B)) => Q (if I is true and B is false, Q is implied), note that Q is a post condition.
5. The loop terminates
6. The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition.

Q.64 Explain about denotational semantics and axiomatic semantics for common programming language features.

☞ **[JNTU : Part B, March-16, Marks 5]**

Ans. : Refer Q.53 and Q.54.

Q.65 Differentiate between static and dynamic semantic.

☞ **[JNTU : Part A, March-16, Marks 2]**

Ans. :

Static Semantic :

- Static semantics is more on the legal forms of programs (syntax rather semantics) and is only indirectly related to the meaning of the programs during execution.
- Static semantics is named as static because the analysis required checking the specifications that can be done at compile time.

Dynamic Semantic :

- Dynamic semantics is describing the meaning of the programs. Programmers need to know precisely what statements of a language do.
- Compile writers determine the semantics of a language for which they are writing compilers from English descriptions.

Fill in the Blanks for Mid Term Exam

Q.1 Programming language FOTRAN stands for _____.

Q.2 The COBOL is a well known programming language for _____ applications.

UNIT - II

2 Data Types and Fundamental Concepts

Part I : Names, Binding and Scopes

2.1 Names and Variables

Q.1 What are names ?

☞ [JNTU : Part A, Marks 2]

Ans. : • Names are used in programs to denote different entities such as variables, types and functions.

• Another term for names is **identifier**.

Q.2 What are design issues for names ?

☞ [JNTU : Part B, Dec.-16, Marks 5]

Ans. : The primary design issues of names are -

1) Are names case insensitive ?

There are some languages like Pascal and Ada which are case insensitive, on the other hand the languages like C, C++, Java are case sensitive. That means in Pascal variable index and INDEX are the one and the same but in C these are treated as two different variables.

2) Are special words of language reserved words or keywords ?

Most languages have a predefined collection of names called reserved words or keywords that carry special meaning and cannot be used as identifiers. Reserved words include names that are useful in parsing a program because they identify major constructs. In C-like languages these keywords include i nt, if, and while.

Q.3 What is the problem with case sensitive names?

☞ [JNTU : Part A, Nov.- 15, Marks 2]

Ans. : The writability problem occurs with case sensitive names. The programmer has to remember the name of the variable with correct case. For example – in Java for converting a string to integer we use the method **parseInt**. If the programmer writes it as **parseint or ParseInt** then these cannot be recognized due to case sensitiveness. Hence the programmer need to remember the method with correct case sensitivity such as parseInt. Some programmer may find it inconvenient.

Case-sensitive violates the design principle that language constructs that look similar should have similar meanings.

Q.4 What is variable ? What are the attributes of a variable ?

☞ [JNTU : Part A, Nov.-15, Dec.-16, Marks 3]

Ans. : Variable is binding of name to a memory address.

A program variable has four basic attributes and those are :

1) **Name :** The variable name is introduced during the declaration statement. For example in C

```
int a;
```

Here the variable name a is specified at declaration statement.

2) **Address :** The address of variable is machine address is what is required when the name of a variable appears in the left side of an assignment. It is possible to have multiple variables that have the

same address. When more than one variable name can be used to access the same memory location, the variables are called aliases

3) **Type :** The type of variable determines the range of values the variable can store For example – If the variable is character of type then its value ranges from -128 to +127

4) **Value :** The value of a variable is the contents of the memory cell or cells associated with the variable.

i) l-value

- The term lvalue originally referred to objects that appear on the left (hence the 'l') hand side of an expression. An ***lvalue*** represents an object that occupies some identifiable location in memory
- For example

```
Int x;
x=10;
```

Here variable x is a lvalue, because it is an object with an identifiable memory location. On the other hand

```
10=x;
```

Is invalid.

ii) r-value :

- The r value is encoded value which can be stored in the location associated with the l-value variable. The encoded representation is interpreted according to the variable's type.
- For example :

```
int x;
x=(3+4);
```

- The value of (3+4) represents the r-value.
- l_values and r_values are the main concepts related to program execution. The object is represented with the help of <lvalue, rvalue>.
- Program instructions access variables through their l_value and possibly modify their r_value.

2.2 Concept of Binding

Q.5 What is binding ? Explain the concept of static and dynamic binding.

☞ [JNTU : Part A, Marks 3]

Ans.:

- Any program contains various entities such as variables, routines, control statements and so on. These entities have special properties. These properties are called **attributes**. For example - the programming entity **routine** or **function** has number of parameters, type of parameters, parameter passing methods and so on. Specifying the nature of attributes is called **binding**.
- When the attributes are bound at program translation time then it is called **static binding**.
- When the attributes are bound at program execution time then it is called **dynamic binding**.

Q.6 What are the advantages and disadvantages of dynamic type binding ?

☞ [JNTU : Part B, Nov 15, Marks 5]

Ans. : Advantages :

1) It is flexible.

2) There are no definite types declared for example in PHP, JavaScript.

Disadvantages :

1) All required information is present only at the run time and not at the compile time.

2) The type error detection by the compiler is difficult.

Q.7 Differentiate between Compile time binding and run time binding.

☞ [JNTU : Part A, Marks 3]

Ans. :

Sr. No.	Compile Time Binding	Run Time Binding
1	When the attributes of variables, routines and control statements are bound at translation time or compilation time of the program then it is called compile time binding.	When the attributes of variables, routines and control statements are bound at the execution time or run time then it is called run time binding.

2	All required information is present at the compile time.	All required information is present at the run time.
3	It is also called as early binding.	It is also called as late binding.
4	Execution is fast.	Execution is slow.
5	It is efficient.	It is flexible.
6	For example – overloaded function call, overloaded operators.	For example – Virtual function in C++.

2.3 Scope and Scope and Lifetime

Q.8 Explain the scope and Life time of variable.

☞ [JNTU : Part A, Nov.-15, March-18, Marks 3]

Ans. : Scope : The scope of a variable is the range of statements in which the variable is visible

Lifetime of Variable: The lifetime of that variable is the period of time beginning when the method is entered and ending when execution of the method terminates.

The lifetime of that variable is the period of time beginning when the method is entered and ending when execution of the method terminates.

For example - In C and C++, for example, a variable that is declared in a function using the specifier static is statically bound to the scope of that function and is also statically bound to storage. So, its scope is static and local to the function, but its lifetime extends over the entire execution of the program of which it is a part.

Q.9 Explain the lifetime of the variable.

☞ [JNTU : Part A, March-16, Marks 2]

Ans.: Refer Q.8.

Q.10 What are local and non- local variables ?

☞ [JNTU : Part A, Marks 2]

Ans.: Local variable is local in a program unit or block if it is declared there.

Non-local variable of a program unit or block are those that are visible within the program unit or block but are not declared there.

Q.11 What is block ?

☞ [JNTU : Part A, Marks 2]

Ans. : Block is one complete section of relevant code.

Q.12 Write a note on- declaration order.

☞ [JNTU : Part A, Mark

Ans. :

- C99, C++, Java, and C# allow variable declarations to appear anywhere a statement can appear.
- In C99, C++, and Java, the scope of all local variables is from the declaration to the end of the block
- In C#, the scope of any variable declared in a block is the whole block, regardless of the position of the declaration in the block. However, a variable still must be declared before it can be used
- For example, consider the following C# code :

```
{
    int a;
    ...
    {
        int a; // Illegal
        ...
    }
    ...
}
```

- It is illegal because the scope of a declaration is the whole block.
- In C++, Java, and C#, variables can be declared in for statements. In that case, the scope of such variables is restricted to the for construct.

Q.13 Explain the concept of global scope in detail.

☞ [JNTU : Part B, Marks

Ans. : The global scope is a scope of a variable that remains through-out the program execution.

For example, C and C++ have both declarations and definitions of global data.

A declaration outside a function definition specifies that it is defined in another file.

A global variable in C is implicitly visible in all subsequent functions in the file.

A global variable that is defined after a function can be made visible in the function by declaring it to be external, for example -

```
extern int count;
```

2.4 Referencing Environment

Q.14 What is referencing environment ?

☞ [JNTU : Part A, Marks 2]

Ans. : • The referencing environment of a statement is the collection of all names that are visible in the statement

- In a static-scoped language, it is the local variables plus all of the visible variables in all of the enclosing scopes
- The referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to variables from other scopes during run time.

2.5 Named Constant

Q.15 Explain about the named constants.

☞ [JNTU : Part A, March-16, Marks 2]

Ans. : A named constant is a constant that has a name. A named constant is exactly like a variable, except its value is set at compile time (by initializing it) and CANNOT change at runtime.

For example –

```
const float pi=3.1415926
```

Part II : Data Types

2.6 Primitive Types

Q.16 Explain built in types of programming languages.

☞ [JNTU : Part B, Marks 5]

Ans. • Any programming language has a predefined set of built in data types. This built in data type reflects the behavior of underlying hardware

- Due to built in data type, the values are interpreted differently using hardware instructions.
- Following are the types of built in types of programming languages.

1 **boolean :** This data type define only two values TRUE and FALSE. In Boolean algebra the Boolean data type is used. The basic operations performed are and, not , or.

2. **character :** The single character value is represented using char data type. In C the character type variable is

   char choice;

3. **integers :** Integer value can be specified using maximum and minimum limit. In C there are four types of integer specifications int, short, long and char

4. **reals :** The real number can be represented using mantissa-exponent model. In C, the floating point number is represented using float data type.

Q.17 What is primitive data type ?

Ans. : The primitive data types are elementary data types which are not built from some other types. The values of primitive data types are **atomic** and cannot be decomposed into simpler constituents.

Many times the built in data types can be interchangeably used with built in data type but there are exceptions. **For example -** enum in C is a primitive data type which is used to define new constants.

2.7 Character and String Types

Q.18 What are the design issues for character string types? Discuss.

☞ [Part B, Nov 15, Marks 5]

Ans. : A character string type is one in which values are sequences of characters.

The two most important design issues for character string types are –

1) Is it a primitive type or just a special kind of character array ?

- C and C++ use char arrays to store char strings. These char strings are terminated by a special character **null.**
- In Java strings are supported by **String** class whose value are constant string, and the **String Buffer** class whose value are changeable and are more like arrays of single characters.
- C# and Ruby include string classes that are similar to those of Java.
- Python strings are immutable, similar to the String class objects of Java.

2) Is the length of objects static or dynamic ?

- The length can be static and set when the string is created. For example – in C, C++, Java we can have static length string.
- When the string length is dynamic then it has no maximum limit. For example – Perl and JavaScript provide this kind of facility.

Q.19 What are various design choices for string length?

☞ **[JNTU : Part B, Dec.-16, Marks 5]**

Ans. : • The string length can be static or can be dynamic. There are three kind of design choices for string length -

- **Static String Length :** The length can be static and set when the string is created. For example – in C, C++, Java we can have static length string. This is a choice for immutable objes.
- **Limited Dynamic String Length :** This option allows strings to have varying length up to a declared and fixed maximum set by variable's definition.
- **Dynamic String Length :** When the string length is dynamic then it has no maximum limit. For example – Perl and JavaScript provide this kind of facility.

Q.20 Explain in brief the implementation of character string type.

☞ **[JNTU : Part B, Marks 5]**

Ans. : For each of the following type of strings the implementation methods vary -

- **Static Length Strings :** A descriptor for a static character string type, which is required only during compilation, has three fields.
  - Name of the type
  - Type's length
  - Address of first char

It is as shown in following figure –

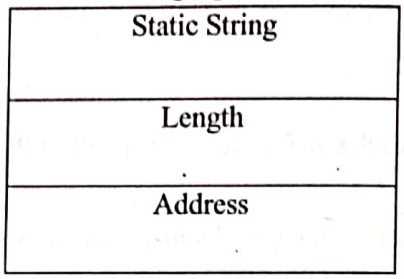


Fig. Q.20.1 Compile - time Descriptor

- **Limited Dynamic Length Strings :** It needs a run-time descriptor for length to store both the fixed maximum length and the current length. The limited dynamic strings of C and C++ do not require run-time descriptors, because the end of a string is marked with the null character. They do not need the maximum length, because index values in array references are not range-checked in these languages It is as shown by following figure -

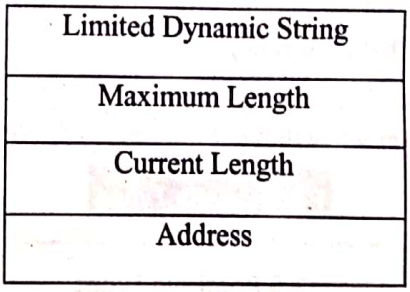


Fig. Q.20.2 Run Time Descriptor

- Dynamic Length strings: Dynamic length string require more complex storage management. The length of a string is not fixed hence storage may grow or shrink dynamically.

2.8 User Defined Ordinal Types

Q.21 Differentiate between user defined and primitive data types with an example.

☞ [JNTU : Part A, Dec.-17, Marks 3]

Ans. : Primitive data types are the data types that are provided by the **language**. For example - In C, the int, float, double are all primitive data types.

User defined data types : These are the data types that are defined by the **programmer** with the help of primitive data types. For example - In C structure is an user defined data type created using the keyword struct -

```
struct student {
int roll;
char name[10];
};
```

Q.22 What is ordinal type ? Enlist two user defined ordinal types supported by the programming languages.

☞ [JNTU : Part A, Marks 2]

Ans.: An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers.

There are two user-defined ordinal types – 1) Enumeration Type 2) Subrange

Q.23 Explain the enumeration type with suitable example.

☞ [JNTU : Part B, Marks 3]

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

- For example in C# we can define the enum data type as follows –

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

- The enumeration constants are typically implicitly assigned the integer values, 0, 1, …, but can explicitly assigned any integer literal in the type's definition. That means, in following example we can explicitly assign the enumeration constants

```
enum suit {
  club = 0,
  diamonds = 10,
  hearts = 20,
  spades = 3,
};
```

Q.24 What are the design issues for enumeration types ?

☞ [JNTU : Part A, Marks 2]

Ans. : Following are design issues for enumeration types -

- Is an enumeration constant allowed to appear in more than one type definition ?
- If enumeration constants appear in more than one type definition, then how the type of an occurrence of that constant is checked ?
- Are enumeration values coerced to integer ?
- Any other type coerced to an enumeration type ?

2.9 Arrays

Q.25 What is arrays ?

☞ [JNTU : Part A, Marks 2]

Ans. : • Array is collection **similar data type elements**.

- It is **homogeneous** aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element.
- Individual array element is specified using **subscription expression**.
- **For example** - In C, C++ array can be declared as

```
int a[10];
```

Q.26 Mention the primary design issues specific to arrays.

☞ [JNTU : Part A, March-17, Marks 3]

Ans. : The primary design issues specific to arrays are the following :

(1) What types are legal for subscripts ?

(2) Are subscripting expressions in element references range checked ?

(3) When are subscript ranges bound ?

(4) When does allocation take place ?

(5) Are ragged or rectangular multidimensional arrays allowed, or both ?

(6) Can arrays be initialized when they have their storage allocated ?

(7) What kinds of slices are allowed, if any ?

Q.27 Explain the subscript binding for arrays.

☞ [JTNTU : Part A, Marks 2]

Ans. : • The binding of subscript type to an array variable is usually static, but the subscript value ranges are sometimes dynamically bound.

- In C, C++ the lower bound of all index ranges is fixed at 0; In Fortran 95, it defaults to 1.

Q.28 Explain the categories of arrays based on bindings to subscript range.

☞ [JNTU : Part B, Marks 5]

Ans. : There are basically four categories -

1) Static Array : It is one in which the subscript ranges are statically bound and storage allocation is static that means it is done before run time. Advantage of this type of arrays is –Its working is efficient as there is no need to allocate and deallocate memory.

2) Fixed Stack Dynamic Array : A fixed stack-dynamic array is one in which the subscript ranges are statically bound, but the allocation is done at elaboration time during execution. The advantage of this type is - it is space efficient. A large array in one subprogram can use the same space as a large array in different subprograms.

3) Fixed Heap Dynamic Array : It is similar to fixed stack-dynamic. The storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack).

4) Heap Dynamic Array : In this type, binding of subscript ranges and storage allocation is dynamic and can change any number of times. The advantage of this type of array is – it is very flexible.

Q.29 Describe the process of Array initialization.

☞ [JNTU : Part B, Nov.-15, Marks 5]

Ans. : • Array initialization is a process in which a list of values that are put in the array in the order in which the array elements are stored in memory.

- Some language allow initialization at the time of storage allocation.
- For example – Following is an array initialization in C

```
int a[]={10,20,30};
```

- Character Strings in C & C++ are implemented as arrays of char. These arrays can be initialized to string constants –
- char name[] ="ISRO" //Note that this array will contain 5 elements and this string is terminated by **null**
- In Java the array of string can be initialized as follows –

```
String[] names=["Aditya","Aakash","Amit"};
```

Q.30 What is an array ? Explain about various array operations with an example.

☞ [JNTU : Part B, May 18, Marks 5]

Ans. : • Array is a collection of similar data type elements.

- The common array operations are - assignment, catenation, comparison for equality and inequality, and slices.
- The C-based languages do not provide any array operations, except through the methods of Java, C++, and C#.
- Perl supports array assignments but does not support comparisons.
- Python's arrays' are called lists, they behave like dynamic arrays. In this lists the objects can be of any datatype. Hence arrays are heterogeneous. Python

also supports array catenation and element membership operations. In Python, assignment with an = on lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory. For example

```
color=[10,20,30]
a=color;
```

- The a list will point to the same memory location as that of color.
- Ruby also provides array catenation.
- APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators.

Q.31 What happens when a nonexistent element of an array is referenced in Perl ?

☞ [JNTU : Part A, Marks 2]

Ans.: A reference to a nonexistent element in Perl yields undef, but no error is reported.

2.10 Associative Arrays

Q.32 Explain associative arrays, their structures and operations. ☞ [JNTU : Part B, Dec.-16, Marks 5]

Ans. : • An associative array is an unordered collection of data elements that are indexed by an equal number of values called keys.

- Each element of an associative array is a pair of a **key** and a **value.**
- Associative arrays are supported by the standard class libraries of Java, C++, C#, and F#.
- For Example: In Perl, associative arrays are often called **hashes.** Names begin with %; literals are delimited by parentheses. Hashes can be set to literal values with assignment statement as shown below –

```
%ages=("AAA" =>25, "BBB"=>10, "CCC"=>55);
```

- Subscripting is done using braces and keys. For example -

```
$ages{"CCC"}=48;
```

- In Python, the associative arrays are called as dictionaries. For example -

```
thisdict = {
  "brand": "Honda",
  "model": "Honda-Civic",
  "year": 2019
}
```

- PHP's arrays are both normal arrays and associative array.
- A Lua table is an associate array in which both the keys and the values can by any type.
- C# and F# support associative arrays through a .NET class.

2.11 Record

Q.33 Explain record data structure.

☞ [JNTU : Part B, Marks 5]

Ans.: A data structure which is collection of components of different data types is termed as record.

Specification : The attributes of record are

1. The number of components
2. The data type of each component
3. The selector used to name each component.

In C the syntax for record declaration is denoted by struct.

- For example - In C, C++, and C#, records are supported with the struct data type

```
struct {
   int roll_no;
   char name[20];
}
```

Q.34 Explain the differences between subtypes and derived types.

☞ [JNTU : Part B, Nov.-15, Marks 5]

Ans. : • A derived type is a new type that is based on some previously defined type with which it is incompatible.

- The derived type is a incompatible with its parent type.
- A sub-type is a possibly range-constrained version of an existing type.
- A subtype is compatible with its parent type.

- A derived class is a subtype if it has an is-a relationship with its parent class.
- For example –

```
subtype Week_Days is Integer range 1..7;
```

Since this is a subtype, you can (for example) add 1 to a weekday to get the next weekday. A derived type is a completely separate type that has the same characteristics as its base type. You cannot mix operands of a derived type with operands of the base type. If, for example -

```
type Week_Day is new Integer range 1..7;
```

Then we will not be able to add an integer to a weekday to get another weekday.

2.12 Tuple Types

Q.35 What is tuple ? Explain it with suitable example.

☞ [JNTU : Part A, Marks 2]

Ans. : • Tuple is a data type in which there is a sequence of elements that are not named.

- Python supports the concept of tuple type.
- The tuples are immutable. That means tuples can not be changed once created. If a tuple needs to be changed, it can be converted to an array with the list function.
- Example of Tuple

```
myTuple=(10,4.5,'red')
```

- Note that the elements of tuple need not be of same data type.
- The tuple element can be referenced with the help of index. For instance myTuple[1] will refer to the value 10, because tuple indexing begins with 1.
- For concatenating the tuples the + operator is used.

2.13 List Types

Q.36 Explain List type. Give example of Lists supported by various programming languages.

☞ [JNTU : Part B, Marks 5]

Ans. : • List is a collection of elements which were first supported in the first programming language – LISP.

- Lists in common Lisp is a data type in which elements are not separated by any punctuation mark and are delimited by parenthesis. For example –

```
(10 20 30 40)
```

- The nested lists can also be created as follows –

```
(10 (20 30) 40)
```

- The list(20 30) is nested inside the outer list.
- **List operations in Scheme :**
- CAR returns the first element of its list parameter

For example

```
(CAR '(10 20 30))          returns 10
```

- CDR returns the remainder of its list parameter after the first element has been removed. For instance –

```
(CDR '(10 20 30))          returns (20 30)
```

- CONS puts its first parameter into its second parameter, a list, to make a new list

```
(CONS 'X '(Y Z))     returns (X Y Z)
```

- LIST returns a new list of its parameters

```
(LIST 'X '(Y Z))     returns (X (Y Z))
```

- **List in Python :**
  - The list in Python serves as array.
  - Unlike the programming languages like Scheme, Common Lisp, ML, and F#, Python's lists are **mutable**.
  - Elements can be **of any type**
  - We can Create a list with an assignment

```
myList = [25, 10.4, "green"]
```

  - List elements are referenced with subscripting, with indices beginning at zero

```
ele = myList[1] Assign 10.4 to ele
```

  - List elements can be deleted with **del**

```
del myList[1]
```

2.14 Union Types

Q.37 Define Union types

☞ [JNTU : Part A, May-18, Marks 2]

Ans. : A union is a type whose variables are allowed to store different type values at different times during execution.

For example – In C we can declare union as follows

```
union Student
{
char name[30];
int rollno;
}s1,s2;
```

Here s1 and s2 are the variables of union.

We can access the elements of union using dot operator. For instance s1.rollno

Note that only one member of union is accessed at a time. For example if –

```
s1.name="AAA";
s1.rollno=10;
```

Then as an output we get s1.rollno=10 and s1.name="AAA";

Q.38 Explain the different types of Union with an example. **[JNTU :Part B, Dec 17, Marks 5]**

Ans. : Two types of unions are :

1) Free Union : The union constructs in which there is no language support for type checking is called free union. C, C++ provide such type of unions. For example –

```
union test {
    int a;
    float b;
}
union test u;
float x;
...
u.a=10;
x=u.b;
```

Note that the integer value is assigned to the float variable in the last assignment and there is no type checking mechanism.

2) Discriminated Union : Discriminated union is a union structure in which each union include the type indicator. The discriminated unions are supported by ALGOL 68, ML, HASKELL.

Q.39 Write EBNF description for the C Union.

☞ **[JNTU : Part A, March-16, Marks 3]**

Ans. :

```
<struct-or-union-specifier> ::=
                        <struct-or-union> <identifier> { {<struct-declaration>}+ }
                      | <struct-or-union> { {<struct-declaration>}+ }
                      | <struct-or-union> <identifier>

<struct-or-union> ::= struct
                    | union

<struct-declaration> ::= {<specifier-qualifier>}* <struct-declarator-list>

<specifier-qualifier> ::= <type-specifier>
                        | <type-qualifier>
```

Q.40 Explain the syntax of Ada Union Type. ☞ [JNTU : Part B, May-18, Marks 5]

Ans. : The syntax is as follows -

```
type Typ (Choice : Discrete_Type) is record
 case Choice is
   when Choice_1 =>
     N1 : Typ1;
     ...
   when Choice_2 | Choice_3 =>
     ...
   when others =>
     ...
 end case;
end record;
```

For example -

```
type Traffic_Light is (Red, Yellow, Green);
 type Union (Option : Traffic_Light := Traffic_Light'First) is
   record
     -- common components

     case Option is
       when Red =>
         -- components for red
       when Yellow =>
         -- components for yellow
       when Green =>
         -- components for green
     end case;
   end record;
```

Q.41 Distinguish between Pascal union types and Ada Union types. ☞ [JNTU : Part A, March-16, Marks 3]

Ans. : The ADA union is similar to Pascal union types but there is a difference between the two –

1) In ADA, there are no free unions. That means the tag is specified with union declaration.
2) When tag is changed, all appropriate fields must be set.

2.15 Pointer and Reference Types

Q.42 Write the difference between C++ pointer and Java Reference variable. ☞ [JNTU : Part A, March-17, Marks 2]

Ans. : Reference : A reference is a variable that refers to something else and can be used as an alias for that something else.

Pointer : A pointer is a variable that stores a memory address, for the purpose of acting as an alias to what is stored at that address.

So, a pointer is a reference, but a reference is not necessarily a pointer.

The differences between the C++ pointer and Java Reference are -

(1) Java doesn't support pointer explicitly, But java uses pointer implicitly.

(2) C/C++ allows pointer arithmetic but Java References not.

(3) References are strongly typed.

Q.43 How pointers are represented and used in C and C++ ?

[JNTU : Part A, May-18, Marks 3]

Ans. : Pointrs are basically the variables that contain the **location of other data objects**. It allows to construct complex data objects. In C or C++ pointer are data objects that can be manipulated by the programmer. For example -

```
int *ptr;
ptr=malloc(sizeof(int);
```

The * is used to denote the pointer variable.

Q.44 What are the uses of pointers ?

Ans. :

(1) Provide the power of **indirect addressing**.

(2) Provide a way to manage **dynamic memory**. A pointer can be used to access a location in the area where storage is dynamically created usually called **a heap**.

Q.45 Describe about the pointers in FORTRAN 90, Ada, Pascal with an example.

[JNTU : Part B, Dec.-17, Marks 5]

Ans. : FORTRAN 90

The first step in using Fortran pointers is to determine the variables to which you will need to associate pointers. They must be given the TARGET attribute in a type statement. **For example -**

```
real, target :: a, b(1000), c(10,10)
integer, target :: i, j(100), k(20,20)
```

Then we define some pointers as -

```
real, pointer :: pa, aptr, pb(:), pc1(:), pc2(:,:)
```

The type of the pointer must match the type of the intended target.

ADA

Pointers in ADA are known as **access types**. There are four kinds of access types in Ada : **pool access types, general access types, anonymous access types,access to subprogram types.**

For example –

```
type Int    is range -100 .. +500;
type Acc_Int is access Int;
```

PASCAL

Pascal support use of pointers. Pointers are the variables that hold the address of another variable.

For example –

```
Program pointers;
type
 Buffer = String[255];
 BufPtr = ^Buffer;
Var B  : Buffer;
 BP : BufPtr;
 PP : Pointer;
```

In this example, BP is a pointer to a Buffer type; while B is a variable of type Buffer.

Q.46 Why are reference variables in C++ better than pointers for formal parameters ?

☞ [JNTU : Part B, Marks 5]

Ans. : • Reference variables are aliases of another variable while pointer variables are special type of variables that contain the addresses of another variable.

- For example –

```
int a;
int &ref_a=a; //reference variable

int a;
int *ptr_a=&a; //pointer variable
```

- Reference and pointers both can be used to refer the actual variable by providing direct access to the variables.
- In pointers to access the value of actual variable we need to explicitly dereference the pointer variable by using value at address or using * operator. In references, to access the value of actual variable we do not need to explicitly dereference the reference variable.
- In C++, the reference variable is a better choice for formal parameter than pointer. It must be initialized with the address of some variable in its definition and after initialization a reference type variable can never be set to reference any other variable,
- Reference can never point to NULL value whereas pointer can point to NULL. Thus with the use of reference there can not be NULL pointer assignment problem.

Q.47 Why do we use pointers ? Also explain the problems in pointer implementation.

Ans. : Pointers are basically the variables that contain the location of other data objects. It allows to construct complex data objects. In C or C++ pointer are data objects that can be manipulated by the programmer. For example -

```
int *ptr;
ptr=malloc(sizeof(int);
```

The * is used to denote the pointer variable.

Following are **implementation problems** when pointers are used -

1. **Management of heap storage area :** Due to creation of objects of different sizes during execution time requires management of general heap storage area.
2. **The garbage problem :** Sometimes the contents of the pointers are destroyed and object still exists which is actually not at all accessible.
3. **Dangling references :** The object is destroyed however the pointer still contains the address of the used location, and can be wrongly used by the program.

Q.48 For a language that provides a pointer type for programmer- constructed data objects and operations such as new and dispose that allocate and free storage for data objects, write a program segment that generates a dangling reference. If one or the other program segment cannot be written, explain why.

Ans. : The dangling reference is a live pointer that no longer points to a valid object.

```
var q : integer;
```

```
var p : ↑ integer;
begin
    new(p);
    q = p;
    dispose(p);
    ...
end
```

The live pointer p has created reference for q and then p is deleted. This creates dangling reference for q.

Q.49 Define the term dangling pointer ?

Ans. : Refer Q.47.

2.16 Type Checking

Q.50 Discuss the advantages and disadvantages of static and dynamic type checking.

☞ [JNTU : Part B, Marks 7]

Ans. : Static Type Checking

Advantages :

1) Programming errors are detected at early stage.
2) There are more opportunity for compiler optimization
3) Reduced memory usage and increased runtime efficiency.

Disadvantages :

1) It cannot handle change in requirements.
2) It cannot handle change in variable type.

Dynamic Type Checking

Advantages :

1) It has better prototyping system with changing or unknown requirements.
2) It allows programs to generate types and functionality based on run time data.
3) It is flexible.

Disadvantages :

1) Detection of errors is late i.e. at run time.
2) More cost is required in development process.
3) Complex to fix errors at runtime.
4) Worst execution time.
5) More code and effort is required to write the exceptions.
6) Type checker need to check all classes during run-time

Q.51 Explain the concept of type systems.

☞ [JNTU : Part B, Mark

Ans. : Definition : The type system is defined as set of rules used by the language to structure and organize collection of various types.

Type systems play an important role in determining the semantics of a language.

Let use define some important terminologies that are associated with type system

1. **Type :** The type can be defined as a specific set of values and particular set of operations that can be applied to such values.
2. **Object or data object :** The data object can be defined as location in memory with assigned name. Refer Fig. Q.51.1.

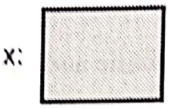


Fig. Q.51.1 Data object x

3. **Type error :** The illegal operation that manipulate the data objects are called type error.
4. **Type safe or type secure :** The operation or a program is said to be type safe if all the operations in a program are guaranteed to always apply to data of the correct type. Thus in type safe operations no type error will occur.

Q.52 Differentiate between static and dynamic type checking.

☞ [Part B, Marks

Ans. :

Sr.No.	Static Type Checking	Dynamic Type Checking
1.	Static checking is a kind of checking which is done at **compile time**.	Dynamic checking is a kind of checking which is done at **run time**.
2.	For example- In **C, C++** the static type checking is performed.	For example – In **Perl and Prolog** the dynamic type checking is performed.
3.	The static type checking is also called as **compile time checking**.	The dynamic type checking is also called as **execution time checking**.
4.	**Merits :** It allows detection of programming errors without running the code. Due to this many errors can be detected quickly, reliably and automatically.	**Merits :** Different Architectures VLIW Architecture, DSP Architecture, SoC architecture, MIPS Processor and programming (Chapter - 6) The dynamic type checking brings flexibility in program design.
5.	**Demerit :** It increases the compile time overhead.	**Demerit :** The checking information need to be kept maintained during the execution. This causes extra storage requirement during execution of program.

2.17 Strong Typing

Q.53 What is strong type system ? [JNTU : Part A, Marks 2]

Ans.: • The **strong type system** is a type system that guarantees not to generate type errors. A language with strong type system is said to be strongly typed language. A type system is said to be weak if it is not strong. Hence a language with weak type system is said to be weakly typed language.

- There are **static type system** which are the **strong type systems.** The static type system is a type system in which type of every expression is known at compile time.

2.18 Type Equivalence

Q.54 Explain the following terms : i) Type checking ii) Coercion iii) Type compatibility [JNTU : Part B, Marks 6]

Ans. : i) Type Checking : Type checking is a programming technique in which compatibility of variable with data type is checked. There are two types of type checking –

1) **Static type checking :** The static checking is a kind of checking which is done at compile time. For example in C, C++ dynamic type checking is performed. It is also called as **compile time checking.**
2) **Dynamic Type Checking :** The dynamic checking is a kind of checking which is done at run time. For example in Prolog and Perl the dynamic type checking is done. It is also called as **Execution time checking.**

(ii) Coercion : Type coercion is the process of converting one type to another.

There are two types of conversions : implicit conversion and explicit conversion.

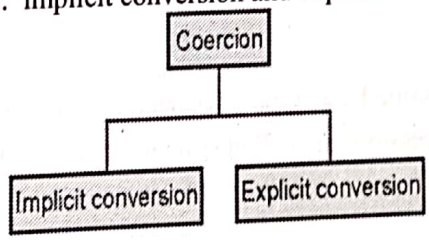


Fig. Q.54.1 Two types of conversion

If the conversion is done automatically by the compiler then it is called **implicit conversion**. The implicit conversions are also called as coercions. In this type of conversion there is no loss of information for example integer can be converted to float but not vice versa. That means in conversion from integer to float there is no loss of information but for converting the information from float to integer there is a loss.

The conversion is said to be **explicit** if the programmer specifically writes something for converting one type to another.

For example

```
int xyz,p;
p=(float)xyz;
```

The identifier **xyz** is type-caste and this is how explicit conversion from int to float takes place.

iii) Type compatibility :

- A strict type system in which one operand type A is allowed to perform operation with another operand with type B. This feature of type system is called type compatibility.
- Type compatibility is also called as type equivalence or conformance.
- During type compatibility conformance, type checking procedures can verify that all operations are correctly invoked. That means type of operands are type compatible with type of operations.

Q.55 Explain about type compatibility with an example. ☞ [JNTU : Part B, March-16, Marks 5]

Ans. : Refer Q.54.

Q.56 Explain name and structural equivalence. ☞ [JNTU : Part B, Marks 5]

Ans. : Name Equivalence :

In the name equivalence the type expressions are given the names. Hence two type expressions are said to be name equivalent if and only if they are identical.

For example : Consider a code segment as follows :

```
typedef struct Node
 {
      int x;
}node;
node *first,*second;
struct Node *last1,*last2,;
```

In the above code the variables **first** and **second** are name equivalent similarly variables **last1** and **last2** are name equivalent. But **first** and **last1** are not name equivalent as the names for these two type expressions are different. Although all the four variables **first, second,last1,last2** are similar under structural equivalence since they are representing the same type **pointer('struct Node')**.

Structural Equivalence

- When two expressions are the same basic type or formed by applying the same constructor to structurally equivalent types then those expressions are called structurally equivalent.

For example :

```
struct student
{
 int ID;
char name[10];
};
struct employee
{
int ID;
char name[10];
};
student s1;
employee e1;
If we write
s1=e1;
```

then it is said to be valid, because s1 and e1 are structurally equivalent. Thus s1 and e1 are structurally equivalent.

Q.57 What is subtyping ?

☞ [JNTU : Part A, Marks 2]

Ans. : A **subtype** is a new type based on a parent type but usually with a more restricted range. It inherits all of the characteristics of the parent type and in addition, it can be freely intermixed with the parent type in calculations and assignment statements.

For example - following code defines the concept of subtypes

```
type NEW_INT     is new INTEGER range 12..55;
type NEW_INT_TYPE is new INTEGER;
subtype SUB_INT    is NEW_INT;
subtype NEW_SUBTYPE is NEW_INT range 12..55;
```

Q.58 Write short note on type structure of C++ ?

☞ [JNTU : Part B, Marks 5]

Ans.: • The type structure of C++ is divided into two main categories - **Fundamental types** and **Derived Types.**

- The fundamental type is – **integer** and **float**. The integer data type is classified into int, char, short, long. The float data type is classified into float and double
- The derived type of actually derived from the fundamental types- For example, arrays, class, structure, union, functions and pointers pointers

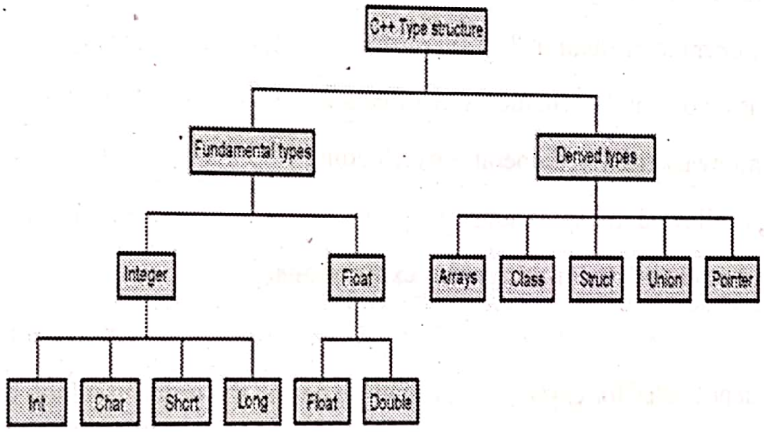


Fig. Q.58.1 Type structure of C++

- Arrays is collection of similar data type elements. For example

```
int a[10];          //It will create an array of 10
                    //integer elements
```

- The class, structure and union allows the programmer to create user defined data types. The class is well known entity in C++ by which one can encapsulate data members and member functions in C++. For example

```
class stack
{
   private:
      int s[10];
      int top;
   public:
      void push(int item)
      int pop();
}
```

The above class creates a **stack** data structure using class.

Part III : Expression and Statements

2.19 Arithmetic Expressions

Q.59 What is arithmetic expression ?

☞ [JNTU : Part A, Marks

Ans. : Arithmetic expressions consist of operators, operands, parentheses, and function calls.

For example x=y+2*sqrt(25);

The purpose of an arithmetic expression is to specify an arithmetic computation

Q.60 Design issues for arithmetic expressions.

☞ [JNTU : Part A, Marks

Ans. : Design issues for arithmetic expressions are -

(1) What are the operator precedence rules ?

(2) What are the operator associativity rules ?

(3) What is the order of operand evaluation ?

(4) Are there restrictions on operand evaluation side effects ?

(5) Does the language allow user-defined operator overloading ?

(6) What mode mixing is allowed in expressions ?

Q.61 Explain precedence and associativity in arithmetic expressions.

Ans. : Precedence :

- The operator precedence rules for expression evaluation define the order in which the operators of different precedence levels are evaluated.
- Many languages also include unary versions of addition and subtraction.
- Unary addition (+) is called the identity operator because it usually has no associated operation and thus has no effect on its operand.
- In all of the common imperative languages, the unary minus operator can appear in an expression either at the beginning or anywhere inside the expression, as long as it is parenthesized to prevent it from being next to another operator. For example, unary minus operator (-) :

x + (- y) * z // is legal

x + - y * z // is illegal

- Exponentiation has higher precedence that unary minus.
- The precedence of Ruby and C language operators is as given in following table

Precedence	Ruby	C/C++
Highest	**	postfix ++,--
	unary +, -	prefix ++,--,unary +,-
	*,/,%	*,/,%
Lowest	binary +, -	binary +, -

Associativity :

- The operator associativity rules for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated. An operator can be either left or right associative.
- Typical associativity rules :
  - Left to right, except exponentiation **, which is right to left.
  - For example – a – b + c // left to right
  - Sometimes unary operators associate right to left (Fortran)

```
A ** B ** C // right to left
(A ** B) ** C // in Ada it must be parenthesized
```

- The associativity rules for a few common languages are given here :

Language	Associativity Rule
Ruby, FORTRAN	**Left :** *, /, +, - **Right :** **
C-Based Languages	**Left :** *, /, %, binary +, binary - **Right :** ++, --, unary –, unary +

Q.62 Give the operand evaluation order. ☞ [JNTU : Part A, Marks 2]

Ans.: The operand evaluation order is as given below -

1) Variables : fetch the value form memory
2) Constants : sometimes a fetch from memory; sometimes the constant in the machine language instruction and not require a memory fetch.
3) Parenthesized expression : evaluate all operands and operators first

2.20 Overloaded Operators

Q.63 How user defined operator overloading harm the readability of a program ? Explain.

☞ [JNTU : Part B, Dec.-16, Marks 7]

Ans. : User-defined operator can overloading harm the readability of a program because the built in operator has the precision and compiler knows all the precision between the operators, and it works on that precision. User can also create its own operator but the compiler does not come to know how to make precision of this operator. This will be the cause of the overloading harm.

2.21 Type Conversions

Q.64 Explain type conversion in detail.

Ans. : There are two type of conversions – (1) narrowing conversion and (2) Widening Conversion.

A narrowing conversion is one that converts an object to a type that cannot include all of the values of t original type. For example - conversion from double to float.

A widening conversion is one in which an object is converted to a type that can include at least approximatio to all of the values of the original type. For example - from int to float.

There are two types of type conversions -

(1) **Implicit Type Conversion :** This type of type conversion is also called as coercion. The one data typ element is automatically converted into another data type by widening conversion.

- For example in the following code the character data type of variable c is converted implicitly to integer type.

```
int num = 10;
char c = 'm'; /* ASCII value is 109 */
int sum;
sum = num + c;// output:119
In Ada, there are virtually no coercions in expressions
In ML and F#, there are no coercions in expressions
```

(2) Explicit Type conversion : It is also called as type casting is C based languages.

Syntax is

```
(type-name) expression
```

For example –

```
float a = 1.5;
int b = (int)a + 1;//output will be 2
```

Q.65 What is the major disadvantage of coercion ?

Ans. : The disadvantage of coercion is They decrease in the type error detection ability of the compiler .

Q.66 What is mix-mode expression ?

Ans.: If the operands in the expression are of different data types then that expression is called mix mode expressio

2.22 Relational and Boolean Expressions

Q.67 Explain relational and boolean expressions in detail with suitable examples.

Ans. : Relational Expression

- Any relational expression is comprised of relational operator and two operands. A relational operator operator that compares the values of its two operands
- The value of a relational expression is Boolean, unless it is not a type included in the language
- Operator symbols used vary somewhat among languages (!=, /=, .NE., <>, #)
- The syntax of the relational operators available in some common languages is as follows :

Operation	Ada	C-Based Languages	Fortran 95
Equal	=	==	.EQ. or ==
Not Equal	/=	!=	.NE. or <>
Greater than	>	>	.GT. or >
Less than	<	<	.LT. or <
Greater than or equal	>=	>=	.GE. or >=
Less than or equal	<=	<=	.LE. or >=

- JavaScript and PHP have two additional relational operator, === and !==

Boolean Expression

- In boolean expression, the operands are Boolean and result of the expression is Boolean.
- Boolean operators used in various languages are –

FORTRAN 90	C	Ada
and	&&	and
or	||	or
not	!	not

During evaluation of Boolean expression, the leftmost operator is evaluated first because in C language the relational operators are left associative producing either 0 or 1.

2.23 Short Circuit Evaluation

Q.68 Explain the short circuit evaluation with suitable example.

Ans. : Short circuit evaluation is a kind of evaluation in which the expression is evaluated only if it is necessary. MODULA-2 uses short circuit evaluation for OR and AND operators. The short circuit evaluation is also allowed in C as well.

For example –

```
while( p>=0 && q<=10)
 i=i+1;
```

If p>=10 is true then only the control reaches to test q<=10

With short circuit evaluation,

i) While evaluating E1 or E2, if E1 is true then the whole expression is true and in that case E2 is not evaluated.

ii) Similarly, while evaluating E1 and E2, if E1 is false then E2 is not evaluated.

2.24 Assignment Statements

Q.69 What is the purpose of assignment statement ?

☞ [JNTU : Part A, Dec.-16, Marks

Ans. : The purpose of assignment statement is to assign the value to a variable.

Q.70 What are different types of assignment statements ? ☞ [JNTU : Part B, March-16, Marks 5]

Ans.: **1) Simple Assignments :** The language like C, C++, JAVA makes use of = operator for assignment statemen The ALGOL, PASCAL, ADA makes use of := operator for assignment purpose.

2) Conditional Targets : In Perl the conditional target can be specified as –

```
($count ? $total : $subtotal) = 0
```

The above code is equivalent to

```
if ($count){
        $total = 0
 } else {
        $subtotal = 0
}
```

3) Compound Assignment operator : This is commonly used form of assignment statement in which shorthand method of using assignment statement. For example –

```
a=a+b
```

can be written as

```
a+=b;
```

4) Assignment as an Expression : In the C-based languages, Perl, and JavaScript, the assignment statement produces a result and can be used as an operand

```
while ((ch = getchar())!= EOF)  {
...
}
```

In above code, ch = getchar() is carried out; the result (assigned to ch) is used as a conditional value for the while statement

5) Unary Assignment Operator : It is possible to combine increment or decrement operator with assignments. The operators ++ and -- can be used either in expression or to form stand-alone single-operator assignment statements. They can appear as prefix operators. For example -

```
sum = ++ count; is equivalent to count=count+1; sum=count;
```

If the same operator appears as postfix operator then

```
sum=count++; sum=count; count=count+1
```

6) Multiple Assignments : It is possible to perform multiple assignments at a time. For example - the Perl, Ruby, and Lua provide multiple-target multiple-source assignments

```
($one, $two, $three) = (100, 200, 300);
```

2.25 Mix Mode Assignment

Q.71 Explain mixed mode assignment statement with relevant example. ☞ **[JNTU : Part B, March-17, Marks 5]**

Ans.: • Assignment statements can also be mixed-mode.

- In Fortran, C, and C++, any numeric value can be assigned to any numeric scalar variable; whatever conversion is necessary is done.
- In Java and C#, only widening assignment coercions are done.
- In Ada, there is no assignment coercion.
- In all languages that allow mixed-mode assignment, the coercion takes place only after the right side expression has been evaluated. For example, consider the following code :

```
int a, b;
float c;
...
c = a / b;
```

In above code, c is float, the values of a and b could be coerced to float before the division.

Part IV : Control Structures

2.26 Selection Statements

Q.72 Explain about the control structures with an example. ☞ **[JNTU : Part B, Dec.-17, Marks 5]**

Ans. : A control structure is a control statement and the statements whose execution it controls. There are two types of control structures

(1) Selection Statements: Refer Q.73.

(2) Iterative Statements: Refer Q. 77.

Design Issue : There is only one design issue for control structure -

" should a control structure have multiple entries ?"

Q.73 Explain Selection statements with suitable example.

Ans. : For choosing one correct path from two or more paths of execution, the selection statement is used.

There are two general categories of selection statements

1) Two way Selectors : The general form of two way selector statement is -

```
if control_expression
then clause
else clause
```

Here

Control expression: These are specified in parenthesis. Some languages like C, C++ do not use the reserved word **then.** The Java, C, C++ makes use of boolean expressions for control expressions.

Clause : In most contemporary languages, the then and else clauses either appear as single statements or compound statements. C-based languages use braces to form compound statements. One exception is Perl, in which all the and else clauses must be compound statements, even if they contain single statements. In Python and Ruby, clauses are statement sequences. Python uses indentation to define clauses. For example -

```
if a> b :
    a = b
    print " a is greater than b"
```

(2) Multiple-way Selectors : It allows the selection of one of any number of statements or statement groups. For example – In C we can use switch case statement as

```
switch (choice) {
        case 1 : c=a+b;
           break;
        case 2 :  c=a-b;

           break;
        case 3 : c=a*b;
           break;
        case 4 : c=a/b;
           break;
        default : printf("exit");
}
```

Q.74 What are the design issues for multi-way selection statements ? ☞ **[JNTU : Part B, Marks 5]**

Ans. : 1. What is the form and type of the control expression ?

2. How are the selectable segments specified ?
3. Is execution flow through the structure restricted to include just a single selectable segment ?
4. How are case values specified ?
5. What is done about unrepresented expression values ?

Q.75 Explain the syntax of 'case' statement in Pascal using BNF notation and syntax graphs.

☞ [JNTU : Part B, March-17, Marks

Ans. : The syntax of 'case' statement in PASCAL is as follows –

```
case (expression) of
  Label1 : Statement1;
  Label2: Statement2;
  ...
  ...
  Labeln: Statementn;
end;
```

The BNF notation

case <tag field> <type identifier> **of** <variant> { ; <variant>}

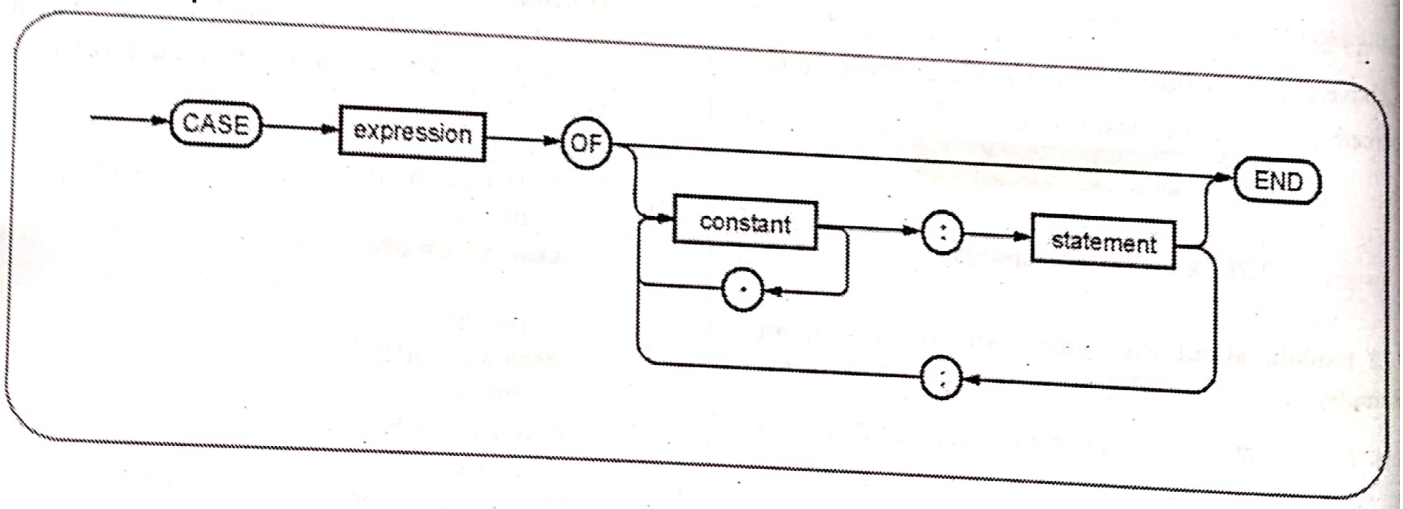
<tag field> ::= <field identifier> : | <empty>

<variant> ::= <case label list> : (<field list>) | <empty>

<case label list> ::= <case label> {, <case label>}

<case label> ::= <constant>

Syntax Graph



Q.76 Explain the practical problems associated with the if-then-else statement.

☞ [JNTU : Part B, March-17, Marks

Ans. : The BNF rules for an Ada if-then-else statement are as follows :

<if_stmt> → if <logic_expr> **then** <stmt>

if <logic_expr> **then** <stmt> **else** <stmt>

If we also have <stmt> → <if_stmt>, this grammar is ambiguous because it becomes difficult to decide which IF is associated with ELSE,

Consider the following example :

```
if A then if B then S1 else S2

as

if A then
{
  if B then
    S1
  else
    S2
}
or

if A then
{
  if B then
    S1
}
else
  S2
```

Two different interpretations will lead to ambiguity. This problem is also referred as dangling else problem.

2.27 Iterative Statements

Q.77 Explain iterative statements in detail.

Ans. : • **Definition :** Iterative statements is one that cause a statement or collection of statements to be executed zero, one, or more times.

- The repeated execution of statements can be achieved either by iterative statements or by using recursion.
- The iterative statements are normally called as **loop.**
- The General design issues for iterative statements are
  - How is iteration controlled ?
  - Where should the control mechanism appear in the loop statement ?
- The loop has **body** which is basically collection of statements whose execution is controlled by iterative statements.
- The iteration statement and the associated loop body together form an iteration statement.
- There are two important terms associated with loop and those are –
  - **Pretest :** The term pretest means that the loop completion occurs before the loop body is executed.
  - **Posttest :** The term pretest means that the loop completion occurs after the loop body is executed.
- **Example of Loop :** Following is an example of for loop used in C

```
for(i=0;i<10;i++)
{
   printf("i = %d",i);
}
```

Q.78 List and explain the design issues for all iterative control statements

☞ [JNTU : Part B, May-18, Marks 5]

Ans. : The design issues for all iterative control statements are –

- How is iteration controlled ?
- Where should the control mechanism appear in the loop statement ?

- The iteration control is done using either logical statements or using counting statement or using both i.e. the combination of logical and counting statements.
- The location of control mechanism is normally at the top or at the bottom of the loop.
- It is important to affect the control before and after execution of statement's body.
- The body of an iterative statement is the collection of statements whose execution is controlled by the iteration statement.

Q.79 Explain in detail counter-controlled loops.

☞ [JNTU : Part B, Dec.-16, Marks 5]

Ans. : • The counter-controlled loops are those loop that possess a variable called the loop variables. This loop variable is used to maintain the count value.

- The loop variables of counter controlled loops include the means of specifying the initial and

terminal values of the loop variable, and the difference between sequential loop variable values, called the stepsize.

- The initial, terminal and stepsize are called the loop parameters.
- Design issues for the counter controlled loops are –
  - What are the type and scope of the loop variable ?
  - Should it be legal for the loop variable or loop parameters to be changed in the loop body, and if so, does the change affect loop control ?
  - Should the loop parameters be evaluated only once, or once for every iteration ?
- **Example of counter controlled loop :** Following code in Java that displays the character and its' ASCII value for all lower case characters.

```
for (char z = 'a'; z <= 'z'; z++)
System.out.println(z + " = " + (int) z);
```

Q.80 What are design issues of logically controlled loop statements ? Explain briefly.

Ans. : The logically controlled loop is a kind of loop statement in which repetition control is based on a **Boolean expression** rather than a counter.

Design Issues are

1) Should the control be pretest or posttest?
2) Should the logically controlled loop be a special form of a counting loop or a separate statement ?

Q.81 Explain logically controlled loops with suitable example.

Ans. : • The logically controlled loop is a kind of loop statement in which repetition control is based on a **Boolean expression** rather than a counter.

- C and C++ have both pretest and posttest forms, in which the control expression can be arithmetic :

For example :

while(control_expression) loop body	do loop body while(control_expression)

In both C and C++ it is legal to branch into the body of a logically-controlled loop.

- Java is like C and C++, except the control expression must be Boolean.

Q.82 Explain user-located loop control mechanism.

☞ [JNTU : Part B, Marks

Ans. : This is a control structure in which programmer choose a location for loop control. This location generally other than the top or bottom of the loop.

In C or C++ the user located loop control mechanism is accomplished using unlabeled exits such as **break.**

Following is an example user located loop control using C#

```
UP:
for (row = 0; row < n; row++)
for (col = 0; col < m; col++) {
sum += a[row][col];
if (sum > 1000.0)
break UP;
}
```

- The motivation for user-located loop exits is simple.
- They fulfill a common need for goto statements through a highly restricted branch statement.
- The target of a goto can be many places in the program, both above and below the goto itself.

Q.83 Explain different looping statements used in 'C' language.

☞ [JNTU : Part B, Marks 5]

Ans : The iteration means repeated execution of statements based on some condition. In C the **for, while, do-while** statements are used for iterations.

1. for loop

The general form of for loop is

```
for(initialization;condition;step)
{
    ... //statements
}
```

2. While statement

The general form of while statement is

```
while(condition)
{
   ...//statements
}
```

3. do-while

The general form of do-while is

```
do
{
   ...//statements
}while(condition);
```

For example - Following code will store numbers 10,11,12,13,14,15 in an array and display them.

```
#include<stdio.h>
void main()
{
   int i;
   for(i=0;i<=5;i++)
   {
      a[i]=i+10;
      printf("%d",a[i]);
   }
}
```

2.28 Unconditional Branching

Q.84 Explain the unconditional statements with an example. **[JNTU : Part B, March-16, Marks 5]**

Ans. : • An unconditional branch statement transfers execution control to a specified place in the program

- The unconditional branch, or goto, is the most powerful statement for controlling the flow of execution of a program's statements.
- For example -

```
sum=0;
for(int i = 0; i<=10; i++)
{
   sum = sum+i;
   if(i==5)
{
goto addition;
   }
  }
  addition:
  printf("%d", sum);
```

- Java, Python, and Ruby do not have a goto. C# uses goto in switch statement.

Q.85 Explain the problems associated with unconditional Branching.

☞ **[JNTU : Part B, Nov.-15, Marks 5]**

Ans. : • The unconditional branch, or goto, is the most powerful statement for controlling the flow of execution of a program's statements.

- Without restrictions on use, imposed by either language design or programming standards, goto statements can make programs very difficult to read, and as a result, highly unreliable and costly to maintain.
- The gotos have ability to force any program statement to follow any other in execution sequence, regardless of whether the statement proceeds or follows previously executed statement in textual order.

2.29 Guarded Commands

Q.86 Explain about guarded commands.

☞ **[JNTU : Part A, Dec.-17, Marks 2]**

OR Discuss the guarded commands with an example.

☞ **[Part B, March-16, 17, Marks 5]**

Ans.: • New and quite different forms of selection and loop structures were suggested by Dijkstra.

- The main purpose of guarded commands is to support a new program design methodology that ensured correctness (verification) during development.
- **Basic idea :** if the order of evaluation is not important, the program should not specify one.
- Dijkstra's selection guarded command has the form
  - if <Boolean expr> -> <statement>
  - [] <Boolean expr> -> <statement>
  - ...
  - [] <Boolean expr> -> <statement>
  - fi
- **Semantics :** when construct is reached,
  - Evaluate all Boolean expressions.
  - If more than one are true, choose one non-deterministically.
  - If none are true, it is a runtime error.

UNIT - III

3 Subprograms, Implementation and ADT

Part I : Subprograms

3.1 Fundamentals of Subprograms

Q.1 What is subprogram ?

☞ [JNTU : Part A, Marks 2]

Ans. : Subprograms can be viewed as abstract operations on a predefined data set. A **subprogram definition** describes the interface to and the actions of the subprogram abstraction.

- A **subprogram call** is an explicit request that the subprogram be executed. For example - In Python the header of subprogram is as follows -
- **def** swap(parameters):
- There are two categories of sub-program - procedure and Functions.

Q.2 What are different categories of subprograms ?

☞ [JNTU : Part A, May-18 Marks 2]

Ans. : There are types of subprograms used – **procedure** and function

- **Procedure** – Procedure is set of commands executed in order.
- **Function** - Function is set of instructions for used for some computation.

Q.3 What are general subprogram characteristics ? Explain. ☞ [JNTU : Part B, May-18, Marks 5]

Ans. : Each subprogram has a single entry point

- The calling program is suspended during execution of the called subprogram
- Control always returns to the caller when the called subprogram's execution terminates

Q.4 Explain the terms - subprogram call, subprogram header and signature. ☞ [JNTU : Part A, Marks 3]

Ans. : The **subprogram call** is an explicit request that subprogram be executed.

- The **subprogram header** is first part of the definition, including the name, the kind of subprogram and the formal parameters.
- The **signature** is also called as parameter profile which is nothing but the number, order and types of its parameters.

Q.5 What are different elements of procedure ?

☞ [JNTU : Part B, Marks 5]

Ans. : Elements of procedure

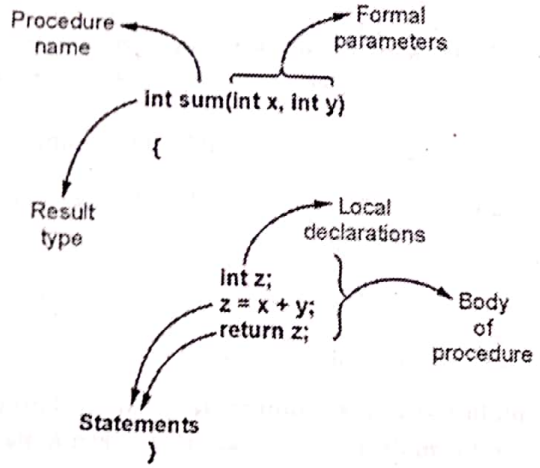
Various elements of procedure are -

1. Name for declaration of procedure
2. Body consisting of local declarations and statements
3. Formal parameters which are the placeholders of actuals
4. Optional result type.

For example

Consider following C code

```
# include <studio.h>
void main
{
    int a,b;
    int c;
    a = 10;
    b = 20;
    c - sum(a,b);
    printf('%d',c);
}
```



Q.6 What are the benefits of procedure ?

☞ [JNTU : Part B, Marks 5]

Ans. : Following are some benefits of using procedures

1. The implementation of certain piece of code can be **separated out** from the main implementation. Suitable names used for such procedure represents its purpose as well. For instance - If we call swap(a,b) function in the main implementation then it will indicate interchanging of the values a and b.
2. Writing a separate procedure allows to hide **implementation** details. The main procedure looks abstract with simply the call to the procedure.
3. Procedures can be used to partition a program into smaller modules. These **modules** are useful to maintain a large and complex code.
4. **Finding errors** from a large complex program becomes simplified when procedures or functions are written.
5. The **modifications** can be made in the progra without affecting rest of the programming code.
6. There are some standard procedures which are t part of **standard library**. For example - so function in C is used for finding the square ro of the number.

Q.7 Differentiate between function and procedure.

☞ [JNTU : Part A, Dec.-16, Marks 2

Ans. :

Sr.No.	Procedure	Function
1.	Procedure may or may not return a value.	Function returns a value.
2.	Procedure is set of commands executed in order.	Function is set of instructions for used for some computation.
3.	Function can be called from procedure.	Procedures can not be called from function.
4.	Example - Pascal, ADA are some programming languages that make use of procedures.	Example – C, C++, Java are some programming languages that make use of functions.

Example of Procedure in Pascal

```
Procedure MaxNum (x, y : integer; var z : integer)
  begin
  if      x > y then
          z: = x
  else
          z: =  y;
  end;
```

Example of Function in Pascal

```
Function MaxNum (x, y : integer) : integer; ( returnin
maximum between two numbers )
  var
     result  :    integer;
  begin
     if (x > y) then
  result : = x
     else
     result : = y;
  MaxNum : = result;
  end ;
```

3.2 Design Issues for Subprograms

Q.8 Describe the design issues of subprograms and operations.

☞ [JNTU : Part B, May-18, Marks 10]

Ans. : The design issues are as follows -

Are local variables static or dynamic ?

- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided ?
- Are parameter types checked ?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram ?
- Can subprograms be overloaded ?
- Can subprogram be generic ?
- If the language allows nested subprograms, are closures supported ?

3.3 Local and Referencing Environment

Q.9 Explain about local referencing environment.

☞ [JNTU : Part A, Dec.-17, Marks 2]

Ans. : Variables that are defined inside subprograms are called local variables. There are two types of local variables -

1. **Static local variable :** The static local variables are variables whose scope finishes when the subprogram within which it is defined gets terminated and they retain their value once they are initialized.
2. **Stack dynamic variable :** The stack dynamic local variables are kind of local variables which are bound to storage when the subprogram begins execution and are unbound from storage when the execution terminates. Stack-dynamic variables are allocated from the run-time stack.

Q.10 Explain stack dynamic local variables and static local variable with their advantages and disadvantages.

☞ [JNTU : Part B, Marks 5]

Ans. : 1) Stack dynamic local variable - Refer Q.9.

Advantages and Disadvantages -

Advantages

1) The stack dynamic local variables support for recursion.
2) The storage for locals is shared among some subprograms.

Disadvantages

1) The cost is required to allocate, initialize, and deallocate the variables for each call to subprogram.
2) Accesses to stack dynamic local variables must be indirect.
3) When all the local variables are stack dynamic, subprograms cannot be history sensitive. That means they can not retain data values of local variables between calls.

2) Static local variable - Refer Q.9

Advantages and disadvantages - Refer Q.11

Q.11 What are the advantages and disadvantages of static local variables ? ☞ [JNTU : Part B, Marks 5]

Ans. : Advantages :

1) The static local variables are efficient than stack-dynamic local variables as they require no run time overhead for allocation and deallocation.
2) It allows direct accessing.
3) They allow subprograms to be history sensitive.

Disadvantages :

1) They do not support for recursion.
2) The storage for locals cannot be shared among some programs.

Q.12 What is referencing environment ? Write its various components. ☞ [JNTU : Part B, Marks 5]

Ans. : Each program or subprogram has a set of identifiers that has associations for the use in referencing during its execution. Thus referencing environment is a complete set of bindings active at a certain point in a program.

- The **scope of binding** is the region of program or subprogram in the program's execution during which the binding is active.
- The referencing environment of a subprogram has following type of declarations.

1. **Local referencing environment :** This kind of referencing environment is denoted by the **set of identifiers** that are **created on the entry of the subprogram** and are used within the subprogram. The meaning of reference to the name can be found within the local environment of the subprogram definition. For example **local variables**, and formal parameters represent the set of identifiers that are associated with local referencing environment. This kind of referencing environment is also called as **local environment**.
2. **Nonlocal referencing environment :** This kind of referencing environment is denoted by the set of identifiers that may be used within the subprogram but **not created on the entry of the subprogram.**
3. **Global referencing environment :** This kind of referencing environment is denoted by the set of identifiers **created at the start of the execution of the main program.** The global environment is part of the non local environment. For example- any global variable, a structure variable declared globally etc.
4. **Predefined referencing environment :** There are some set of identifiers that **have predefined association** with the program and it is **defined directly in language definition.** Any program o subprogram may use these associations witho explicitly creating them. For example - predefine constants, built-in procedures etc.)

Q.13 Define shallow and deep binding for referencing environment of subprograms that have been passed as parameters ☞ [JNTU : Part A, Marks 2

Ans. :

1. The environment of the call statement that enacts th passed subprogram is called **"Shallow binding."**
2. The environment of the definition of the passed subprogram **"Deep binding."**

3.4 Parameter Passing Methods

Q.14 Explain the parameter passing in C. ☞ [JNTU : Part A, March-16, Marks 3

Ans. :

- Parameter passing is the mechanism used to pas parameters to a procedure (subroutine) or function.
- The most common methods are to pass th parameters are call by value and call by reference.
- When the value of the actual parameter is passe then it is called **call by value method.**
- When we pass the address of the memory locatio where the actual parameter is stored to the functio or procedure then it is called as **call by referenc method.**

Q.15 Distinguish between pass by value and pass b reference. ☞ [JNTU : Part B, Marks

Ans. :

Call By Value	Call By Reference
When a function is called, then in that function actual values of the parameters are passed.	When a function is called, then in that function addresses of the parameters are passed.
The parameters are simple variables.	The parameters are pointer variables.

If the values of the parameters get changed in the function then that change is not permanent. That means the values of the parameter get restored when the control returns back to the caller function.	If the values of the parameters get changed in the function then that change is permanent. That means the values of the parameter get changed when the control returns back to the caller function.

Example :
main()
{
x=10;
fun(x);
printf("%d",x);}
void fun(x)
{
x=15;
}
Output will be 10 and not 15
	Example
	main()
	{
	*x=10;
	fun(x);
	printf("%d",*x);
	}
	void fun(x)
	{
	x=15;
	}
	Output will be 15 and not 10

Compiler executes this type function slowly as values get copied to formal parameters.	Compiler executes this type of function faster as addresses get copied to the formal parameters.

Q.16 Give a detailed note on pass-by-name and pass-by-reference parameter passing methods.

☞ **[JNTU : Part B, March-17, Marks 5]**

Ans. : Refer Q.17.

Q.17 Explain the different parameter passing methods with an example.

☞ **[JNTU : Part B, Dec.-17, Marks 10]**

Ans. : There are two types of parameters.

i) Formal parameters

ii) Actual parameters

And based on these parameters there are various parameter passing methods, the most common methods are,

1. **Call by value :** This is the simplest method of parameter passing.
   - The actual parameters are evaluated and their r-values are passed to called procedure.
   - The operations on formal parameters do not changes the values of actual parameter.

   Example : Languages like C, C++ use actual parameter passing method. In PASCAL the non-var parameters.
2. **Call by reference :** This method is also called as **call by address** or **call by location.**
   - The L-value, the address of actual parameter is passed to the called routines activation record.
   - The values of actual parameters can be changed.
   - The actual parameters should have an L-value.

   Example : Reference parameters in C++, PASCAL'S var parameters.
3. **Copy restore :** This method is a hybrid between call by value and call by reference. This method is also known as **copy-in-copy-out or values result.**
   - The calling procedure calculates the value of actual parameter and it is then copied to activation record for the called procedure.
   - During execution of called procedure, the actual parameters value is not affected.
   - If the actual parameter has L-value then at return the value of formal parameter is copied to actual parameter.

   Example : In Ada this parameter passing method is used.
4. **Call by name :** This is less popular method of parameter passing.
   - Procedure is treated like macro. The procedure body is substituted for call in caller with actual parameters substituted for formals.
   - The actual parameters can be surrounded by parenthesis to preserve their integrity.

- The locals names of called procedure and names of calling procedure are distinct.

Example : ALGOL uses call by name method.

Q.18 Obtain the output for various parameter passing methods. Consider the following piece of code.

```
procedure exchange (m,n:integer);
var t : integer;
begin
t := m;
m := n;
n ;= t;
end;
...
i ;= 1
a[i] := 50 { a:array [1....10] of integer}
print (i,a[i]);
exchange{i,a[i]);
print (i,a[1]);
```

Ans. : The output for all the above discussed method is

Call by value		Call by reference		Copy restore		Call by name	
1	50	1	50	1	50	1	50
1	50	50	1	50	1	Error	

The error occurs because

T := i ∴ t := 1

I := a[i] ∴ I := 50 as a[i] = 50

a[i] := t ∴ a[50] = t = 1 then index is out of bounds.

Q.19 What is the output of the following C program, if the compiler uses dynamic scope ? Briefly justify your answer.

```
int r;
void write (void) {
printf(" %d", r);
}
void Display (void) {
int r = 37.24 ;
write ();
}
void main (void) {
r = 11.34
write ( ) ;
Display ( );
```

Ans. :

The output with dynamic scope will be

11.34 37.24

Initially r behaves as a global variable and hence val 11.34 will be printed. In the Display function variable has a local scope and its value gets changed to 37.2 And it is printed in Display function itself. Hence 11. 37.24 is the output.

Q.20 Write the output of the following C progra using following parameter passing methods.

i) Call by value.

ii) Call by reference.

iii) Call by value-result and

iv) Call by name.

```
#include<stdio.h>
int i = 0;
int j = 0;
void p(int x, int y)
{
x+ = 1;
i+ = 1;
y+ = 1;
}
void swap(int x, int y)
{
x = x+y;
```

```
y = x-y;
x = x-y;
}
main()
{
int a[2] = {1, 1};
int b[3] = {1, 2, 0};
p(a[i], a[i]);
printf("%d, %d\n", a[0], a[1]);
swap(j, a[j]);
printf("%d,%d, %d\n", b[0], b[1], b[2]);
return 0;
}
```

Ans. : i) Call by value

When P(a[i], a[i]) is called then

P (1,1)

↓ ↓

P (x, y)

In the function body P, x and y will be increment. Hence x = 2, i = 2, y = 2. But on returning to main, the x and y values will not be preserved.

Printf (" % d % d", a[0], a[1]) will result as **1 1**

When call to swap (j, a[j]) given it will be

↓ ↓

swap (x, y)

↓ ↓

swap (0, 1)

In the function body of x

i.e

$$x = x + y$$
$$x = 0 + 1 = 1$$
$$y = x - y$$
$$y = 1 - 1 = 0$$
$$x = x - y$$
$$x = 1 - 0$$
$$x = 1$$

$\therefore$ x = 1, y = 0. The x and y values get swapped. But on returning to main, the x and y values do not get preserved.

The printf statement is for printing b[0] and b[2] values. Therefore we will get **1 2 0.**

Hence output will be

1 1

1 2 0

ii) Call by reference

In call by reference, the formal parameters get associated with actual parameter.

```
P (a[i], a[i])
   ↓  ↙
P ( 1, 1 )
   ↓ ↓
P ( x, y )
```

Here a[0] is passed, as a reference.

```
{
    x + = 1    /* x becomes 2, y becomes 2 */
    i + = 1    /* i = 1 */
    y + = 1    /* y becomes 3 */
}
```

When control returns to main **a[0]** = x = y = 3 and **a[1]** remains as it is i.e. **1.**

After calling swap function the print f statement is printing b[0], b[1] and b[2] i.e. 1, 2, 0.

Hence output will be,

3 1

1 2 0

iii) Call by value result

In call by value result the actual parameter applied by the caller is copied into callee's formal parameter the function executes and then formal parameter is copied back to caller. No alias or reference is created between formal and actual parameters. Hence

a[0] x, a[0] y

x += 1

i.e. x = 1 + 1 = 2

y + = 1

i.e. y = 1 + 1 = 2

On returning to main value of x gets copied to a[0] a[0] = 2. The a[1] remains 1. After swap function the printf prints b[0], b[1] and b[2].

Hence output will be,

2 1

1 2 0

iv) Call by name

In this method the actual parameters will be substituted for all the occurrences of formal parameter.

Hence,

```
P ( int x, int y )          P(a[0], a[1] )
{                           {
      x + = 1                     a[0] = 1 + 1 = 2
      i + = 1     becomes         i + = 0 + 1 = 1
      y + = 1                     a[1] = 1 + 1 = 2
}                           }
```

On returning from P we get **2, 2** as an output of first printf.

After calling swap, the printf prints b[0], b[1] and b[2].

Hence output will be,

2 2

1 2 0

Q.21 Explain parameter passing methods for C, C++, Java, ADA, and C#.

☞ **[JNTU : Part B, Marks 5]**

Ans. : (1) C

- It supports pass by value method.
- The pass by reference is achieved by using pointers as parameter.

2) C++ :

- A special pointer type called reference type is used for pass-by-reference.

3) Java :

- All parameters are passed by value
- Object parameters are passed by reference

4) ADA :

- There are three semantics modes of parameter transmission: in, out, in out; in is the default mode.
- Formal parameters declared out can be assigned but not referenced; those declared in can be referenced but not assigned; in out parameters can be referenced and assigned

5) C# :

- Default method of parameter passing is pass by value.
- Pass-by-reference is specified by preceding both formal parameter and its actual parameter with ref.

Q.22 Explain the concept of type checki parameters.

☞ **[JNTU : Part B, Marks 5**

Ans. :

- It is neccesary for ensuring the reliability of softwar that the types of actual parameters be checked fo consistency with the types of the correspondin formal parameters.
- For example -

  result = fun(10)

- Here the actual parameter is an integer constant. I the formal parameter of fun is a floating-point type no error will be detected without parameter type checking.
- Early languages, such as Fortran 77 and the original version of C, did not require parameter type checking.
- In Pascal, Java, and Ada the type checking of parameters is always required
- Relatively new languages Perl, JavaScript, and PHP do not require type checking

3.5 Parameters that are Subprograms

Q.23 What are the two fundamental design considerations for parameter passing methods ?

☞ **[JNTU : Part A, March-17, Marks 2]**

Ans. : 1) The first design consideration is how much the method is efficient.

2) And second is whether it's a one-way or a two-way data transfer. But a safe and reliable programming should uses one way parameters as far as possible and to avoid using two ways parameters as long as it can. Also the most efficient way is to use pass-by-Reference which is a two-way passing method.

Q.24 Explain how subprogram names are passed as parameters. Illustrate with example.

☞ [JNTU : Part B, Nov-15, Dec.-16, Marks 5]

Ans. :

- Subprogram names can be passed as parameters to other subprogram.
- If only transmission of subprogram code is the only requirement, then it could be done by passing a single pointer.
- But this passing a subprogram as a parameter faces two problems –
  - First, there is the matter of type checking the parameters of the activations of the subprogram that was passed as a parameter. In C and C++, functions cannot be passed as parameters, but pointers to functions can. The type of a pointer to a function includes the function's protocol.
  - The second complication with parameters that are subprograms appears only with languages that allow nested subprograms. The issue is what referencing environment for executing the passed subprogram should be used.
- There are three choices:
  - The environment of the call statement that enacts the passed subprogram **(Shallow binding)**
  - The environment of the definition of the passed subprogram **(deep binding)**
  - The environment of the call statement that passed the subprogram as an actual parameter **(ad hoc binding)**
- For example –

```
function A() {
  var x;
  function B() {
    alert(x); // Creates a dialog box with the value
              // of x
  };
  function C() {
    var x;
    x = 30;
    D(B);
  };
  function D(FB) {
    var x;
    x = 40;
    FB();
  };
  x = 10;
  C();
};
```

- Consider the execution of B when it is called in D.
  - **For shallow binding,** the referencing environment of that execution is that of D, so the reference to x in B is bound to the local x in D, and the output of the program is 40.
  - **For deep binding,** the referencing environment of B's execution is that of A, so the reference to x in B is bound to the local x in A, and the output is 10.
  - **For ad hoc binding,** the binding is to the local x in C, and the output is 30.

Q.25 What are the design issues when subprogram names are passed as parameters ?

☞ [JNTU : Part A, Marks 2]

Ans. : The design issues are -

1. Are parameter types checked ?
2. What is the correct referencing environment for a subprogram that was sent as a parameter ?

3.6 Calling Subprograms Indirectly

Q.26 Explain the calling of subprograms indirectly.

☞ [JNTU : Part B, Marks 5]

Ans. :

- The calling of subprograms indirectly occurs in a situation when there are several possible subprograms to be called and the correct one on a particular run of the program is not known until execution particularly in event handling.

- In C/C++ such calls are made through function pointers.
- In C#, method pointers are implemented as objects called delegates
  - A delegate declaration:

public delegate int display(int x);

- This delegate type, named display, can be instantiated with any method that takes an int parameter and returns an int value.

Now consider a method:

```
static int fun(int x)
{
//function body
}
```

- We can create an instance of **display** method by passing the **fun** as parameter.

display displayfun = **new** display(fun);

- The call to the instance **displayfun** can be made as follows –

displayfun(100);

- A delegate can store more than one address, which is called a **multicast delegate.**
- C and C++ allow a program to define a **pointer to function,** through which the function can be called.

3.7 Overloaded Subprograms

Q.27 What is an overloaded subprogram ? Give an example.

☞ [JNTU : Part A, March-17, Marks 3, Part B, Dec.-17, Marks 5]

OR Explain how subprogram is overloaded ? Give examples. ☞[JNTU : Part B, Nov-15, March-16, Marks 5]

Ans. :

- **Definition :** Overloading is a mechanism in which we can use many methods having the same function name but can pass different number of parameters or different types of parameter. For example :

```
int sum(int a,int b);
double sum(double a,double b);
int sum(int a,int b,int c);
```

- That means, by overloading mechanism, we ca handle different number of parameters or differen types of parameter by having the same method name
- Overloaded subprograms provide **adho polymorphism.**
- Subtype polymorphism means that a variable of type T can access any object of type T or any type derive from T (OOP languages).
- A subprogram that takes a generic parameter that is used in a type expression that describes the type o the parameters of the subprogram provides parametric polymorphism
- **Example Program in Java**

```
public class OverloadingDemo
{
 public static void main(String args[])
 {
  System.out.println("Sum of two integers    ");
  Sum(10,20); <---------- line A
  System.out.println("Sum of two double numbers    ");
  Sum(10.5,20.4); <---------- line B
  System.out.println("Sum of three integers    ");
  Sum(10,20,30); <---------- line C
 }
 public static void Sum(int num1,int num2)
 {
  int ans;
  ans=num1+num2;
  System.out.println(ans);
 }
 public static void Sum(double num1,double num2)
 {
  double ans;
  ans=num1+num2;
  System.out.println(ans);
 }
 public static void Sum(int num1,int num2,int num3)
 {
```

```
    int ans;
    ans=num1+num2+num3;
    System.out.println(ans);
  }
}
```

Output

```
F:\test>javac OverloadingDemo.java
F:\test>java OverloadingDemo
Sum of two integers
30
Sum of two double numbers
30.9
Sum of three integers
60
```

Program Explanation :

- In above program, we have used three different methods possessing the same name. Note that,

on the line A :

- We have invoked a method to which two integer parameters are passed. Then compiler automatically selects the definition **public static void Sum(int num1, int num2)** to fulfil the call. Hence we get the output as 30 which is actually the addition of 10 and 20.

on line B :

- We have invoked a method to which two double parameters are passed. Then compiler automatically selects the definition **public static void Sum(double num1,double num2)** to fulfil the call. Hence we get the output as 30.9 which is actually the addition of 10.5 and 20.4.

on line C :

- We have invoked a method to which the three integer parameters are passed. Then compiler automatically selects the definition **public static void Sum(int num1,int num2,int num3)** to fulfil the call. Hence we get the output as 60 which is actually the addition of 10, 20 and 30.

3.8 Generic Subprograms

Q.28 Explain about generic sub-programs with examples.

☞ [JNTU : Part B, March-17, Marks 5]

Ans. : There are situations in which the similar subprograms are written several times because of some minor differences in parameters. For example if we want to perform addition of two integers and addition of two double values then we write following routines -

```
int add_int(int a, int b)
{
    int c;
    c=a+b;
    return c;
}
double add_double(double a, double b)
{
    double c;
    c=a+b;
    return c;
}
```

- Note that the instructions in both the subprograms are very much similar but due to change in data types we need to write two different routines. To avoid these efforts, programming languages offer the concept of **generic subprogram.**
- **Definition :** Generic subprogram is a subprogram which implements the desired algorithm using general data type.
- In C++, the concept of generic routine is achieved by **using templates.**
- **For example** – In C++ the above code can be implemented using generic routine as follows

```
template <class T>
T add(T a, T b)
{
    T c;
    c=a+b;
    return c;
}
```

Q.29 If a Java 5.0 generic method is called with three different generic parameters, how many versions of the method will be generated by the complier ?

Ans. : Although Java generic methods can be instantiated with any number of times, only **one copy** of code is built.

- The internal version of generic method is called a raw **method**. It operates on **object** class objects.
- At the point where the generic value of generic method is returned the compiler inserts a typecast to the proper type.

Q.30 What are generic subprograms in Ada with an example ?

☞ **[JNTU : Part B, March-16, Marks 5]**

Ans. : In Ada a program unit (either a subprogram or a package) can be a generic unit.

- This generic unit is used to create instances of the code that work with actual data types.
- The data type required is passed in as a parameter when the generic unit is instantiated.
- Generics are usually presented in two parts, the generic specification and then the generic package.
- **Example :** The following is a compilation unit. It includes the keyword generic, a list of generic parameters and the procedure specification.

```
generic :
type element is private;  -- private here means
                          -- element is a parameter to
                          -- the generic sub program
procedure exchange(a,b :in out element);
```

- The body of the generic procedure is presented as a seperate compilation unit. It is as follows

```
procedure exchange(a,b :in out element) is
    temp  :element;
begin
    a:=temp;
    a:=b;
    b:=temp;
end exchange;
```

- The code above is simply a template for an actu procedure that can be created. It can't be called. It equivalent to a type statement - it doesn't alloc any space, it just defines a template We can actua create a procedure as follows -

```
procedure swap is new exchange(integer);
```

or

```
procedure swap is new exchange(character);
```

3.9 Design Issues for Functions

Q.31 Explain the design issues for functions.

☞ **[JNTU : Part A, Dec.-17, Marks 3**

OR Explain the design issues that are involved i functions.

☞ **[JNTU : Part B, Nov-15, Marks 5**

Ans. : Following are two important design issues tha are used for function

1) Are side effects allowed ?

- Parameters to the function must be in-mode forma parameters. For example ADA is a language that ha in-mode formal parameters to avoid the function causing the side effects through its parameters.

2) What type of return values are allowed ?

- Most imperative languages restrict the return types
- C allows any type except arrays and functions
- C++ is like C but also allows user-defined types
- Ada subprograms can return any type
- Java and C# methods can return any type.

Q.32 Write an example of call and return statements

Ans. :

```
#include<stdio.h>
void main()
{
        a=10;
        b=20;
        add(a,b);//call to the function
}
        int add(int a,int b)
{
        return a+b; //return from the function
}
```

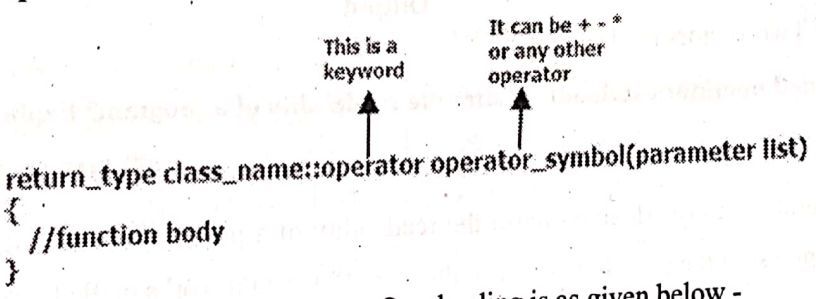
3.10 User Defined Overloaded Operators

Q.33 Discuss user defined overloaded operators ☞ **[JNTU : Part B, Dec.-16, Marks 5]**

Ans. : There are predefined operators such as +, – ,*, / and so on which operate on the fundamental data types such as integer, double, char and so on.

- In order to make the user defined data type as natural as fundamental data type, the user defined data types can be associated with the set of predefined operators the concept called **operator overloading** is used.
- For instance - If we want to perform the operations on two complex numbers then with the help of operator overloading the operations such as addition, multiplication and so on can be carried out. In this case the class for Complex number is created.
- **Definition of Operator Overloading :** Operator overloading is a mechanism in which the existing operator is used to give user defined meaning to it without changing the basic operation.
- Languages like C++, ADA, Python, Ruby use the concept of operator overloading.

Syntax of overloading an operator



- **Example :** C++ Program demonstrating Operator Overloading is as given below -

```
#include <iostream>
using namespace std;
class vector {
    public:
        int p,q;
        vector() {p=0;q=0;}// constructor without parameters
        vector (int,int);//constructor with parameters
        vector operator + (vector);//definition of operator +
};
vector::vector (int a, int b) {
    p = a;
    q = b;
}
vector vector::operator+ (vector obj)
```

```
{
    vector temp;
    temp.p = p + obj.p;
    temp.q = q + obj.q;
    return (temp);
}
int main ()
{
    vector a (10,20);
    vector b (1,2);
    vector c;
    c = a + b;
    cout<<"\n The Addition of Two vectors is...";
    cout << c.p <<" and "<<c.q;
    retrun 0;
}
```

Output

The Addition of Two vectors is...11 and 22

Q.34 How can user defined operator overloading harm the readability of a program? Explain

☞ [JNTU : Part B, Dec.-16 Marks 7]

Ans. : User-defined operator can overloading harm the readability of a program because the built in operator has t precision and compiler knows all the precision between the operators, and it works on that precision.

- User can also create its own operator but the compiler does not come to know how to make precision of this operato
- This will be the cause of the overloading harm.

3.11 Closures

Q.35 Explain the concept of closures.

☞ [JNTU : Part A, Marks 3]

Ans. : A closure is a subprogram and the referencing environment where it was defined.

- The referencing environment is needed if the subprogram can be called from any arbitrary place in the program.
- A static-scoped language that does not permit nested subprograms doesn't need closures.
- Closures are only needed if a subprogram can access variables in nesting scopes and it can be called from anywhere.
- Following is an example of a closure written in JavaScript:

```
function Multiply(x) {
    return function(y) {
```

```
        return x * y;
        }
    }
    ...
    var mulBy = Multiply(10);
    document.write("10*20 = " + mulBy(20) +
    "<br/>");
```

- The closure is the anonymous function returned by 'Multiply'
- The output of this code, assuming it was embedded in an HTML document and displayed with a browser, is as follows :

```
10*20 = 200
```

3.12 Co-routines

Q.36 What is Co-routines? Explain

☞ [JNTU : Part A, Nov-15, Marks 2]

OR What are the characteristics of co-routine feature? List the languages which allow co-routines.

☞ [JNTU : Part B, Dec.-16,17, Marks 5]

Ans. : A coroutine is a subprogram that has multiple entries and controls them itself. It is also called as symmetric control because caller and called coroutines are on a more equal basis.

- Coroutines have means to maintain their status between activations. They have history sensitive and static local variables.
- A coroutine call is named as **resume** and not a call.
- Only one co-routine is in execution at a time.
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine.
- The execution of co-routine is interleaved and not overlapped. Even though there is single processor in the system, all the executing programs in such a system appear to run concurrently by sharing the processor. It is sometimes called as **quasi-concurrency.**

Part II : Implementing Subprograms

3.13 General Semantics of Calls and Returns

Q.37 What is subprogram linkage ?

☞ [JNTU : Part A, Marks 2]

Ans. : The subprogram call and return operations of a language are together called its subprogram linkage.

- Any implementation method for subprograms must be based on the semantics of the subprogram linkage.

Q.38 Enlist the actions associated with subprogram call.

☞ [JNTU : Part A, Marks 3]

Ans. : A subprogram call has numerous actions associated with it-

- Parameter passing methods
- Static local variables
- Execution status of calling program
- Transfer of control
- Subprogram nesting

3.14 Implementing Simple Subprograms

Q.39 What are the call and return semantics ? Explain.

☞ [JNTU : Part B, Marks 5]

Ans. : Call Semantics:

- Save the execution status of the caller
- Pass the parameters
- Pass the return address to the callee
- Transfer control to the callee

Return Semantics :

- If pass-by-value-result or out mode parameters are used, move the current values of those parameters to their corresponding actual parameters (arguments)
- If it is a function, move the functional value to a place the caller can get it
- Restore the execution status of the caller
- Transfer control back to the caller

Q.40 What are the two separate parts required for implementing simple subprogram ? ☞ **[JNTU : Part A, Marks 2]**

Ans. : The two separate parts are - (1) The actual code and (2) the non code part which includes local variables and data that can change.

Q.41 What is the need of activation record in implementing subprogram ? Explain with an example. ☞ **[JNTU : Part B, March-17 Marks 5]**

Ans. : Definition : The activation record is a block of memory used for managing information needed by a single execution of a procedure.

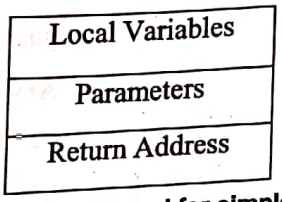
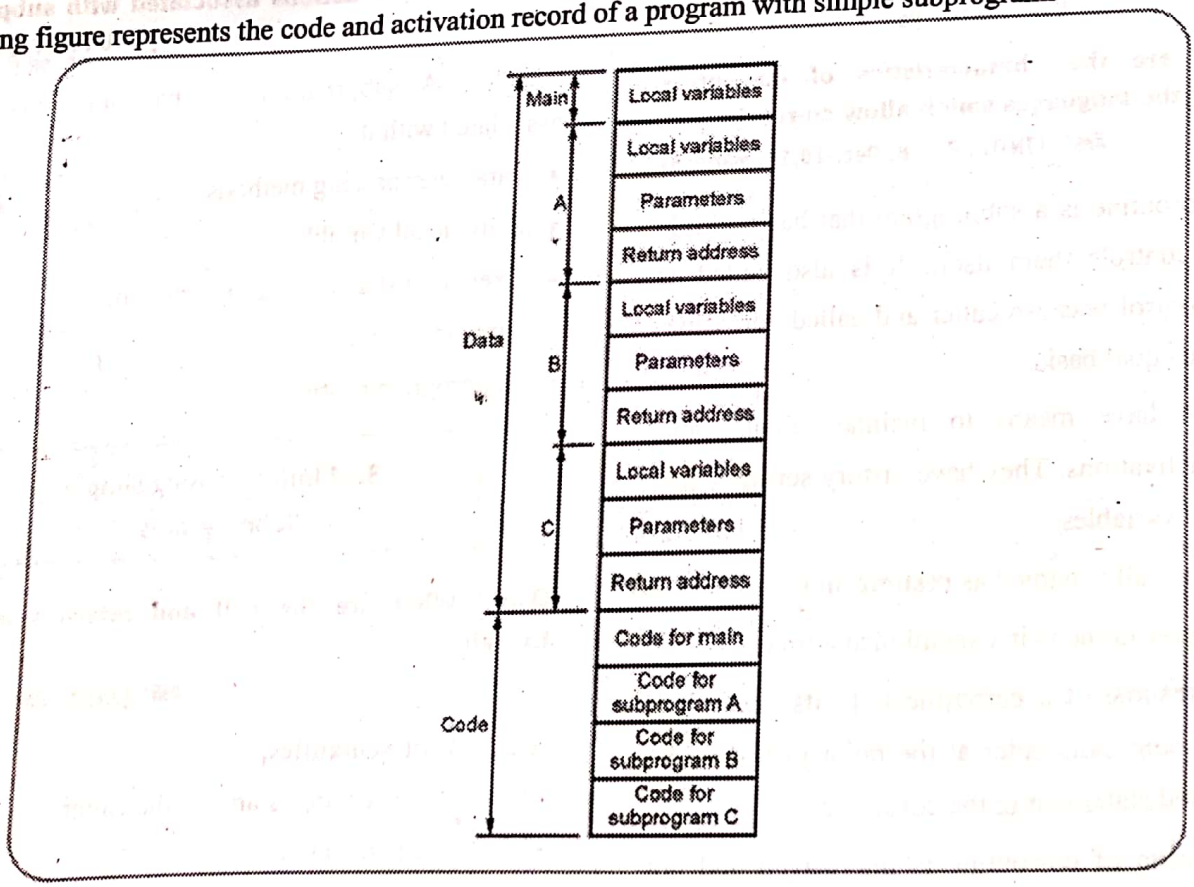


Fig. Q. 41.1 Activation record for simple subprogram

Following figure represents the code and activation record of a program with simple subprograms

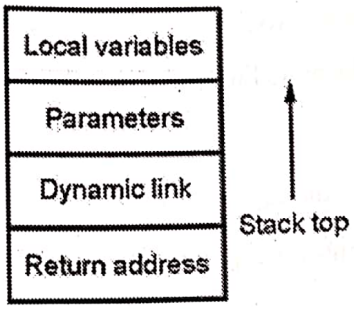


- The code can be attached to ARIs Also, the four program units could be compiled at different times. Linker put the compiled parts together when it is called for the main program.

3.15 Implementing Subprograms with Stack Dynamic Local Variables

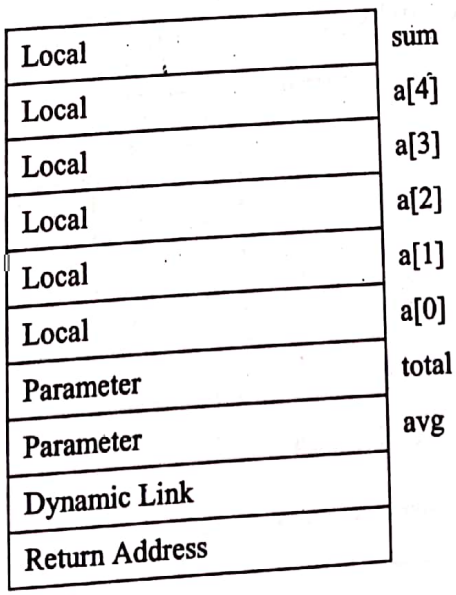
Q.42 Explain the implementing subprograms with stack dynamic local variables. ☞ **[JNTU : Part B, Marks 5]**

Ans. : The activation record is maintained in implementing the subprogram with stack dynamic local variables. The structure of activation record for a language with stack dynamic local variables is as follows –



- The activation record format is static, but its size may be dynamic.
- The dynamic link points to the top of an instance of the activation record of the caller
- An activation record instance is dynamically created when a subprogram is called.
- Activation record instances reside on the run-time stack.
- The Environment Pointer (EP) must be maintained by the run-time system. It always points at the base of the activation record instance of the currently executing program unit.
- For example – Consider following code in C

```
void add(int total, float avg)
{
 int a[5];
int sum;
 ...
}
```



The caller actions are as follows:

1. Create an activation record instance.
2. Save the execution status of the current program unit.
3. Compute and pass arguments.
4. Record the return address for the callee.
5. Transfer control to the callee.

The prologue actions of the callee are as follows:

1. Save the old EP in the stack as the dynamic link and create the new value.
2. Allocate local variables.

The epilogue actions of the callee are as follows:

1. If there are pass-by-value-result or out-mode parameters, the current values of those parameters are move
to the corresponding actual arguments.
2. If the subprogram is a function, the functional value is moved to a place accessible to the caller.
3. Restore the stack pointer by setting it to the value of current EP minus one and set the EP to the old dynami
link.
4. Restore the execution status of the caller.
5. Transfer control back to the caller.

Q.43 Write a program in C for finding factorial of a given number using recursion. Also represent various instan of activation records during the execution and after the execution of recursive call. ☞[JNTU : Part B, Marks

Ans. : The C code for implementing the factorial function using recursion is as given below. Note that the position the program at which the instances of activation records are obtained are represented by numbering them.

```
int factorial(int n)
{ <----------------------------------------------1
    if (n<=1) return 1; else return n*factorial(n-1);
    <----------------------------------------------2
}
int main()
{
    int value;
    value = factorial(3);
    <----------------------------------------------3
return 0;
}
```

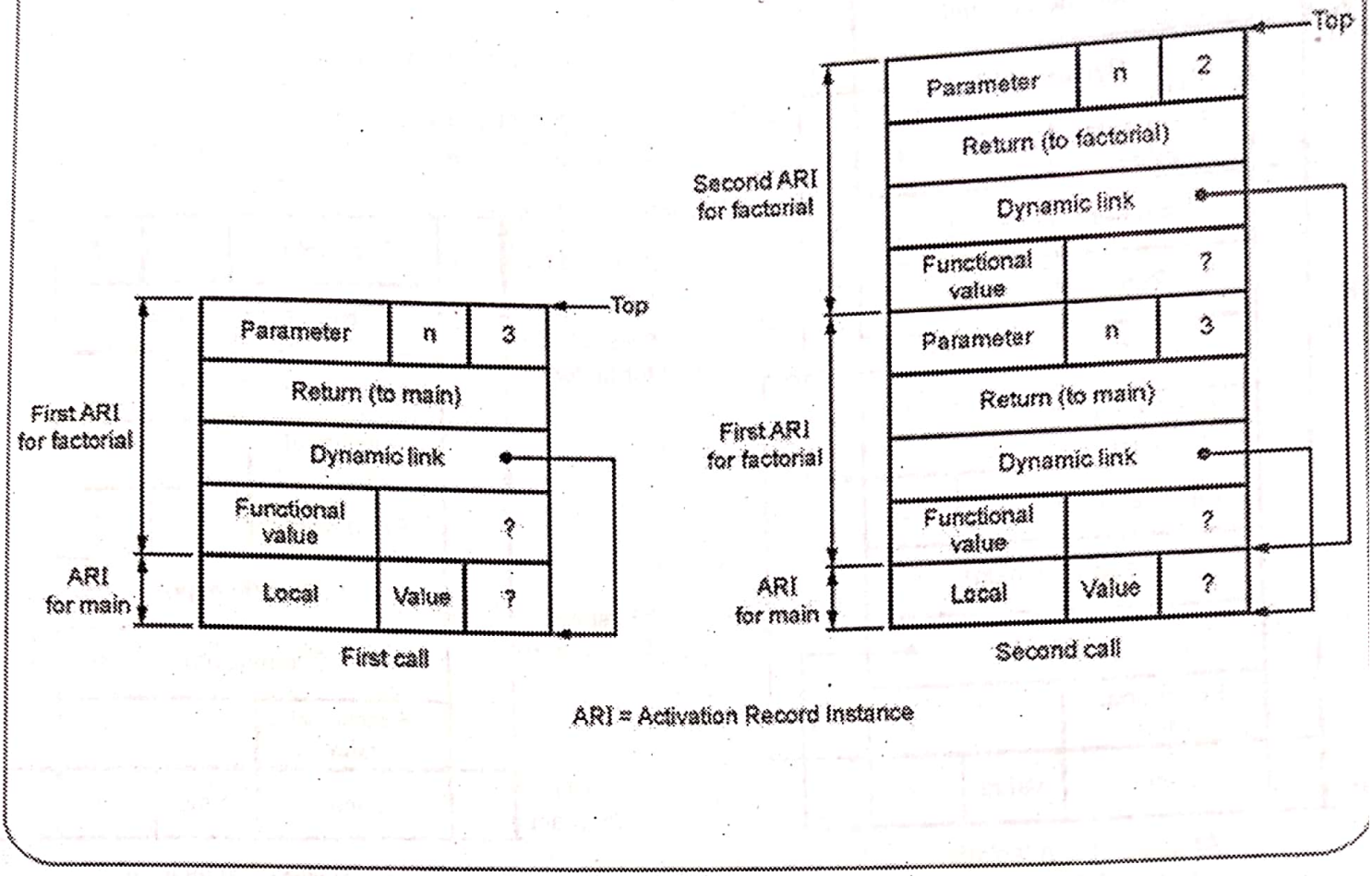
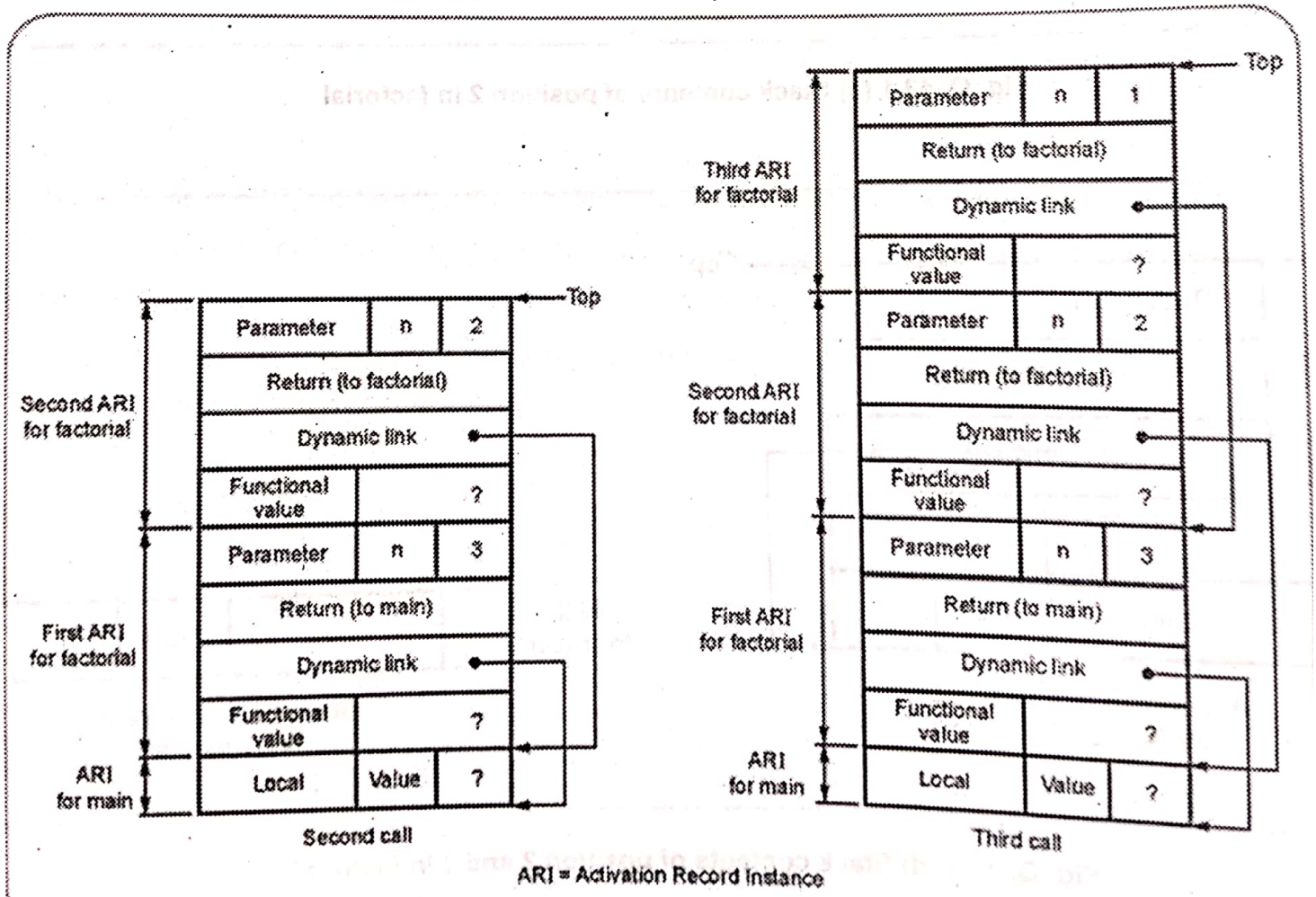


Fig. Q. 43.1 (a) Stack contents at position 1 in factorial



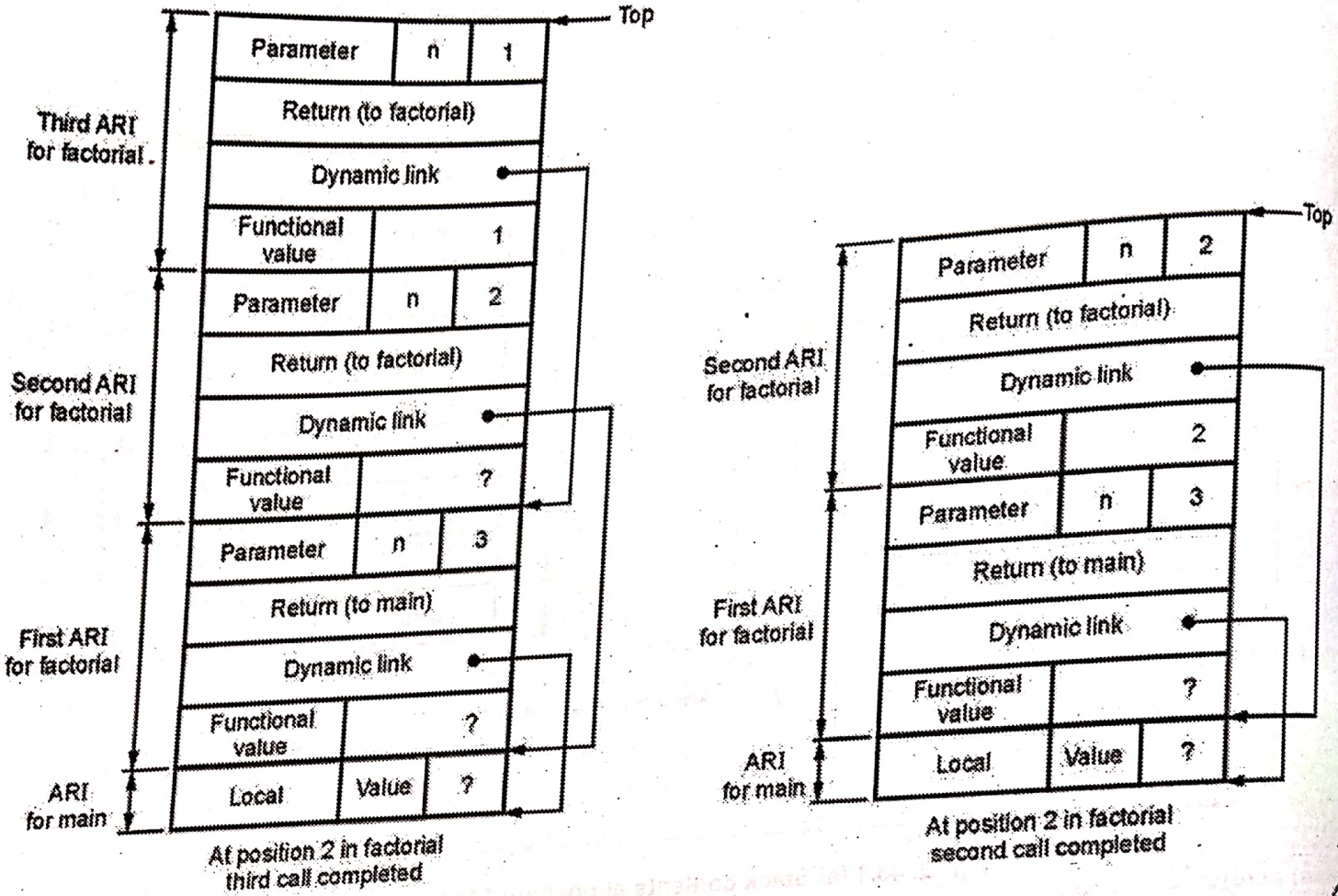


Fig. Q. 43.1 (c) Stack contents of position 2 in factorial

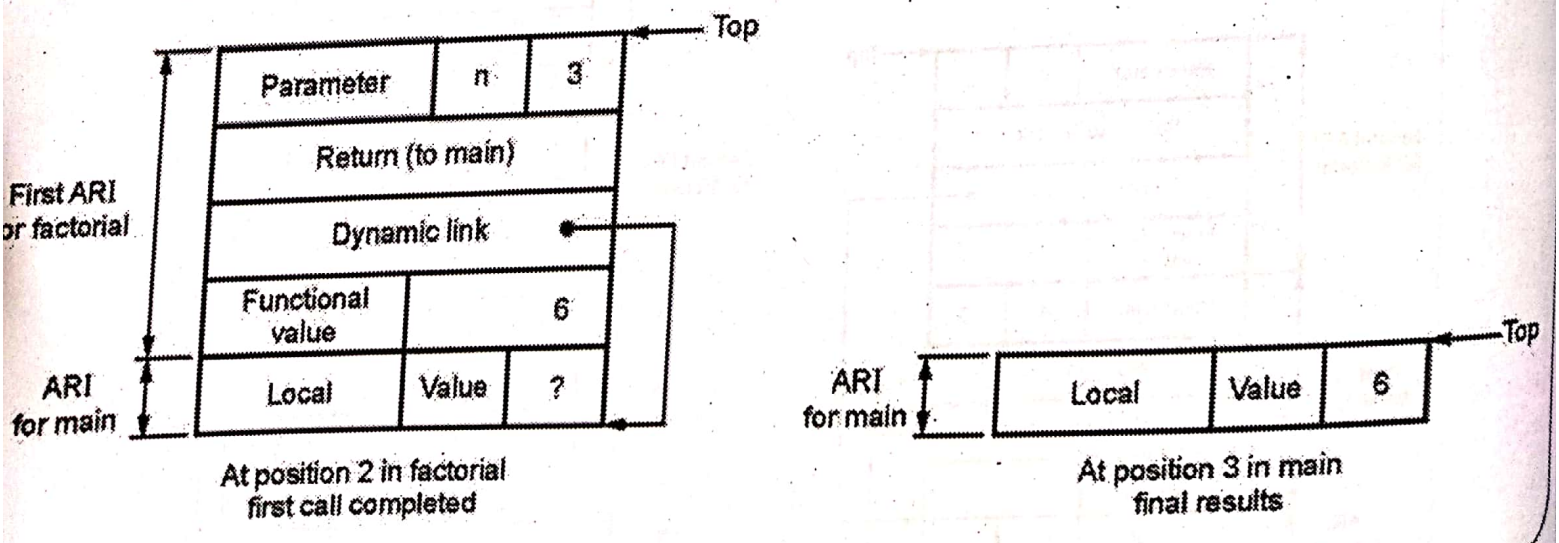


Fig. Q. 43.1 (d) Stack contents of position 2 and 3 in factorial

3.16 Nested Subprograms

44 Write short note on - nested subprograms.

ns. : Some non-C-based static-scoped languages such as Fortran 95, Ada, Python, JavaScript, and so on use stack-namic local variables and allow subprograms to be nested. ☞ [JNTU : Part B, Marks 5]

- All variables that can be non-locally accessed reside in some activation record instance in the stack.
- The process of locating a non-local reference is as follows -
  1. Find the correct activation record instance
  2. Determine the correct offset within that activation record instance
- Locating a Non-local Reference
  - Finding the offset is easy
  - Finding the correct activation record instance
- Static semantic rules guarantee that all non-local variables that can be referenced have been allocated in some activation record instance that is on the stack when the reference is made.

3.17 Blocks

45 What is block ? Explain the two methods of implementing block.

s. : Block is a specific group of instructions with user specified scope for variables. For example – ☞ [JNTU : Part B, Marks 5]

```
{
    int x;
    x=a[0];
    a[1]=x;
}
```

The lifetime of variable x begins when control enters the block. There are two methods of implementing block-

(1) Treat blocks as parameter-less subprograms that are always called from the same location - Every block has an activation record; an instance is created every time the block is executed

(2) Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record.

The variable storage is as follows -

```
int main()
{
 int a,b,c;
 while(...) {
    p,q,r;
    ...
    while(...) {
        i,j;
}
 while(...){
    x,y;
}
.....
    return 0 ;
}
```

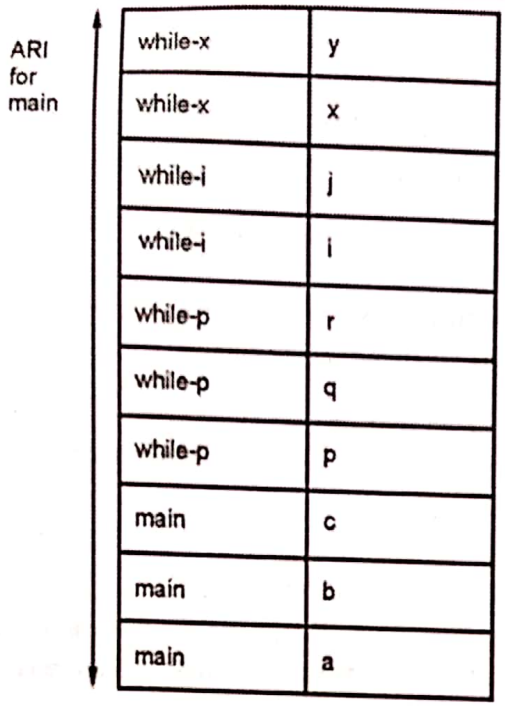


Fig. Q. 45.1 Block variable storage

3.18 Implementing Dynamic Scoping

Q.46 Briefly explain the static and dynamic scope.

☞ [JNTU : Part B, May-18, Marks 5]

Ans. : The variable is bound to its scope statically or dynamically.

- The **static scope** is in terms of lexical structure of a program. That means - the scope of variable is obtained by examining the complete source program without executing it. For example C program makes use of static scope.
- The **dynamic scope** is in terms of program execution. That means- the scope of variable can be determined during the execution of the program. For example - LISP, SNOBOL4 languages make use of dynamic scoping.

Q.47 What is the scope of a loop parameter in ADA ? Compare it with static and dynamic scope.

☞ [JNTU : Part B, Marks 5]

Ans. : The scope of loop parameter in ADA is **static scope**. For example - consider following procedure in ADA

```
procedure MainProg is
 a:Integer;
 procedure p1 is
     a:Integer;
     begin   //procedure p1 begins
     ...
     end;    //procedure p1 ends
 procedure p2 is
     a:Integer;
```

```
    begin   //procedure p2 begins
    ...a...
    ...
    end;    //procedure p2 ends
begin  //beginning of procedure MainProg
...
end  //procedure MainProg ends
```

- In above procedure, the **a declared in MainProg** can be accessed in procedure **p2** by the reference as **MainProg.a** Thus ADA uses static scope.
- **Comparison of between Static and Dynamic Scope**

Sr. No.	Static Scope	Dynamic Scope
1.	With static scope, a variable always refers to its top-level environment.	With dynamic scope, a variable always refer to identifier associated with the most recent environment.
2.	All scoping can be determined at compile time.	All scoping can be determined at run time or execution time.
3.	Static scope rules are usually encountered in compiled languages.	Dynamic scope rules are usually encountered in interpreted languages.
4.	Most languages, including Algol, Ada, C, Pascal, Scheme, and Haskell, are statically scoped.	APL, Snobol, early dialects of Lisp, and Perl have dynamic scoping.

- **Example code for illustrating static and Dynamic scope**

Consider following code -

```
#include<stdio.h>
int a=1;
int fun1()
{
    return a;
}
int fun2()
{
    int a=2;
    return fun1();
}
int main()
{
    printf("%d",fun2());
    return 0;
}
```

By static scope the output will be 1

By dynamic scope the output will be 2

Q.48 Explain the concept of static chain in detail.

☞ [JNTU : Part B, Marks 5]

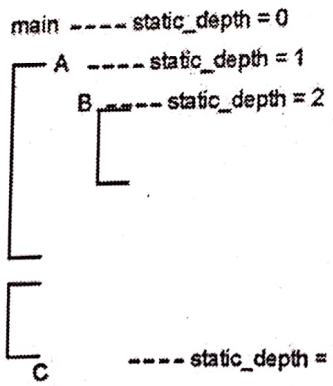
Ans. : A **static chain** is a chain of static links that connects certain activation record instances in the stack.

- The **static link**, static scope pointer, in an activation record instance for subprogram A points to one of the activation record instances of A's static parent.
- The static link appears in the activation record below the parameters.
- The static chain from an activation record instance connects it to all of its static ancestors.
- During the execution of a procedure P, the static link of its activation record instance points to an activation of P's static program unit.
- That instance's static link points, in turn, to P's static grandparent program unit's activation record instance, if there is one.
- So the static chain links all the static ancestors of an executing subprogram, in order of static parent first.
- This chain can obviously be used to implement the access to non-local vars in static-scoped languages.

Q.49 Explain the term static depth with the help of suitable example.

☞ [JNTU : Part A, Marks 3]

Ans. : A static_depth is an integer associated with a static scope whose value is the depth of nesting of that scope.



Q.50 What are the two methods of dynamic scoping ?

☞ [JNTU : Part A, Marks 2]

Ans. : The two methods of implementing non local accessing under dynamic scoping -

1) Deep Access :

- In this method, the stack of active variables is maintained.
- Nonlocal references are found by searching the activation record instances on the dynamic chain
- Length of chain cannot be statically determined
- Every activation record instance must have variable names.

2) Shallow Access :

- In this method, there exists a central storage and one slot is allotted for every variable name.
- If the names are not created at runtime then the storage layout can be fixed at compile time. Otherwise, when new activation procedure occurs, then that procedure changes the storage entries for its local at entry and exit.

Q.51 Compare the deep and shallow access methods.

☞ [JNTU : Part A, Marks 2]

Ans. :

Sr.No.	Deep access	Shallow access
1.	It requires a symbol table at run time.	It has a overhead of handling procedure entry and exit.
2.	It takes longer time to access the non locals.	It provides the fast access.

Q.52 Describe the shallow access method of implementing dynamic scoping.

☞ [JNTU : Part B, March-17, Marks 5]

Ans. : In this method, there exists a central storage and one stack is allotted for every variable name.

- In the shallow-access method, variables declared in subprograms are not stored in the activation records of those subprograms.
- If the names are not created at runtime then the storage layout can be fixed at compile time.
- There exists a central table with an entry for each variable name.

- **For example** - Consider following code skeleton –

```
void f3() {
    int x, y;
    x = a + b;
    ...
}
void f2() {
    int c, x;
    ...
}
void f1() {
    int b, c;
    ...
}
    void main() {
        int b, a;
        ...
}
```

Suppose the following sequence of function calls occurs:

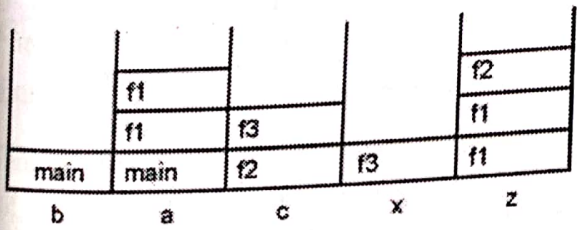
main calls f1

f1 calls f1

f1 calls f2

f2 calls f3

Following figure shows the variable stacks when using shallow access method of implementation -



Part III : Abstract Data Types

3.19 The Concept of Abstraction

Q.53 Explain the concept of abstraction and encapsulation with suitable example.

☞ **[JNTU : Part B, Marks 5]**

Ans. : Abstraction :

- An abstraction is a view or representation of an entity that includes only the most significant attributes.
- It is fundamental aspect of programming.
- Almost all the languages support process of abstraction with subprograms.
- Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.
- In Java, abstraction is achieved using **Abstract classes and interfaces.**

Encapsulation :

- Encapsulation is a mechanism for wrapping the data (variables) and code acting on the data (methods) together as a single unit.
- The logical module of C++ is class. The class defines the encapsulated unit. It encapsulates the data members and the operations that can be performed on these data members.

For example

```
class Test
{
private:
int a,b,c;
public:
void add();
void display();
};
```

- Thus encapsulation approach is useful in two ways -
  - It binds the data members with operations or methods.
  - It imposes the level of abstraction in a program.

3.20 Introduction to Data Abstraction

Q.54 What is data abstraction ?

☞ [JNTU : Part A, Marks 3]

Ans. : The concept of data abstraction is represented by means of **abstract data type.**

- An abstract data type is a user-defined data type that satisfies the following two conditions :

1) The representation of objects of the type is hidden from the program units that use these objects.
2) The declarations of the type and the protocols of the operations on objects of the type are contained in a single syntactic unit. Other program units are allowed to create variables of the defined type.

Q.55 What is Abstract Data Type (ADT) ?

☞ [JNTU : Part A, Marks 3]

Ans. : Refer Q.54.

Q.56 What are the benefits of data abstraction ?

☞ [JNTU : Part A, March-16, Marks 2]

Ans. : 1) It allows organization of data and corresponding operations in a single unit which can be considered as a data structure together.

2) Modifications can be made easily without affecting rest of the code.

3) By hiding the data representations, user code cannot directly accessible. This helps in making the programs reliable.

Q.57 Give the two kind of abstractions in programming languages.

☞ [JNTU : Part A, March-17, Marks 5]

Ans. : Two types of abstractions in programming language are - process abstraction and data abstraction.

Q.58 Write short note on data and procedural abstraction.

☞ [JNTU : Part B, Dec.-16, Marks 2]

Ans. : Data Abstraction - Refer Q. 54.

Procedural Abstraction - It specifies what a procedure does and ignores how it does. Following are advantages of procedural abstraction –

(1) Locality: Programmers don't need to know the implementation details.

(2) Modifiability: Replacing of one code does not affect another code.

(3) Language Independence : Due to procedural abstraction implementation could be done in any programming language.

Procedural abstractions are normally characterized in a programming language as "function/sub-function" or "procedure" abstraction.

Q.59 Differentiate between procedural and data abstraction.

☞ [JNTU : Part A, Nov-15, Marks 3]

Ans. :

Data abstraction	Procedure abstraction
The focus is primarily on data and then on procedure for abstraction.	The focus is only on procedures for abstraction.
Data abstraction is characterized as a data structure unit.	Procedural abstractions are normally characterized in a programming language as "function/sub-function" or "procedure" abstraction.
The advantage of data abstraction over procedural abstraction is that the data and the associated operations get specified together and hence it is easy to modify the code when data changes.	Due to procedural abstraction implementation could be done in any programming language.

3.21 Design Issues

Q.60 What are the design issues for the abstract data types ? ☞ [JNTU : Part A, Marks 3]

Ans. : Following are the design issues of abstract data types -

(1) What is the form of the container for the interface to the type ?

(2) Can abstract types be parameterized ?

(3) What access controls are provided ?

3.22 Language Examples

Q.61 Write a note on Abstract Data Types in Ruby. ☞ [JNTU : Part B, March-17, Marks 5]

Ans. : In Ruby the encapsulation construct is the **class**

- Local variables have "normal" names.
- Instance variable names begin with "at" signs (@)
- Class variable names begin with two "at" signs (@@)
- Instance methods have the syntax of Ruby functions (def ... end)
- Constructors are named initialize,implicitly called when new is called.
- If more constructors are needed, they must have different names and they must explicitly call new.
- Class members can be marked private or public, with public being the default.
- Classes are dynamic.
- For example –

```
class Item ← Class definition
 def initialize(price) ← Constructor
  @price = price ← Instance variable
 end

 def price
  @price
 end
 def compare_price(c) ← Instance Method
 if c.price > price
  "Second item's price is bigger."
 else
  "Second item's price is equal or less."
 end
end
protected :price
end

item1 = Item.new(55) ← Using new call to the constructor
item2 = Item.new(34)
puts item1.compare_price(item2) ← Call to the function
puts item1.price
```

Q.62 Write a note on Abstract Data Types in ADA. ☞ [JNTU : Part B, Marks 7]

Ans. : Definition of Encapsulation : Encapsulation is a mechanism by which the program components can be combined together to provide some specific functionality.

- Encapsulation supports for **information hiding** of implementation of the modules.
- Encapsulation is described by two parts - **specification and implementation.**
  - The **specification** describes how the services provided by the module can be accessed by clients.
  - The **implementation** describes the coding and logic building of the module that provides specific functionality.
  - Example of package specification and implementation in ADA is as given below :

Step 1 : Create a file containing following code and save it using **.ads** extension.

```
package MyExample is
   function Fun(x : in Float) return Float;
end MyExample;
```

Package Specification

Step 2 : Create another file containing following code and save it using **.adb** extension.

```
package body MyExample is
   function Fun(x : in Float) return Float is
   begin
    return x * 10.00;
   end Fun;
      end MyExample;
```

Package Body

Step 3 : To access the function in the main file, here is an example :

```
with Ada.Text_IO;
use Ada.Text_IO;
with MyExample;
use MyExample;

procedure Exercise is
  number : Float;
  result : Float;

begin
  number := 48.36;
  result := Fun(number);

  put_line(Float'Image(number) & " * 10 = " & Float'Image(result));
end Exercise;
```

The output of above code will be 10 times of the value 48.36

Q.63 Explain about the data abstraction for SIMULA 67. ☞ [JNTU : Part B, Dec.-17, Marks 5]

Ans. : This is a first language that used the notion of class.

- Objects are called instances of class.
- Class is similar to type but includes procedures, functions and variables. Following is a representation of class in Simula

Classes are made of :
- Parameters
- Attributes
- Methods
- Life (Body)

- Following is a sample code for class in SIMULA 67

```
Class Rectangle (Width, Height); Real Width, Height;
                    ! Class with two parameters;
Begin
  Real Area, Perimeter;  ! Attributes;

  Procedure Update;      ! Methods;
  Begin
    Area := Width * Height;
    Perimeter := 2*(Width + Height)
  End of Update;

  Boolean Procedure IsSquare;
    IsSquare := Width=Height;

  Update;              ! Life of rectangle started at creation;
  OutText("Rectangle created: "); OutFix(Width,2,6);
  OutFix(Height,2,6); OutImage
End of Rectangle;
```

3.23 Parameterized ADT

Q.64 What is parameterized ADT ? ☞ [JNTU : Part A, Marks 2]

Ans. : Parameterized ADTs allow designing an ADT that can store any type elements – only an issue for static typed languages.

- It is also known as generic classes
- C++, Ada, Java 5.0, and C# 2005 provide support for parameterized ADTs.

Q.65 Give an example of parameterized ADT in ADA. ☞ [JNTU : Part B, Marks 5]

Ans. : The generic package can be used for implementing the stack of any type of elements is as follows –

```
generic
  Max: Positive;
  type Element_T is private;
package Generic_Stack is
  procedure Push (E: Element_T);
  function Pop return Element_T;
end Generic_Stack;
package body Generic_Stack is
  Stack: array (1 .. Max) of Element_T;
  Top  : Integer range 0 .. Max := 0;  -- initialise to empty
  -- ...
end Generic_Stack;
```

A stack of a given size and type could be defined in this way:

```
declare
  package Float_100_Stack is new Generic_Stack (100, Float);
```

```
use Float_100_Stack;
begin
 Push (45.8);
 -- ...
end;
```

Q.66 Explain generic abstraction in C++. ☞ [JNTU : Part B, Marks 5]

Ans. : Templates are considered to be the generic abstract data type. In case of templates, the data components can b of different types however, the operations are the same.

Using class template we can write a class whose members use template parameters **as types.**

The complete program using class template is as given below.

```
#include <iostream.h>
template <class T>
class Compare { //writing the class as usual
            T a, b;  //note we have used data type as T
            public:
           Compare (T first, T second)
          {
           a=first;
           b=second;
          }
          T max ();//finds the maximum element among two
    };
    //template class member function definition
    //here the member function of template class is max
template <class T>
T Compare <T>:: max ()
{
    T val;
    if(a>b)
       val=a;
    else
        val=b;
     return val;
}
    void main ()
{
    Compare <int> obj1 (100, 60);//comparing two integers
    Compare <char> obj2('p','t');  //comparing two characters
    cout <<"\n maximum(100,60) = "<obj1.max();
    cout<<"\n maximum('p','t') = "<obj2.max();
}
```

Output

```
maximum(100,60) = 100
maximum('p','t') = t
```

Q.67 Explain implementation of parametrized ADT in JAVA. ☞ [JNTU : Part A, Marks 3]

Ans. : In Java, Generic parameters must be classes.

- Most common generic types are the collection types, such as LinkedList and ArrayList.
- It eliminates the problem of having multiple types in a structure.
- For example -

ArrayList myArray =new ArrayList ();

3.24 Encapsulation Constructs

Q.68 Write short note on – encapsulation constructs. ☞ [JNTU : Part B, Marks 5]

Ans. : Large programs have **two special needs :**

- Some means of **organization,** other than simply division into subprograms
- Some means of **partial compilation** (compilation units that are smaller than the whole program)

- The solution is making a grouping of subprograms that are logically related into a unit that can be separately compiled (compilation units). Such collections are called **encapsulation.**

Q.69 What are nested subprograms ? ☞ [JNTU : Part A, Marks 2]

Ans. : • Organizing programs by nesting subprogram definitions inside the logically larger subprograms that use them.

- Nested subprograms are supported in Ada, Fortran 95, Python, and Ruby.

Q.70 Explain the encapsulation in C with suitable example.

Ans. : Different languages provide different encapsulation facilities. For example in C - file is a unit of encapsulation.

For example - Following steps demonstrate the encapsulation mechanism used in C.

Step 1 : Create an external file named **add.h** in which the implementation of addition functionalities is specified.

```
int add(int x, int y)
   {
      return x + y;
   }
```

Step 2 : Now create a C program which simply invokes the addition functionality to perform addition of two numbers

```
#include<stdio.h>
#include "d:\\add.h"
void main()
{
   int result = add(10, 20);  //implementation details of this function are hidden
   printf("%d", result);
}
```

Step 3 : On compiling the above C program, the output of above program is 30.

- In C++, the encapsulation is provided by means of a class. For instance

```
class Test
   {
      private:
      int a,b;
      public:
      int add(a,b);
```

Encapsulated data members and member functions in a single unit called class.

Q.71 Explain the encapsulation in C++.Also explain use of friend function in C++ with example.

Ans. : In C++, the classes encapsulate data members and member functions.

- The member definitions can be defined in a separate file
- Friends provide a way to grant access to private members of a class.

friend Function is a special function that can be declared anywhere in the class but it can access private members of class. The interaction with a member function of a class module is possible via friend function. Following C++ program shows how a friend function interacts with the private member of a class.

```
class test
{
 int data;
 friend int fun(int x);//declaration of friend function
 public:
     test()//constructor
     {
         data = 5;
     }
 };
 int fun(int x)
 {
  test obj;
  //accessing private data by friend function
  return obj.data + x;
 }
 int main()
 {
  cout << "Result is = "<< fun(4)<<endl;
 }
```

Similar to a friend function one can declare a class as a friend to another class. Thus grouping of several units and interaction among those units can be possible using friend function or friend class in C++.

3.25 Naming Encapsulations

Q.72 What is the use of naming encapsulation in programming languages ? ☞ [JNTU : Part A, Marks 2]

Ans. : Large programs define many global names need a way to divide into logical groupings.

- A naming encapsulation is used to create a new scope for names.
- C++, C#, Java, Ada, and Ruby provide naming encapsulations.

Q.73 What are namespaces ? [JNTU : Part A, Marks 2]

Ans. : Namespaces are used to group the entities like class, variables, objects, function under a name. The namespaces help to divide global scope into sub-scopes, where each sub-scope has its own name.

For example - following C++ program shows how to create namespaces

```
namespace ns1
{
 int a = 5;
}
namespace ns2
{
 char a[] = "Hello";
}
int main ()
{
 cout<<ns1::a << endl;
 cout<<ns2::a << endl;
 return 0;
}
```

Output
5
Hello

Program Explanation :

In above program there are two different namespaces containing the variable **a**. The variable **a** in the first namespace **ns1** is of type **int** but the variable **a** in the second namespace **ns2** is of array of characters. The both the entities are treated separately. There is no re-declaration error for variable **a**.

All the files in the C++ standard library declare all of its entities within the **std** namespace. That is why in the C++ program the **std** namespace is declared as follows -

using namespace std;

This statement is used in the program which uses any entity declared in **iostream**.

Fill in the Blanks for Mid Term Exam

Q.1 Most of the programming language support two types of routine and those are _____ and _____.

Q.2 The default parameter passing method used in C++ is _____.

Q.3 ADA support the creation of generic unit by using the keyword _____.

Q.4 The _____ is an explicit request that subprogram be executed.

Q.5 The _____ is first part of the definition, including the name, the kind of subprogram and the formal parameters.

Q.6 _____ is a mechanism in which we can use many methods having the same function name but can pass different number of parameters or different types of parameter.

Q.7 The subprogram call and return operations of a language are together called its _____.

Q.8 The scope of loop parameter in ADA is _____.

UNIT - IV

4 Concurrency and Exception Handling

Part I : Concurrency

4.1 Introduction to Subprogram Level Concurrency

Q.1 What are the three possible levels of concurrency in programs ? ☞ [JNTU : Part A, Marks 3]

Ans. : 1) **Instruction level :** Executing two or more machine instructions simultaneously.

2) **Statement level :** Executing two or more high-level language statements simultaneously.

3) **Unit level :** Executing two or more subprogram units simultaneously.

Q.2 What is the difference between physical and logical concurrency ? ☞ [JNTU : Part A, Marks 2]

Ans. : • Physical concurrency is several program units from the same program that literally execute simultaneously.

- Logical concurrency is multiple processors providing actual concurrency, when in fact the actual execution of programs is taking place in interleaved fashion on a single processor.

Q.3 Briefly explain the sub-program level concurrency.

Ans. ; • The term concurrency can be defined as the expression of the task in the form of multiple interacting sub-tasks that can be potentially executed at the same time.

- If the communication between dependent tasks is not appropriately designed then data race condition can occur.
- Interaction among the processes take two forms -
  - **Communication :** It involves exchange of data between processes either through messages, or shared variables.
  - **Synchronization :** It involves synchronization of one task with another. Generally thread execution is involved in synchronization.
- Using appropriate **concurrency models** the proper coordination of communication and synchronization between tasks can be achieved.
- The concurrent models dictate when and how the communication between the tasks should occur.
- Concurrent programming can be done with the help of **threads**. Thread is a lightweight process which makes use of the address space of its parent process for execution. Each thread while running, executes some task.

Q.4 Distinguish between competitive synchronization and cooperation synchronization.

☞ [JNTU : Part A, Dec.-17, Marks 3]

Ans. : Cooperation Synchronization : The cooperation synchronization is a kind of synchronization in which Task A must wait for task B to complete some specific activity before task A can continue its execution. For example – Producer consumer problem is based on cooperation synchronization.

Competitive Synchronization : The competitive synchronization is a kind of synchronization in which two or more tasks must use some resource that cannot be simultaneously used. For example - Shared counter.

Q.5 What is scheduler ? Explain.

☞ [JNTU : Part A, Marks 2]

Ans. : Task execution control is maintained by a program called the scheduler, which maps task execution onto available processors.

4.2 Semaphores, Monitors, Message Passing

Q.6 What is semaphore ? What are the operations on semaphores ? ☞ [JNTU : Part B, Nov 15, Marks 5]

Ans. :

- Semaphore is a shared variable or integer variable used to synchronize the progress of interacting processes.
- The semaphore works using two operations wait and signal designated by P() and V() respectively. P() means try to decrease and V() means try to increase().
- **Working of Semaphore :**

Step 1 : Semaphore may be initialized to non negative value.

Step 2 : The **wait** operation decrements the semaphore value by one. If the value becomes negative, then the process executing **wait()** operation is blocked.

Step 3 : The **signal** operation increments the semaphore value by one. If the value is not positive, then the process blocked by wait operation is unblocked.

- Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until $s > 0$, this can only happen when P1 finishes its critical section and calls V operation on semaphores. This way mutual exclusion is achieved

Q.7 What are advantages and disadvantages of semaphore ?

Ans. : Advantage :

1) Semaphores allow only one process into the critical section. Hence it is efficient method of synchronization.
2) There is no wastage of resources.
3) Semaphores are machine independent.

Disadvantages :

1) The code must be correctly implemented otherwise it may lead to deadlock.
2) There are chances that low priority processes may access the critical section before the high priority processes.

Q.8 What is semaphore ? Explain its role in concurrency. ☞ [JNTU : Part B, March 17, Marks 5]

Ans. : Refer Q.6.

Q.9 What is semaphore ? Explain the different types of semaphores. ☞ [JNTU : Part B, Dec 17, Marks 5]

Ans. : Semaphore : Refer Q.6.

There are two types of semaphores.

1) **Binary Semaphore :** Binary semaphore is used when there is only one shared resource.

   Binary semaphore exists in two states i.e. Acquired(Take), Released(Give). Binary semaphores have no ownership and can be released by any task. Binary semaphore allows only one thread to access the resource at a time.

2) **Counting Semaphore:** To handle more than one shared resource of the same type, counting semaphore is used. Counting semaphore will be initialized with the count(N) and it will allocate the resource as long as count becomes zero after which the requesting task will enter blocked state.

Q.10 Explain how Binary semaphore provides solution to the competition synchronization problem with example pseudo code.

☞ [JNTU : Part B, May 18, Marks 10]

Ans. : Consider an implementation of a counting semaphore S. Clearly the implementation needs to keep track of the value of S, say in an integer variable S.val. Furthermore, if a process is supposed to wait on S, then the implementation has to make the process wait. Because binary semaphores are the only synchronization construct allowed in the implementation, the only way that the

implementation can make the process wait is to have it wait on a binary semaphore, say S.wait

Consider that P(S) and V(S) to denote wait and signal operations on a semaphore S.

The pseudo code can be as follows -

```
record S {
   integer val initially K,
   BinarySemaphore wait  initially 0
}
P(S) {
   if S.val = 0
     then { P( S.wait ) }
     else { S.val := S.val - 1 }
}
V(S) {
   if "processes are waiting on S.wait"
     then  {  V( S.wait ) }
     else { S.val := S.val + 1 }
}
```

Q.11 What is monitor ? What are its advantages and disadvantages over semaphores ?

☞ [JNTU : Part B, Marks 5]

OR Compare Semaphores with monitors.

☞ [JNTU : Part A, March 16, Marks 3]

Ans. : The Monitor type is an abstract data type whose construct allow one process to get activate at one time. Only one process can be active in a monitor at a time. Other processes that need to access the shared variables in a monitor have to line up in a queue and are only provided access when the previous process release the shared variables.

Advantages and disadvantages of monitor over semaphores

a) Mutual exclusion in monitor is automatic while mutual exclusion in semaphore needs to be coded explicitly.

b) Shared variables are global to all processes in the monitor while shared variables are hidden in semaphores.

c) Synchronization in monitors is provided by condition variables while there are no condition variables in semaphores.

d) Access to the monitor is through protected shared variables while access to shared variables in semaphores is not protected.

e) Monitors are programming language constructs not only they need shared memory but they are not available in many programming languages/

Q.12 What advantages do monitors have over semaphores ?

☞ [JNTU : Part A,Nov 15, Marks 3]

Ans. : Refer Q.11.

Q.13 Define Semaphore and monitor

☞ [JNTU : Part A, Dec16, Marks 3]

Ans. : Refer Q.6 and Q.11.

4.3 Ada Support for Concurrency

Q.14 Explain the Task execution in Ada in detail with pseudo code.

☞ [JNTU : Part B, marks 10]

Ans. : A major feature of the Ada programming language is the facilities it provides for concurrent programming.

Ada tasks have specification and body parts, like packages; the specification has the interface, which is the collection of entry points :

Step 1 : Specification : The specification gives the entries

```
task type T is
    entry Put (data : in Integer);
    entry Get (result : out Integer);
end T;
```

The entries are used to access the task

Step 2 : Task Body Declaration :

Task body gives actual code of task

```
task body T is
    x : integer;
    -- local per thread declaration
begin
    ...
    accept Put (M : Integer) do
       ...
    end Put;
    ...
end T;
```

Step 3 : Creating an Instance of Task

Declare a single task as follows -

```
X : T;
or an array of tasks
P : array (1 .. 50) of T;
or a dynamically allocated task
type AT is access T;
P : AT;
...
P := new T;
```

Step 4 : Execution of Task

i) an accept call

wait for someone to call entry, then proceed with rendezvous code, then both tasks go on their way

ii) an entry call

wait for addressed task to reach corresponding accept statement, then proceed with rendezvous, then both tasks go on their way.

During the Rendezvous, only the called task executes, and data can be safely exchanged via the entry parameters.

Step 5 : Termination of Tasks

A task terminates when it reaches the end of the begin-end code of its body.

Tasks may either be very static (create at start of execution and never terminate)Or very dynamic, e.g. create a new task for each new radar trace in a radar system.

Q.15 How will you set message passing priorities in Ada ? ☞ **[JNTU : Part B, Marks 2]**

Ans. : The priority of any task can be set with the pragma Priority

pragma Priority (static expression);

The priority of a task applies to it only when it is in the task ready queue

Q.16 What is protected objects in Ada ?

Ans. :

- A more efficient way of implementing shared data to allow access to a shared data structure to be done without rendezvous.
- A protected object is similar to an **abstract data type.**
- Access to a protected object is either through messages passed to entries, as with a task, or through **protected subprograms** (i.e. either subprogram or function).
  - **A protected procedure** provides **mutually exclusive read-write access** to protected objects.
  - **A protected function** provides concurrent read-only access to protected objects.

4.4 Java Threads

Q.17 Describe the life cycle of Java thread. ☞ **[JNTU : Part B, Marks 5]**

Ans. : Thread is a light-weight program. Java life cycle specifies how a thread gets processed in the Java program. By executing various methods. Following Fig. Q.17.1 represents how a particular thread can be in one of the state at any time.

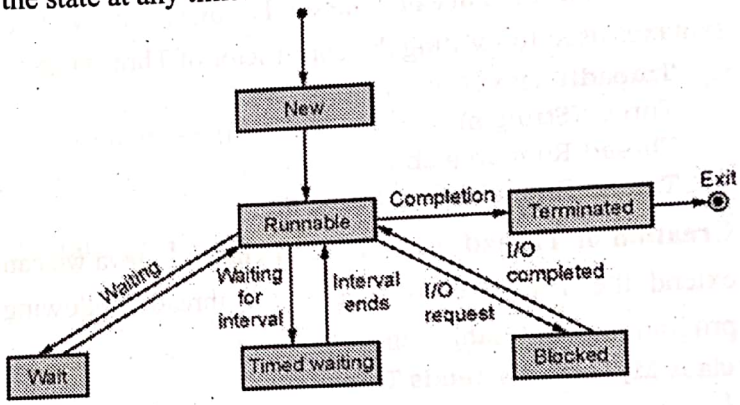


Fig. Q.17.1 Life cycle of thread

New state

- When a thread starts its life cycle it enters in the new state or a create state.

Runnable state

This is a state in which a thread starts executing.

- Waiting state
- Sometimes one thread has to undergo in waiting state because another thread starts executing.

Timed waiting state

- There is a provision to keep a particular threading waiting for some time interval. This allows to execute

high prioritized threads first. After the timing gets over, the thread in waiting state enters in runnable state.

Blocked state

- When a particular thread issues an Input/Output request then operating system sends it in blocked state until the I/O operation gets completed. After the I/O completion the thread is sent back to the runnable state.

Terminated state

- After successful completion of the execution the thread in runnable state enters the terminated state.

Q.18 Explain in detail, how to create a thread in Java ?

☞ [JNTU : Part B, Marks 5]

Ans. : In Java, Thread can be created using

1. Thread class 2. Runnable interface.

Constructor of Thread Class : Following are various syntaxes used for writing the constructor of Thread Class.

```
Thread()
Thread(String s)
Thread(Runnable obj)
Thread(Runnable obj, String s);
```

Creation of Thread using Thread class : In java we can extend the Thread class to create a thread. Following program illustrates this concept

```
class MyThread extends Thread
{
  public void run()
  {
    System.out.println("Thread is created!!!");
  }
}
class ThreadProg
{
  public static void main(String args[])
  {
    MyThread t=new MyThread();
    t.start();
  }
}
```

The thread can also be created using **Runnable** interface.

Creation of Thread using Runnable interface : Following Java program shows how to implement **Runnable interface** for creating a single thread.

```
class MyThread implements Runnable
{
  public void run()
  {
    System.out.println("Thread is created!");
  }
}
class ThreadProgRunn
{
  public static void main(String args[])
  {
    MyThread obj=new MyThread();
    Thread t=new Thread(obj);
    t.start();
  }
}
```

Q.19 Explain the methods used for controlling thread execution.

☞ [JNTU : Part B, Marks 5]

Ans. : 1) The yield is a request from the running thread to voluntarily surrender the processor

2) The sleep method can be used by the caller of the method to block the thread

3) The join method is used to force a method to delay its execution until the run method of another thread has completed its execution

Q.20 What is meant by thread priority in Java ?

☞ [JNTU : Part A, Marks 3]

Ans. : In Java threads scheduler selects the threads using their priorities. The thread priority is a simple integer value that can be assigned to the particular thread. These priorities can range from 1 (lowest priority) to 10 (highest priority).

There are two commonly used functionalities in thread scheduling -

- setPriority
- getPriority

The function setPriority is used to set the priority to each thread

```
Thread_Name.setPriority(priority_val);
```

The priority_val is a constant value denoting the priority for the thread. It is defined as follows -

MAX_PRIORITY = 10
MIN_PRIORITY = 1
NORM_ PRIORITY = 5

The function getPriority is used to get the priority of the thread.

```
Thread_Name.getPriority();
```

Q.21 How does Java achieve the synchronization in thread exectution ? ☞ **[JNTU : Part B, Marks 5]**

Ans. : 1. When two or more threads need to access shared memory, then there is some way to ensure that the access to the resource will be by only one thread at a time. The process of ensuring one access at a time by one thread is called synchronization.

2. There are two ways to achieve synchronization – i) Using Synchronized methods ii) Using Synchronized blocks

Q.22 Discuss about wait( ), notify( ) and notifyAll( ) methods in Java.

Ans. : Polling is a mechanism generally implemented in a loop in which certain condition is repeatedly checked.

- To better understand the concept of polling, consider producer-consumer problem in which producer thread produces and the consumer thread consumes whatever is produced.
- Both must work in co-ordination to avoid wastage of CPU cycles.
- But there are situations in which the producer has to wait for the consumer to finish consuming of data. Similarly the consumer may need to wait for the producer to produce the data.
- In Polling system either consumer will waste many CPU cycles when waiting for producer to produce or the producer will waste CPU cycles when waiting for the consumer to consume the data.
- In order to avoid polling **there are three in-built methods** that take part in inter-thread communication.

notify()	If a particular thread is in the sleep mode then that thread can be resumed using the notify call.
notifyall()	This method resumes all the threads that are in suspended state.
wait()	The calling thread can be send into a sleep mode.

Q.23 Write short note on - C# Threads

☞ **[JNTU : Part B, Nov 15, March 17, Marks 5**

Ans. : Thread is a lightweight process. It is basically the execution path of a program. Each thread defines a unique flow of control.

In C#, the System.Threading.Thread class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application.

Example Code

```
using System;
using System.Threading;
class ThreadDemo
{
  static void Main()
  {
    Thread th = new Thread(new ThreadStart(fun));
    thread1.Start();
  }

  static void fun()
  {
    Thread.Sleep(100);
    Console.WriteLine('Thread is running');
  }
}
```

Q.24 Differentiate between Java Thread and C# Thread

☞ **[JNTU : Part B, March 16, Marks 5**

Ans. :

Sr.No.	Java	C#
1.	Creation of thread in Java as follows – Make a class extend Thread class or implement Runnable interface: Thread t= new Thread(class);	To create a thread in C# Thread t = new Thread(new ThreadStart(method));

2.	To start the thread we write t.start();	To start a the thread we write t.Start();
3.	To kill the thread t=null;	To kill thread t.Abort();
4.	To interrupt thread t.stop();	To interrupt thread t.Interrupt();

Q.25 What are C# threads ? Give the steps that demonstrate how to create a thread in C#. Also discuss the various states of thread in C#.

Ans. : C# Thread and its demonstration : Refer Q. 23.

States of Threads in C# : There are following states in the life cycle of a Thread in C#.

1) **Unstarted State :** When the instance of Thread class is created, it is in unstarted state by default.
2) **Runnable State :** When start() method on the thread is called, it is in runnable or ready to run state.
3) **Running State :** Only one thread within a process can be executed at a time. At the time of execution, thread is in running state.
4) **Not Runnable State :** The thread is in not runnable state, if sleep() or wait() method is called on the thread, or input/output operation is blocked.
5) **Dead State :** After completing the task, thread enters into dead or terminated state.

4.5 Concurrency in Functional Languages

Q.26 Explain the concurrency in ML.

☞ **[JNTU : Part B, Marks 5]**

Ans. : • Concurrent ML (CML) is an extension to ML.

- It includes a form of threads and a form of synchronous message passing to support concurrency.
- A thread is created in CML with the spawn primitive, which takes the function as its parameter. The function is specified as an anonymous function.
- As soon as the thread is created, the function begins its execution in the new thread. The effects of the function are either output produced or through communications with other threads.
- Channels provide the means of communicating between threads. A channel is created with the channel constructor. For example

```
let val chnl = channel()
```

- The two primary operations (functions) on channels are for sending (send) and receiving (recv) messages.
- The send function that sends the integer value 100 can be written as follows –

```
send(chnl,100)
```

- The recv function names the channel as its parameter. Its return value is the value it received.
- In the concurrent ML communications are synchronous. That means the messages are sent only if both the sender and receiver are ready.
- If a thread sends a message on a channel and no other thread is ready to receive on that channel, the sender is blocked and waits for another thread to execute a recv on the channel.
- Similarly, if a recv is executed on a channel by a thread but no other thread has sent a message on that channel, the thread that ran the recv is blocked and waits for a message on that channel.

Q.27 What is the use of the spawn primitive of CML ?

☞ **[JNTU : Part A, Marks 2]**

Ans. : The use of Spawn primitive of CML is to create a thread.

Q.28 Explain the concurrency in F#.

☞ **[JNTU : Part A, Marks 3]**

Ans. :

- The support for F# is based on the same .NET classes that are used by C#, especially **System. Threading. Thread**
- For example – Suppose we want to run the function myMethod in its own thread. The following function, when called, will create the thread and start the execution of the function in the new thread:

```
let createThread() = let t = new Thread(myMethod)
t.Start()
```

- There exists a Thread class defines the Sleep method, which puts the thread from which it is called to sleep for the number of milliseconds that is sent to it as a parameter.

4.6 Statement Level Concurrency

Q.29 What is the purpose of statement level concurrency ? ☞ **[JNTU : Part A, Marks 2]**

Ans. : The statement level concurrency provides a mechanism that programmer can use to inform compiler of ways it can map the program onto multiprocessor architecture.

Q.30 Explain the concept of High-Perform Fortran

☞ **[JNTU : Part A, Marks 2]**

Ans. : • High-Performance Fortran (HPF) is a collection of extensions to Fortran 90 that are meant to allow programmers to specify information to the compiler to help it optimize the execution of programs on multiprocessor computers.

- The primary specification statements of HPF are for specifying the number of processors, the distribution of data over the memories of those processors, and the alignment of data with other data in terms of memory placement.

Part II : Exception Handling and Event Handling

4.7 Introduction to Exception Handling

Q.31 Explain the design issues of an exception handling system. ☞ **[JNTU : Part B, Dec 17, Marks 5**

Ans. : The design issues of an exception handling system are as follows –

1) How and where are exception handlers specified and what is their scope ?
2) How is an exception occurrence bound to an exception handler ?
3) Can information about the exception be passed to the handler ?
4) Where does execution continue, if at all, after an exception handler completes its execution ? (continuation vs. resumption)
5) Is some form of finalization provided ?

4.8 Exception Handling in Ada, C++, Java

Q.32 Explain the exception handling mechanism in C++ with illustrative example.

☞ **[JNTU : Part B, March 17, Marks 5]**

OR. Explain how to handle the exceptions in C++.

☞ **[JNTU : Part B, Dec 17, Marks 5]**

Ans. : Definition : When any unavoidable circumstances (or runtime errors) occur in our program then exceptions are raised by handling control to special functions called handlers. This provides build-in error handling mechanism which is known as exception handling.

Mainly exception handling is done with the help of three keywords: try, catch and throw. The try and catch block is as shown below -

```
try
{
   throw exception;
 //exception is some value
 // the portion of the code that is to be monitored for
//error detection
}
catch( argument)
{
//catch block softly handles the exception
}
```

The try block contains the portion of the program that is to be examined for error detection. If an exception (i.e. error) occurs in this block then it is thrown using throw. Using catch the exception is caught and processed. If try block contains all the code included in main then effectively the complete program must be scanned for errors. Any exception is caught by the catch block. This catch block should be immediately followed by the try block.

For example – Following C++ program is for handling **divide by zero** exception

```
int main()
{
    double i, j;
    void divide(double, double );
    cout << "Enter numerator : ";
    cin >> i;
    cout << "Enter denominator: ";
    cin >> j;
    divide(i, j);
}
void divide(double a, double b)
{
   try
   {
      if(b==0)
         throw b; // divide-by-zero
      cout << "Result: " << a/b << endl; // for non
zero value
   }
   catch (double b)
   {
    cout << "Can't divide by zero.\n";
   }
}
```

Q.33 Explain how to handle exceptions in Java with an example. ☞ **[JNTU : Part B, Dec 17, Marks 5]**

OR What is exception handling ? Explain about exception handling in Java. ☞ **[JNTU : Part B, May 18, Marks 5]**

Ans. : Exception Handling : Refer Q.32

Example Program in Java

```
class RunErrDemo
{
   public static void main(String[] args)
   {
      int a,b,c;
      a=10;
      b=0;
      try {
         c=a/b;
      }
      catch(ArithmeticException e)  {
         System.out.println("\n Divide by zero");
      }
      System.out.println("\n The value of a: "+a);
      System.out.println("\n The value of b: "+b);
   }
```

Output

Divide by zero

The value of a: 10

The value of b: 0

Q.34 Describe the functionality of 'finally'' clause of JAVA exception handling mechanism. ☞ **[JNTU : Part A, March 17,May 18, Marks 3]**

Ans. : • Sometimes because of execution of try block the execution gets break off. And due to this some important code (which comes after throwing off an exception) may not get executed. That means, sometimes try block may bring some unwanted things to happen.

- The finally block provides the assurance of execution of some important code that must be executed after the try block.

Example Code

```
/* This is a java program which shows the use of finally
block for handling exception */
class finallyDemo {
   public static void main(String args[]) {
      int a=10,b=-1;
      try {
         b=a/0;
      }
      catch(ArithmeticException e) {
         System.out.println("In catch block: "+e);
      }
      finally {
         if(b!=-1)
            System.out.println("Finally block executes
            without occurrence of exception");
         else
            System.out.println("Finally block executes
            on occurrence of exception");
      }
   }
}
```

Q.35 Explain in brief about exception handling in Ada.

[JNTU : Part B, Nov 15, Dec 16, Marks 5]

OR What is exception ? How to handle the exception in Ada with an example ?

[JNTU : Part B, March 16, Marks 10]

Ans. : • **Exception :** An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions during the execution of a program.

- The frame of an exception handler in Ada is either a subprogram body, a package body, a task, or a block
- Because exception handlers are usually local to the code in which the exception can be raised, they do not have parameters.
- If the block or unit in which an exception is raised does not have a handler for that exception, the exception is propagated elsewhere to be handled. For instance -
  - Procedures - propagate it to the caller
  - Blocks - propagate it to the scope in which it appears
  - Package body - propagate it to the declaration part of the unit that declared the package.

Task - no propagation; if it has a handler, execute it; in either case, mark it **completed**

- **Syntax:**
  - User-defined Exceptions form:
    exception_name_list : exception;
  - Raising Exceptions form:
    raise [exception_name]
- **Example of Exception in ADA**

```
with TEXT_IO; use TEXT_IO;

procedure exception_factorial is

package int_io is new integer_io (integer); use int_io;

function factorial(n: integer) return integer is
 result: integer := 1;
 begin
 for i in 1 .. n loop
  result := result*n;
 end loop;
 return result;
exception
 when constraint_error =>
  put("overflow in factorial -- n = ");
  put(n);  new_line;
  return integer'last;
 end factorial;


begin
-- main program here
put(factorial(5));
new_line;
put(factorial(1000));

end exception_factorial;
```

- **Predefined Exceptions :** The built-in exceptions of Ada generally name groups of events. These are the following :
  - **CONSTRAINT_ERROR :** The event group that occurs if a declaration constraint is violated, for example, an index bound is exceeded.
  - **NUMERIC_ERROR :** Arithmetic errors, including underflow and overflow errors, division by zero.
  - **STORAGE_ERROR :** Memory errors (including all allocation-related problems): the memory region referred to is not available.
  - **TASKING_ERROR :** Errors associated with tasks.

Q.36 What is the difference between checked and unchecked exception in Java ?

[JNTU : Part B, Dec 16, Marks 5]

Ans. :

Checked Exception : These types of exceptions need to be handled explicitly by the code itself either by using the try-catch block or by using throws. These exceptions are extended from the **java.lang.Exception** class. For example : **IOException** which should be handled using the try-catch block.

UnChecked Exception : These type of exceptions need not be handled explicitly. The Java Virtual Machine handles these type of exceptions. These exceptions are extended from **java.lang.RuntimeException** class. For example : **ArrayIndexOutOfBounds, NullPointerException, RunTimeException.**

Q.37 Explain throwable class and draw the exception hierarchy in Java.

☞ **[JNTU : Part B, Marks 5]**

Ans. :

- The Throwable class is a superclass of all errors a exceptions in Java. The declaration of Throwable cla is as follows -

public class Throwable extends Object implemen Serializable

- Various exception classes get derived fr **Throwable** class.
- This class hierarchy is given in the follow Fig. Q.37.1

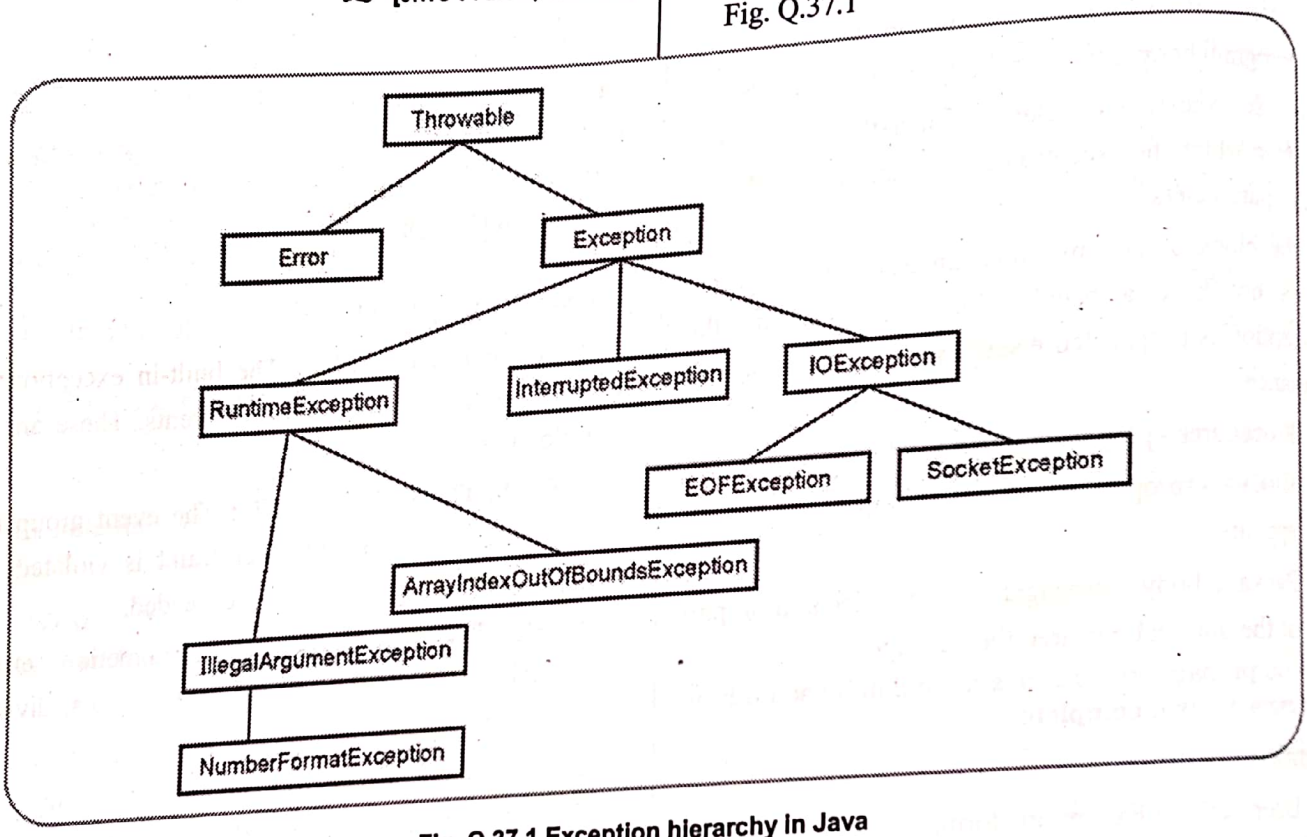


Fig. Q.37.1 Exception hierarchy in Java

Q.38 Explain the two subclasses of throwable in Java.

☞ **[JNTU : Part A, Mar**

Ans. : The Java library includes two subclasses of Throwable :

(1) Error :

- Thrown by the Java interpreter for events such as heap overflow.
- It is never handled by user programs

(2) Exception :

- User-defined exceptions are usually subclasses of this class.
- It has two predefined subclasses such as IOException and RuntimeException and so on.

4.9 Introduction to Event Handling

Q.39 Explain the term event and event handling. ☞ [JNTU : Part A, Marks 2]

Ans. :

Event : An event is a notification that something specific has occurred, such as a mouse click on a graphical button.

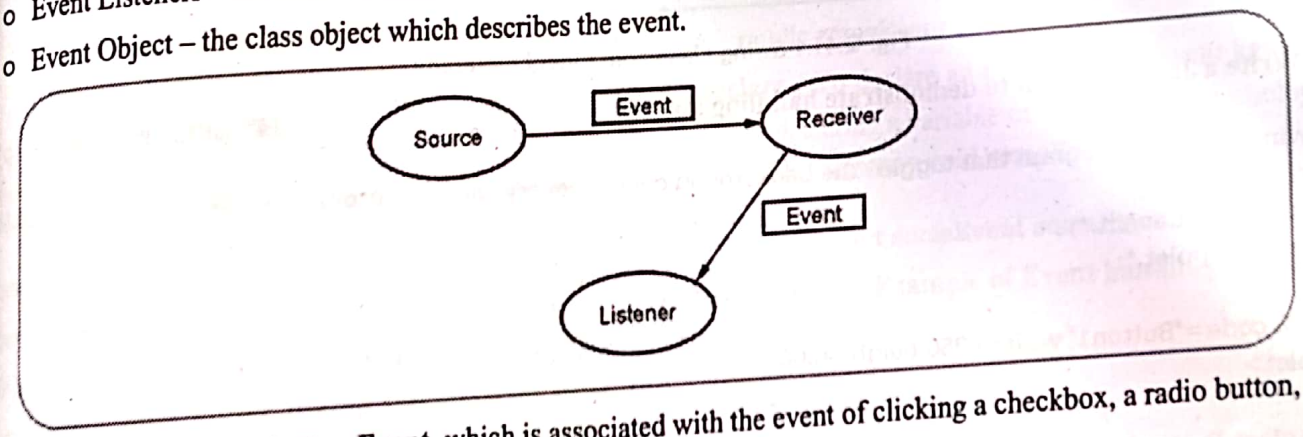
Event Handler : The event handler is a collection of instructions that is executed in response to an event.

4.10 Event Handling with Java and C#

Q.40 Explain event model in Java . ☞ [JNTU : part B, Marks 5]

Ans. :

- The event model is based on the Event Source and Event Listeners.
- Event Listener is an object that receives the messages / events.
- The Event Source is any object which creates the message / event.
- The Event Delegation model is based on – The Event Classes, The Event Listeners, Event Objects.
- There are three participants in event delegation model in Java -
  - Event Source – the class which broadcasts the events
  - Event Listeners – the classes which receive notifications of events
  - Event Object – the class object which describes the event.



- One class of events is **ItemEvent**, which is associated with the event of clicking a checkbox, a radio button, or a list item.
- The **ItemListener** interface prescribes a method, **itemStateChanged**, which is a handler for **ItemEvent** events.
- The listener is created with **addItemListener**

Q.41 What is swing ? Give the swing component hierarchy. ☞ [JNTU : Part B, Marks 5]

Ans. :

- Swing is an approach of graphical programming in Java.
- Swing creates highly interactive GUI applications.
- It is the most flexible and robust approach.
- In order to display any JComponent on the GUI, it is necessary to add this component to the container first. If you do not add these components to container then it will not be displayed on the GUI.
- The swing class component hierarchy is as shown by following Fig. Q.41.1.

- There are two important features of swing components - Firstly, all the component classes begin with the letter and secondly all these GUI components are descendant of JComponent class.

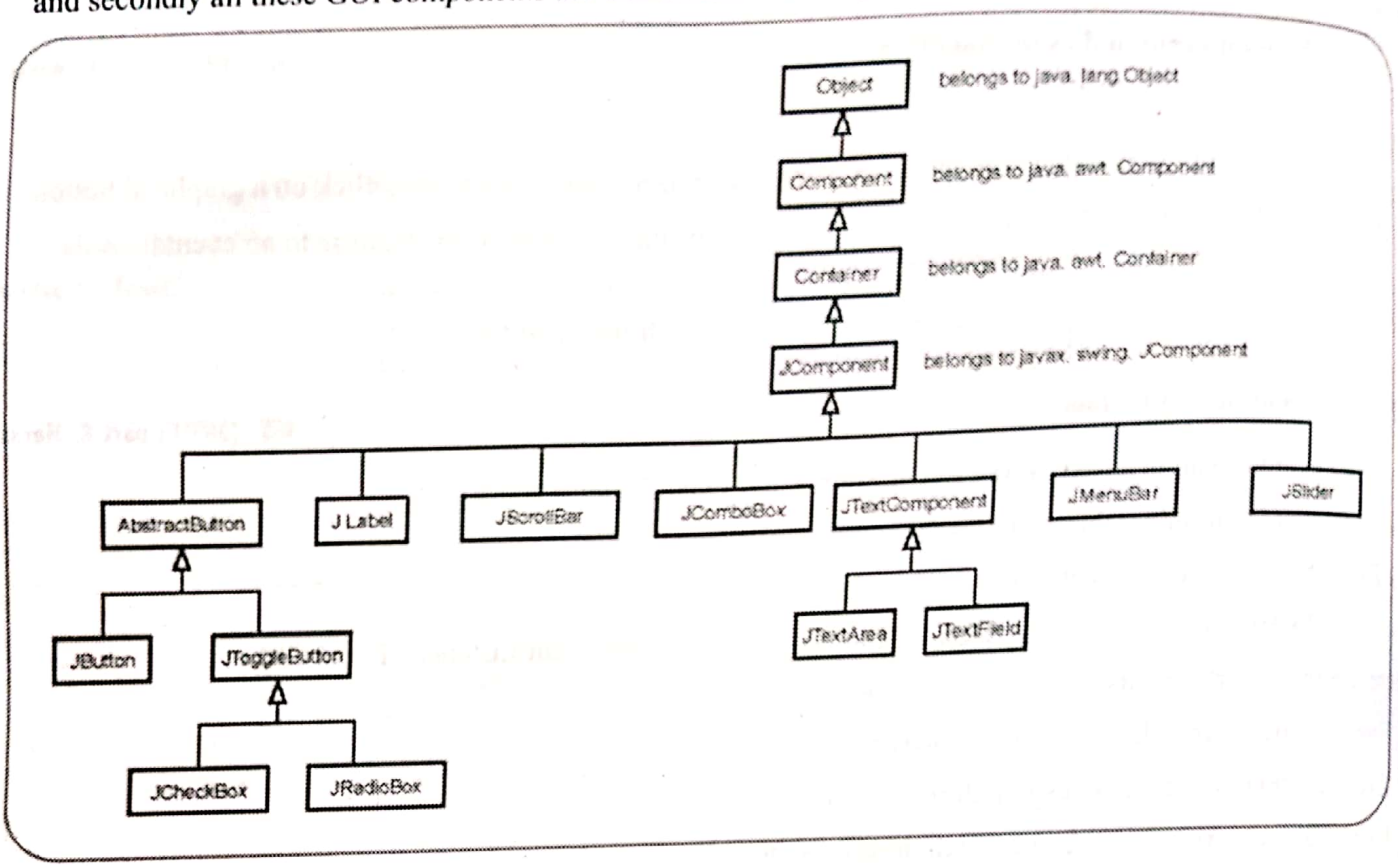


Fig. Q.41.1 Swing class component hierarchy

Q.42 Write a Java program to demonstrate handling an event. [JNTU : Part B, Marks 10]

Ans. :

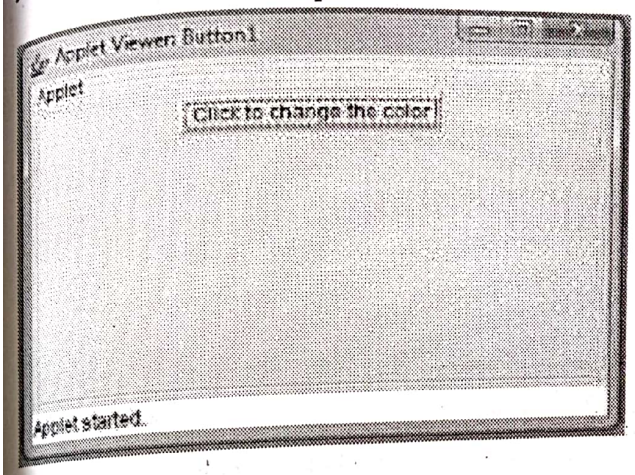
Following is a Java program that toggles the background color on every click of button.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="Button1" width=350 height=200>
</applet>
*/
public class Button1 extends Applet
implements ActionListener
{

Button button=new Button("change the color");
boolean flag=true;
public void init()
{
 add(button);
 button.addActionListener(this);
}
public void paint(Graphics g)
{
 if(flag)
```

Event handler is called here

```
setBackground(Color.yellow);
else
setBackground(Color.red);

}
public void actionPerformed(ActionEvent e)
{
String str=e.getActionCommand();
if(str.equals("change the color"))
{
flag=!flag;
//toggle the flag values on every click of button
repaint();
}
}
}
```

Output



Program Explanation

In above program,

1. When a button is clicked the action of changing the background color occurs. Hence this program must implement the **ActionListener** interface.
2. This listener class must have a method called **actionPerformed** which has a parameter an object-**ActionEvent**.
3. By invoking the method **getActionCommand** we can recognize that the event has occurred by clicking the button.

Q.43 Enlist the features of event handling in C#.

☞ **[JNTU : Part A, Marks 2]]**

Ans. : 1. An Event is created using **event** keyword.

2. An Event has **no return type** and it is always **void.**
3. All events are based on **delegates.**
4. All the published events must have a **listening object.**

Q.44 How event is handled in C# ? Explain.

☞ **[JNTU : Part B, Marks 5]**

Ans. : • In event handler, in C#, is a method that contains the code that gets executed in response to a specific event that occurs in an application.

- Event handlers are used in Graphical User interface (GUI) applications to handle events such as button clicks and menu selections, raised by controls in the user interface.
- The C# event model is based on a **"publish-subscribe"** pattern in which a class (publisher) triggers an event, while another class (subscriber) receives that event.
- **Delegate :** An event is nothing but an encapsulated delegate. We can declare the delegate as shown below :

```
public delegate void someEvent();

public someEvent someEvent;
```

- **Declare :** To declare an event, use the event keyword before declaring a variable of delegate type, as below:

```
public delegate void someEvent();

public event someEvent someEvent;
```

- **Programming Example of Event handling in C#**

```
namespace Test
{
  //This is Subscriber Class
  class Program
  {
    static void Main(string[] args)
    {
      AddNum a = new AddNum();
      //Event gets binded with delegates
      a.ev_OddNumber += new
      AddNum.dg_OddNumber(EventMessage);
      a.Add();
      Console.Read();
    }
    //Delegates calls this method when event raised.
    static void EventMessage()
    {
Console.WriteLine("Event Executed: This is odd
    number!!!");
```

```
    }
  }
  //This is Publisher Class
  class AddNum
  {
    public delegate void dg_OddNumber(); //Declared Delegate
    public event dg_OddNumber ev_OddNumber ; //Declared Events

    public void Add()
    {
      int result;
      result = 10 + 11;
      Console.WriteLine(result.ToString());
      //Check if result is odd number then raise event
      if((result % 2 != 0) && (ev_OddNumber != null))
      {
        ev_OddNumber(); //Raised Event
      }
    }
  }
}
```

Fill in the Blanks for Mid Term Exam

Q.1 The three levels of concurrency in program are ____, ____ and _____.

Q.2 _____ is a shared variable or integer variable used to synchronize the progress of interacting processes.

Q.3 The semaphore works using two operations ___ and ___.

Q.4 The thread starts executing in _____ state.

Q.5 _____method resumes all the threads that are in suspended state.

Q.6 try, catch, throw are the keywords used in Java for handling _____.

Q.7 The _____ class is a superclass of all errors and exceptions in Java.

Q.8 _____is a notification that something specific has occurred, such as a mouse click on a graphical button.

Q.9 The C# event model is based on a _____ pattern.

Q.10 An Exception is another name for a _____error. [JNTU : Aug. 16]

UNIT - V

5 Functional, Logic Programming and Scripting Language

Part I : Functional Programming

5.1 Fundamental of Functional Programming

Q.1 What are functional programming languages ?

☞ [JNTU : Part A, March-16, Marks 3]

Ans. :

- Functional languages are those languages in which the basic building block is functions.
- In functional programming the programmer is concerned only with functionality and not memory related variable storage and assignment sequence.

Q.2 Explain the features of functional programming language.

☞ [JNTU : Part A, March-16, Marks 3]

Ans. :

1. The functional programming languages are modeled on the concept of mathematical functions and make use of only conditional expressions and recursion for the computational purpose.
2. The programs are constructed by building functional applications. That means, the values produced by one or more functions become the parameters to another functions.
3. The purest form of functional programming languages use neither variables nor assignment statements, although this is relaxed somewhat in most applied functional languages
4. The functional languages can be categorized in two types - Pure functional languages and Impure functional languages. The pure functional language support only functional paradigm. Example of pure functional language is ML, Haskell. The impure functional language can be used to write imperative style programs (For example C is an imperative style programming language). Example of impure functional programming language is LISP.

Q.3 Enlist the commonly used functional languages.

☞ [JNTU : Part A, Marks 2]

Ans. : LISP, Sisal, Haskell, ML, APL, Scheme are some commonly used functional languages.

Q.4 What are features of Haskell ?

☞ [JNTU : Part A, March-17, Marks 3]

Ans. :

1) Haskell is **purely functional programming language.**
2) It has **lazy evaluation** technique i.e. the expression is not getting evaluated until it is bound to a variable.
3) The language support **static type checking.** All the values in Haskell have a type which is determined at compile time. That means that a lot of possible type errors are caught at compile time.
4) The Haskell make use of functions to define the **offside rule** using equation. The lexical analyzer handles these offside rules efficiently.
5) It makes use of **List comprehension** technique to represent the list of elements. The list comprehension means building more specific set from the general set. For example, consider that we want a set of first 10 natural even numbers then mathematically it can be denoted as

$$S = \{2*x | x \in N, x \leq 10\}$$

This will give the list of elements as [2,4,6,8,10].

6) It supports the **pattern matching.** Pattern matching is a technique in which the pattern is specified and some data is confirmed to it. Then it is ensured if data behaves according to those patterns or not.

7) An **active and growing community** exists for this language.

Q.5 What are the applications of functional programming language ?

☞ [JNTU : Part A, Dec.-16, 17, Marks 3, Part B, Nov.-15, March-16, Marks 5]

Ans. :

1. The main domain of functional programming language is **Artificial Intelligence.** This language is used for building **expert systems, knowledge representation, machine learning, natural language processing, modeling speech and vision.**
2. Functional language is also used for **building mathematical software.**

Example -

LISP, Sisal, Haskell, ML, APL, Scheme are some commonly used functional languages.

Q.6 Explain about the fundamentals of functional programming languages.

☞ [JNTU : Part B, Dec.-17, Marks 5]

Ans. : Refer Q.1, Q.2 and Q.3.

Q.7 Explain various functional forms in functional programming.

☞ [JNTU : Part B, Marks 5]

Ans. :

- The functional languages can be categorized in two types - Pure functional languages and impure functional languages.
- The pure functional language support only functional paradigm.
- Example of pure functional language is ML, Haskell.
- The impure functional language can be used to write imperative style programs (For example C is a imperative style programming language).
- Example of impure functional programming language is LISP.

Q.8 Write a Haskell code for finding factorial of a given number.

☞ [JNTU : Part B, Marks 5]

Ans. :

```
factorial 0 = 1
factorial 1 = 1
factorial n = n*factorial(n-1)
```

Q.9 Write a Haskell code for finding Fibonacci number at a given position.

☞ [JNTU : Part B, Marks 5]

Ans. :

```
fib 0 = 1
fib 1 = 1
fib n |n>=2
      =fib(n-1)+fib(n-2)
```

5.2 Mathematical Functions

Q.10 Write about functions in ML and Haskell.

☞ [JNTU : Part A, May-18, Marks 2, Part B, Marks 5]

Ans. : Refer Q.17.

Q.11 Explain polymorphic typing.

☞ [JNTU : Part B, Marks 5]

Ans. :

- Type is a label that every expression has. It basically tells us in which category the expression fits. If the expression is "India" then it indicates string, if the expression has value 10 then it is of integer type and so on.
- Haskell supports the polymorphic types. An expression is said to be polymorphic if it has many types.
- In Haskell :: operator is used to denote the type. It can be read as "has type of" explicit type can be denoted with first letter in capital. For instance if the character is 'i' then its type is denoted as Char. The type can also be specified in a square bracket. The square brackets denote a list. So when there is [Char] then we read that as it being a list of characters.

- Functions also have types. When writing our own functions, we can choose to give them an explicit type declaration. It makes use of -> operator to denote the type.
- For example -

  Int→Int means the function takes the integer type parameter and returns an integer.

  In other words, it maps integer to integer.

For example consider following code

```
sumMe :: Int -> Int-> Int
sumMe a b = a + b
Output will be
sumMe 10 20
30
```

- Thus the feature polymorphic typing of Haskell allows to handle multiple types by a single function.

Q.12 Write about Haskell. Explain the functions in Haskell.

☞ **[JNTU : Part B, March-16, Marks 5]**

Ans. : Haskell : Refer Q.4.

Functions Used in Haskell :

- Since Haskell is a functional language, one would expect functions to play a major role, and indeed they do.
- It supports curried function.
- Currying is actually a process of transforming a function that takes multiple arguments into a function that takes just one argument and returns another function if any arguments are still needed.
- For example - sum 1 2 3 is applied then actually it is transformed as ((sum 1) 2) 3. Putting a space between two things is simply function application. The space is sort of like an operator and it has the highest precedence.
- In short, by sum 1 2 3, the function sum 1 is created , it returns a function. So when 2 is applied to that it creates a function that will take a parameter and add it by 3. When 3 is applied to that function the result will be 6. Hence the code for this function could be

```
sum :: (Num a) => a -> a -> a -> a
sum x y z = x+y+z
```

- Another example can be given by a map function of Haskell: Cosider a function add which can be defined as

```
add     ::Integer->Integer->Integer
add x y =x+y
```

Now the map function can be defined as

```
map          :: (a->b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Then we can show that -

```
map (add 1) [1,2,3] => [2,3,4]
```

This denotes that the function add can be passed as a parameter to map function. This is also an example of **higher order function.**

Q.13 What is a curried function ?

☞ **[JNTU : Part A, Marks 2]**

Ans. : A curried function is a function that takes multiple parameters.

Q.14 What is Referential Transparency ?

☞ **[JNTU : Part B, Marks 5]**

Ans. : Referential transparency means that an expression always **evaluates to the same result** in any context. For example if we write -

Let x=f(a) then,

in x+x we can be sure that the meaning of x+x is equivalent to f(a)+f(a) assuming that there is no intervening declaration of x.

In Haskell, due to referential transparency a function invocation can be **freely replaced with its return value without changing program's semantics.** The referential transparency is also said to be **side effect free.**

For example if we define some function say mypowfunction to calculate the x^y then it will always calculate power even if we supply different x and y values to it.

The code for such function will be

```
mypowfunction :: (Integer, Integer) -> Integer
mypowfunction (x,0) = 1
mypowfunction (x,y) = x * mypowfunction (x,y-1)
```

Output

```
*Main> mypowfunction(2,3)
8
*M
```

Q.15 What does partial evaluation mean ?

☞ [JNTU : Part A, Marks 2]

Ans. : The function is evaluated with actual parameters for one or more of the leftmost formal parameters.

Q.16 Describe the scoping rule in Common LISP, ML and Haskell.

☞ [JNTU : Part B, March-16, Marks 5]

Ans. :

- **LISP :**
  - **Lexical scope :** The names of parameters to a function normally are lexically scoped.
  - **Indefinite scope :** That means references may occur anywhere, in any program.
  - **Dynamic extent :** References may occur at any time in the interval between establishment of the entity and the explicit disestablishment of the entity. For example: the with-open-file construct opens a connection to a file and creates a stream object to represent the connection.
  - **Indefinite extent :** The entity continues to exist as long as the possibility of reference remains. For example- most Common Lisp data objects have indefinite extent.
- **ML :**
  - The scoping rule used in ML is called lexical scoping, because names refer to their nearest preceding lexical (textual) definition.
  - For example - if f() returns 12, then let val x = f() in x * x end is an expression that evaluates to 144, using a temporary variable named x to avoid calling f() twice.
- **Haskell :**
  - Haskell makes use of static scope rule.
  - For example -

```
let c = 21 in
let cTimes = \x   ->c * x  in
let c = 5 in
cTimes 2
```

- The result will be 42(21*2) as it uses static scope rule.

Q.17 Compare functions in ML and Haskell.

☞ [JNTU : Part B, Dec.-16, Marks 4]

Ans. :

Sr. No.	Haskell	ML
1.	Haskell is pure, without side effects or exceptions.	ML has mutable data structures such as references and arrays.
2.	Haskell has lazy evaluation	ML has strict evaluation.
3.	Haskell functions are curried.	ML uses tuple or record arguments for multi-argument functions.
4.	Haskell has minimal calculator style syntax.	ML syntax is heavier.
5.	For example - The function type can be described as f :: Bool -> Int f b = if b then 1 else 0	For example - The function type can be described as fun f (b:bool) : int = if b then 1 else 0

Q.18 Describe the two common mathematical functional forms that are provided by scheme.

☞ [JNTU : Part B, March-17, Marks 5]

Ans. : There are two functional forms -

1) **Functional composition :** Function composition is a functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the result of the second actual parameter function. For example -

Form : $h \equiv f \circ g$ which means $h(x) \equiv f(g(x))$ with $f(x) \equiv x + 2$ and $g(x) \equiv 3 * x$

In Scheme we can write it as

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
```

Now the functional composition of f and g can be written as follows:

```
(DEFINE (compose f g) (LAMBDA (x)(f (g x))))
```

It yields -

```
(3 * x) + 2
```

2) **apply-to-all :** Apply-to-all is a functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters. As an example of the use of map, suppose we want all of the elements of a list cubed. We can accomplish this with the following -

```
(map (LAMBDA (num) (* num num num)) '(2 3 5))
```

This call returns (8 27 125).

5.3 LISP

Q.19 What is LISP ? Enlist the features of LISP.

☞ [JNTU : Part A, Marks 3]

Ans. : LISP was invented by John McCarthy in 1958. It is high level programming language. It was first implement by Steve Russell on an IBM 704 computer.

Features of LISP :

1. It is based on expressions. The prefix notations are used to represent the expression.
2. It has advanced object oriented features.
3. It has got automatic garbage collection. Hence the programmer need not have to explicitly free the dynamically allocated memory.
4. A strong support for recursion in programs. Due to this feature this language is widely used and accepted in artificial intelligence.
5. All computation is performed by applying functions to arguments. Variable declarations are rarely used.

Q.20 What are the application areas of LISP ?

☞ [JNTU : Part A, Marks 3]

Ans. : Lisp is mainly used in -

1. Artificial intelligence robotics
2. Computer games
3. Pattern recognition
4. Real time embedded systems
5. List handling and processing
6. For creating applications for educational purpose.
7. For List handling and processing.

Q.21 How do we write expressions in LISP ?

Ans. :

- The expressions are represented using the operands and operators. Note that the prefix expression (that is operator operand1 operand2) is required for the evaluation of expression.

```
>(+ 10 20)
30
> (* (+ 2 3) (+ 5 6))
55
```

- If we do not want to evaluate an expression but want to print the entire expression then the quote operator or ' is used. For example

```
> (quote (+ 10 20))
(+ 10 20)
```

- The eval operator is used to evaluate an expression. For instance

```
> (eval (+ 10 20))
30
```

Q.22 Write about the operations that can be performed on atoms and lists in LISP.

☞ [JNTU : Part B, Dec.-16, Marks 6]

Ans. : Atoms : All expressions in LISP are made up of lists. The list is a linear arrangement of objects separated by blanks and surrounded by parentheses. The objects are made up of either atoms. Space separated elements of a list are known as ATOMS.

Some examples of atoms are -

name

123

Abc

a

c d

Lists : Lists are parenthesized collection of sublists or atoms. For example -

(a b (c d) e)

(10 20 30)

Q.23 What is the use of car functions LISP ?

☞ [JNTU : Part A, Marks 2]

Ans. : The car returns first element of a non empty list. For example

> (car '(1 2 3 4))

1

Q.24 What is the use of cdr functions LISP ?

☞ [JNTU : Part A, Marks 2]

Ans. : The cdr returns rest of the list after the first element is removed. It is pronounced as could-er.

> (cdr '(1 2 3 4))

(2 3 4)

Q.25 What is the use of cons function in LISP ?

☞ [JNTU : Part A, Marks 2]

Ans. : We can combine two lists using cons. The cons function takes two arguments and returns a new cons cell containing the two values.

> (cons 10 '(20 30 40))

(10 20 30 40)

Q.26 Explain the structure of List used in LISP. Also discuss the commonly used list manipulation functions.

☞ [JNTU : Part B, Marks 5]

Ans. : The list can be represented as follows

>(10 20 30)

The list is enclosed within the brackets. The elements in the list are separated by space. The list containing no item is called the empty list. On entering the empty list it will return the value NIL.

> ( )

NIL

Using the word list we can also create a list.

> (list 10 20 30)

(10 20 30)

List Manipulation Functions :

There are various functions that can be used for manipulation of lists.

1. Length :

The length of empty list is 0. We can find out the length of the list as follows -

> (length '(10 20 30 40 50))

5

We can also create a definition of length function as follows -

> (define (length x)

(cond ((null? x) 0)

(else (+ 1(length (cdr x)))))

2. Append :

The append is used to add the element in the list at the end. For example

> (append '(10 20 30) '(40 50))

(10 20 30 40 50)

We can also create a definition of append function as follows -

> (define (append x z)

(cond ((null? x) z)

(else (cons (car x) (append (cdr x) z)))))

3. Map :

The map is basically applied to a function. Using map we can extend the function from elements to list. Map returns a list of the results of applying the function to elements taken from each list. For example

```
>(map abs '(1 -2 3 -4 -5))
          (1 2 3 4 5)
> (map + '(1 2 3) (4 5 6))
  (5 7 9)
```

Thus map can handle more than one list.

4. Association Lists :

An association list or a list for short, records a mapping from keys to values. Using assoc we can obtain the most recent binding of key to value. For example - following assoc returns the appropriate values for the specified keys.

```
>(assoc 'one '((one 1) (two 2) (one 3)))
(ONE 1)
```

Another example

```
(assoc 'age '((name aa) (age 30) (salary 10000)))
(AGE 30)
```

Q.27 What are the differences between CONS, LIST and APPEND ?

☞ **[JNTU : Part A, March-17, Marks 2]**

Ans. : Refer Q.25 and Q.26.

Q.28 How to use conditional statements in LISP ?

☞ **[JNTU : Part B, Marks 5]**

Ans. : The conditions can be specified using **if.** The syntax of writing **if** is

(if(conditional expression) (true-expression) (false expression))

The **true-expression** part will be executed only if the condition specified in **condition expression** is true, otherwise the **false expression** part will be executed.

For example

```
>(if(> 10 20) 10)
NIL
```

As 10>20 is false and there is no false expression, NIL will be returned.

We will change the above expression as follows -

```
> (if (> 10 20) 10 20)
20
```

We can have **nested if statement.** The example is as shown below -

```
> (if(= 10 10) (if(> 5 1) 5 1))
5
```

If we want to display a series of expression on the if condition then these expressions are represented by a block. For this block **progn** is used. **progn** can take any number of expressions, and evaluates each of its expressions in order. **progn** then returns the value of the last expression.

```
> (if (> 20 10)
    |(Progn
    |(print "Hello")
    |(print "Welcome"))
```

True expression multiple statements are executed for true expression

```
    (print "Bye")
```

False expression

```
)
```

Output

```
"Hello"
"Welcome"
"Welcome"
```

Q.28 Write a short note on – quoting in LISP ?

☞ **[JNTU : Part B, Marks 5]**

Ans. : The quoting is used for evaluation of expression. Quoting is used to treat expressions as data. The symbol ' is used to represent the variable.

For example -

```
>pi
3.14159
```

But the

```
>(quote pi)
```

or

```
>'pi
```

will return

```
pi
```

Q.29 Explain how to define function in LISP ?

☞ [JNTU : Part B, Marks 5]

Ans. : The operator **defun** is used to define the procedure. The syntax of procedure writing in LISP is

defun (proc_name(argument1 argument2)
body of procedure)

For example - procedure for addition of two numbers is as follows -

```
> (defun mysum(a b)
(+ a b))
```

Now we can use this procedure the way we want.

```
> (mysum 2 3)
5
```

Q.30 Explain defparameter and setf in LISP.

☞ [JNTU : Part A, Marks 3]

Ans. :

- The variable can be defined using **defparameter**. For example

```
> (defparameter a 10)
A
```

When we type the variable name on the prompt then we get the value assigned to it. It is as shown below-

```
> a
10
```

- The **setf** is used to change the value of the variable. The setf will make global variables.

```
> (setf a 100) ;changing the value of a from 10 to
100
100
> a    ;simply testing whether a=100 or not
100
> (+ a 10) ;performing some operation with the
changed value
110
```

Q.31 Explain how to define lambda function in Lisp ?

Ans. : The lambda function is used to define a function which returns some value. The syntax of this function is

(lambda (parameters) Body)

For example -

```
>((lambda (x y) (* x y)) 10 20)
200
```

Similarly we can assign a value to a function by the return value of some function.

```
>(define (add n) (lambda (x) (+ x y)))
add
>((add 10) 20)
30
```

Here we have defined the function add which returns an object of type function.

Q.32 Write a procedure in LISP to calculate cube of number.

☞ [JNTU : Part A, Marks 3]

Ans. :

```
>(defun cube(a)
(* (* a a) a))
```

Output

```
> (cube 5)
125
```

Q.33 Write a procedure to display sum of squares.

☞ [JNTU : Part B, Marks 5]

Ans. :

```
> (defun square(x)
(* x x))
> (defun sum-of-square(start end)
(if(> start end)
0
(+ (square start) (sum-of-square(+ 1 start) end))
)
)
```

Output

```
> (sum-of-square 1 5)
55
```

Q.34 Write a procedure to display the number and its value after incrementing it by 10.

☞ [JNTU : Part B, Marks 5]

Ans. :

```
-(defun myfun(n) (print n) (+ n 10))
```

Output

```
> (myfun 10)
10
20
```

Q.35 What is the use of let construct ? Write a procedure to find out factorial of a given number.

☞ [JNTU : Part B, Marks 5]

Ans. : The syntax of Let construct is as follows -

(let ((var1 val1) (var2 val2).. (varn valn)), last_val)

Where var1, var2, ..varn are variable names and val1, val2, .. valn are the initial values assigned to the respective variables.

When let is executed, each variable is assigned the respective value and lastly the s-expression is evaluated. The value of the last expression evaluated is returned.

(let (x = 2, y = 3, z = x + y) print(x + y + z))

This will return 10

Factorial of a given number :

```
(defun factorial (n)
(let ((f 1)) ;initialising f=1
      (dotimes (x n)   ;repeating the following block for n times
       (setf f (* f  (1+ x))))     ;f=f*(x+1)
      f))              ; returns the value of f each time.
```

Output

```
> (factorial 5)
120
```

Q.36 Write a function that computes the sum of numbers using vectors in LISP.

☞ [JNTU : Part B, Dec.-17, Marks 5]

Ans. :

```
(defun addition-v2 (v1 v2)
 (if(< (length v1) (length v2))
  (do ((x 0 (+ 1 x)))
  ((>= x (length v1)) v2)
   (setf (aref v2 x) (+ (aref v1 x) (aref v2 x))))))
```

Here v1 and v2 are the vectors such as

v1(vector 10,20,30)

v2(vector 11,22,33)

Q.37 Explain the basic primitives of LISP. Give suitable examples.

☞ [JNTU : Part B, Nov 15, Marks 5]

Ans. : Refer Q.22.

5.4 Support for Functional Programming in Primarily Imperative Languages

Q.38 Explain with example how do the primarily imperative languages support functional programming ?

☞ [JNTU : Part B, Marks 5]

Ans. :

- Support for functional programming is increasingly creeping into imperative languages.
- For example - JavaScript leaves the name out of a function definition.

```
function name (formal-parameters) {
body
}
```

- C# supports lambda expressions that have a different syntax than that of C# functions. For example, we could have the following:

```
i => (i % 2) == 0
```

This lambda expression returns a Boolean value depending on whether the given parameter (i) is even (true) or odd (false).

- Python's lambda expressions define simple one-statement anonymous functions that can have more than one parameter. For example -

```
lambda a, b : a + b
```

Note that the formal parameters are separated from function body by a colon.

- Similarly consider map function in Python as

```
map(lambda x: x ** 3, [2, 4, 5, 6])
```

will return [8,64,125,216]

5.5 Comparison of Functional and Imperative Languages

Q.39 Explain the differences between imperative and functional languages.

☞ [JNTU : Part B, Nov.-15, March-16, 17, Dec.-16, May-18, Marks 5]

Ans. :

Imperative language	Functional language
An imperative language uses a sequence of statements to determine how to reach a certain goal. These statements help to change the state of the program as they get executed.	The functional language treats everything as a function. They does not have a state associated with them.
The values can be assigned to some variable in imperative languages.	The values are returned from the functions depending upon the input given to that function.
The order of execution is very important in imperative languages.	The order of execution is of low importance in functional languages.
The control flow is manipulated by conditional statements, loops, function or method calls.	The control flow is manipulated by functional calls including recursion.
Examples of imperative languages are C, Java, Python, Ruby, JavaScript and so on.	Examples of functional languages are LISP, Haskell, ML and so on.

Part II : Logic Programming

5.6 An Overview of Logic Programming

Q.40 What is logic programming ?

☞ [JNTU : Part A, Marks 2]

Ans. : • Logic programming is a programming paradigm in which the set of sentences are written in

logical form. The logic programs consists of facts and rules about some problem domain.

- Logic programming can be viewed as **controlled deduction.**
- An important concept in logic programming is the separation of programs into their logic component and their **control component.**
- Kowalski illustrates the logic programming by following equation

Algorithm = Logic + Control

- Where "Logic" represents a logic program in the form of facts and rules. The "Control" represents how the algorithm can be implemented by applying the rules in particular order.
- The concept of logic programming is linked up with a language called **prolog. Prolog** is acronym of PROgramming in **LOGic.**

5.7 Basic Elements of Prolog

Q.41 Describe about the basic elements of Prolog.

☞ **[JNTU : Part B, March-15, Marks 5]**

Ans. : Relations and Queries :

- Prolog makes use of relations instead of using functions. Relations are more flexible and treats the arguments and results uniformly.
- For example - Consider following list

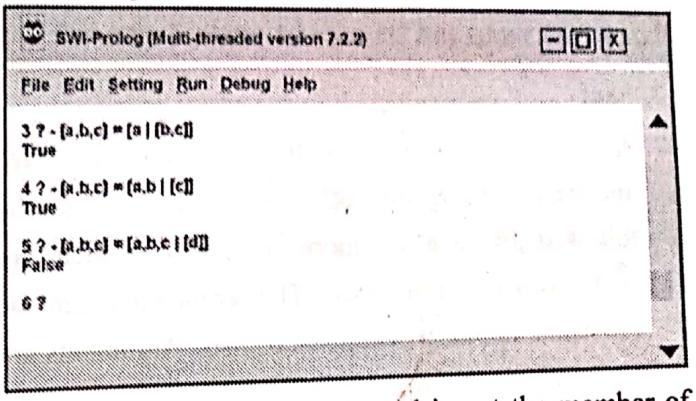
[a,b,c]

- As we know that the list can be considered as [H|T]. That means in the list H i.e head occurs first and then comes T i.e. tail. Hence prolog will treat [a,b,c] as

[a,b,c] =[a | [b,c]].

Queries :

- In prolog, the simple queries can be created to check whether particular tuple belogs to a relation.
- In below given execution, the first two queries represent the simple H and T relation between the members of the list. Hence the answer turn out to be true or yes.



- But in the third query since d is not the member of the list [a,b,c] the answer turn out to be false.

Terms, Goals, Facts and Rules

- A Prolog program consists of a number of clauses. Each clause is either a fact or a rule. After a Prolog program is loaded (or consulted) in a Prolog interpreter, users can submit goals or queries, and the Prolog intepreter will give results (answers) according to the facts and rules.

1) Terms :

Prolog term is a constant, a variable, or a structure. A constant is either an atom or an integer. Atoms are the symbolic values of Prolog.

A variable is any string of letters, digits, and underscores that begins with an uppercase letter or an underscore (_). Variables are not bound to types by declarations. The binding of a value, and thus a type, to a variable is called an instantiation.

2) Goals :

A goal is a statement starting with a predicate and probably followed by its arguments. In a valid goal, the predicate must have appeared in at least one fact or rule in the consulted program, and the number of arguments in the goal must be the same as that appears in the consulted program. Also, all the arguemnts (if any) are constants.

The purpose of submitting a goal is to find out whether the statement represented by the goal is true according to the knowledge database (i.e. the facts and rules in the consulted program). This is similar to proving a hypothesis - the goal being the hypothesis, the facts being the axioms and the rules being the theorems.

3) Fact

- A fact must start with a predicate (which is an atom) and end with a fullstop. The predicate may be followed by one or more arguments which are enclosed by parentheses. The arguments can be constants, numbers, variables or lists. Arguments are separated by commas.
- The syntax of fact is

  pred(arg1, arg2, ... argN).

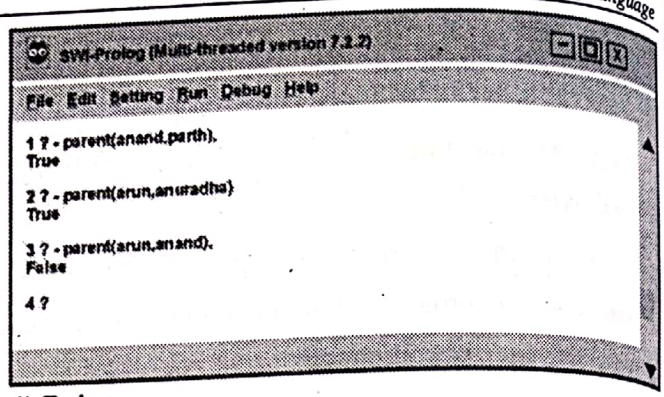
  Where

  pred is the name of the predicate and arg1, ...argN are the arguments

For example -

```
man(anand).
man(arun).

woman(anuradha).
woman(jayashree).

parent(anand,parth). % means anand is parent of parth
Parent(anuradha,parth).
parent(arun,anuradha).
parent(jayashree,anuradha).
```

The result as either true or false depending upon the facts. It is as follows -



4) Rule :

- A rule can be viewed as an extension of a fact with added conditions that also have to be satisfied for it to be true. It consists of two parts. A rule is no more than a stored query. Its syntax is

  head :- body.

  where

  head is a predicate definition (just like a fact),

  :- is the neck symbol, sometimes read as "if"

  body is one or more goals (a query)

For example -

```
father(F,C):-man(F),parent(F,C).
```

Example using Fact and Rule

```
/* Facts */

man(anand).
man(arun).
woman(anuradha).
woman(jayashree).

parent(anand,parth). % means anand is parent of parth
parent(anuradha,parth).
parent(arun,anuradha).
parent(jayashree,anuradha).

/* A general rule */
father(F,C):-man(F),parent(F,C).
mother(M,C):-woman(M),parent(M,C).
```

If we query

```
-? father(X,parth).
```

The above query can be read who is father of parth? The answer is X=anand is a father of parth.

Q.42 Write down the use of predicate calculus in logic programming.

☞ [JNTU : Part B, Marks 5]

Ans. : Predicate calculus is a fundamental notation for representing and reasoning with logical statements. It extends propositional calculus by introducing the quantifiers, and by allowing predicates and functions of any number of variables.

1) **Propositional symbols :** P, Q, R, S, T, ... denote propositions, or statements about the world that maybe either true or false, such as "Sun is bright" or "India is a country".

2) **Truth symbols :** true, false.

3) **Connectives :** ∧ (and), ∨ (or), ¬ (not), ⇒, ≡

4) **Propositional calculus sentences :** Every propositional symbol and truth symbol is a sentence.

For example The negation of a sentence is a sentence. Denoted as ¬ P

Any two sentences connected by one of {∧, ∨,⇒,=} are a sentence. For instance P ∨ ¬ Q ⇒ R.

5) **Conjucts and disjuncts :** In expressions of the form P ∧ Q, P and Q are called conjuncts.

In P ∨ Q, P and Q are called disjuncts.

6) **Symbols :** The symbols ( ) and [ ] are used to group symbols into sub-expressions and so control their order of evaluation and meaning. (P ∨ ¬ Q) = R and P ∨ (¬ Q = R) are different.

Q.43 Explain the Terms and Goal Statements in Prolog.

☞ [JNTU : Part B, Nov.-15, Marks 5]

Ans. : Refer Q.41.

Q.44 Discuss the features of Prolog.

☞ [JNTU : Part B, Marks 5]

Ans. : The important features of prolog are -

1) **Relations :** Prolog makes use of relations instead of using functions. Relations are more flexible and treats the arguments and results uniformly.

2) **Unification :** Unification is a derivation of new rule from given rule by binding of variables.

3) **Backtracking :** When a task fails, prolog traces backwards and tries to satisfy previous task.

4) **Recursion :** Recursion is the basis for any search in program.

5) **Cuts :** Cut is used in prolog to cut of the unexplored part of the rule searching. The cut makes the computations more efficient by eliminating futile searching and backtracking.

5.8 Deficiencies of Prolog

Q.45 Discuss the deficiencies of Prolog.

☞ [JNTU : Part B, Marks 5]

Ans. : Although Prolog is a logical programming language, it is not perfect logic programming language. There are some problems associated with Prolog and those are -

1) Resolution order control :

In a pure logic programming environment, the order of attempted matches is nondeterministic and all matches would be attempted concurrently

2) The closed-world assumption :

The only knowledge is what is in the database. In Prolog, things can be proved true or false using the information present in the database. When the system receives a query and the database does not have information to prove the query absolutely, the query is assumed to be false. Prolog can prove that a given goal is true, but it cannot prove that a given goal is false. It simply assumes that, because it cannot prove a goal true, the goal must be false.

3) The negation problem :

Anything not stated in the database is assumed to be false.

4) Intrinsic limitations :

It is easy to state a sort process in logic, but difficult to actually do as it doesn't know how to sort

5.9 Applications of Logic Programming

Q.46 What is meant by logic programming ? Explain different types of applications of logic programming.

☞ [JNTU : Part A, March-16, May-18, Marks 2, Part B, Dec.-16, Marks 5]

Ans. : Logic Programming : Refer Q.40.

Applications :

1) Prolog is used in **artificial intelligence and expert system** to solve complex problems.
2) It is used on **pattern matching algorithm.**
3) It is used to develop systems based on **object oriented concepts.**
4) Prolog is used to build **decision support systems** that aid organizations in decision-making For example- decision systems for medical diagnoses.

Part III : Scripting Language

5.10 Pragmatics

Q.47 What are scripting languages ?

☞ [JNTU : Part A, Marks 2]

Ans. : A scripting language is a programming language designed for integrating and communicating with other programming languages. Some of the most widely used scripting languages are JavaScript, VBScript, PHP, Perl, Python, Ruby, ASP and Tcl.

Scripting languages, which can be embedded within HTML, commonly are used to add functionality to a Web page, such as different menu styles or graphic displays or to serve dynamic advertisements.

Sr. No.	Scripting Language	Programming Language
1.	Scripting languages are those that use interpreter.	Programming languages are those that use compiler.
2.	The scripting languages run inside another program. For example scripting languages execute within the web browser.	These run independent of parent program or exterior.
3.	Designed to make coding fast and simple.	Designed to give full usage of a language.
4.	It does not create any .exe file.	It creates .exe file.
5.	Scripts are just piece of code.	Programming is making full code of program.
6.	These are easy to learn to write.	These are complex to learn.

5.11 Key Concepts

Q.48 Write advantages of scripting languages.

☞ [JNTU : Part A, Dec.-16, Marks 2]

Ans. : Following are some advantages of scripting languages -

- Scripting languages easy to learn and use
- They minimum programming knowledge or experience required
- These languages allow complex tasks to be performed in relatively few steps
- They allow simple creation and editing in a variety of text editors
- The scripting languages allow the addition of dynamic and interactive activities to web pages
- Editing and running code is fast in scripting languages.

Q.49 Explain common characteristics of the scripting languages.

☞ [JNTU : Part B, May-18, Marks 10]

Ans. : Various characteristics of scripting languages are -

1) These languages are kept in source form and not in compiled form.
2) These are dynamically typed languages.

3) These languages are not concerned for underlying machines.
4) The scripting languages have the code that doesn't stand alone as full-fledged executables, but rather is run within some container (like JavaScript within a web browser).
5) These are extensible.
6) The scripting languages are application specific, that means can be developed for gaming, networking, and data processing and so on.
7) They have an interactive shell for typing in and executing fragments.
8) The scripting languages re-execute changed code instantly, without the hideous overhead of explicit recompilation and deployment.
9) The scripting languages support event handling such as mouse clicks, keyboard inputs.
10) These languages emphasize flexibility and rapid development.

5.12 Case Study : Python

Q.50 Enlist any five features of Python.

☞ [JNTU : Part A, Marks 5]

Ans. : Following are features of python :

1) Python is **free and open source** programming language. The software can be downloaded and used freely.
2) It **high level programming language.**
3) It is **simple and easy** to learn.
4) It is **portable.** That means python programs can be executed on various platforms without altering them.
5) It is an **object oriented programming** language.
6) It is **rich set of functionality** available in its huge standard library.
7) Python has a powerful set of **built-in data types** and **easy-to-use control constructs.**

Q.51 What are data types supported in Python ?

☞ [JNTU : Part A, Nov.-15, Marks 3; Part B, Dec.-17, May-18, Marks 5]

Ans. :

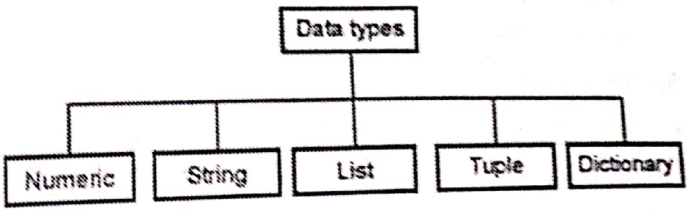


Fig. Q. 51.1 Data types in Python

1) **Numeric :** Python numeric data type is used to hold numeric values like;
- int - holds signed integers of non-limited length.
- long- holds long integers.
- float- holds floating precision numbers and it's accurate upto 15 decimal places.
- complex - holds complex numbers.
- **For example -**

```
a=10
b=12.22
c=10+8j
```

2) **String :**
- String is a collection of characters.
- In Python, we can use single quote, double quote or triple quote to define a string.
- **For example -**

```
Str1="My"
Str2="Python"
```

3) **List :**
- It is similar to array in C or C++ but it can simultaneously hold different types of data in list.
- It is basically an ordered sequence of some data written using square brackets ([]) and commas (,).
- **For example -**

```
a=[10,20,30,40]
b=['Sun','Mon','Tue','Wed']
```

4) **Tuple :**
- Tuple is a collection of elements and it is similar to the List. But the items of the tuple are separated by comma and the elements are enclosed in ( ) parenthesis.

- Tuples are immutable or read-only. That means we can not modify the size of tuple or we cannot change the value of items of the tuple.
- **For example -**

```
a=(10,20,30,40)
```

5) Dictionary :

- Dictionary is a collection of elements in the form of **key:value** pair.
- The elements of dictionary are present in the curly brackets.
- **For example -**

```
a={1:'Red',2:'Blue',3:'Green'}
```

Q.52 Explain the use of input operation with example.

☞ [JNTU : Part B, Marks 5]

Ans. :

- In python it is possible to input the data using keyboard.
- For that purpose, the function **input()** is used.
- **Syntax**

```
input([prompt])
```

where prompt is the string we wish to display on the screen. It is optional.

For example - Following python code is for addition of two numbers that makes use of input() operation.

```
print("Enter first number")
a=int(input())
print("Enter second number")
b=int(input())
c=a+b
print("Addition of two numbers is: ")
print(c)
```

Q.53 What is the significance of indentation in Python ?

☞ [JNTU : Part A, Marks 3]

Ans. :

- Leading white space at the beginning of the logical line is called indentation.
- **Python programs get structured through indentation**, i.e. code blocks are defined by their indentation.
- All statements with the same distance to the right belong to the same block of code, i.e. the statements within a block line up vertically.
- If a block has to be more deeply nested, it is simply indented further to the right.
- **For example -**

```
age = int(input('How old are you? '))
if age <= 2:
    print(' Infant')
```

Here the line is indented that means this statement will execute only if the if statement is true

Q.54 Discuss the storage and control in Python.

☞ [JNTU : Part A, May-18, Marks 3, Part B, March-17, Marks 10]

Ans. : 1) Selection Statement :

if-else Statement

- The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

Syntax

```
if condition :
        statement
else :
        statement
```

- If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.

For example -

```
print("Enter value of n")
n=int(input())
if n%2==0:
    print("Even Number")
else :
    print("Odd Number")
```

if-elif-else statement

- Sometimes there are more than two possibilities. These possibilities can be expressed using chained conditions. The syntax for this is as follows

```
if condition:
    Statement
elif condition:
    Statement
...
else:
    Statement
```

- The chained conditional execution will be such that each condition is checked in order.
- The **elif** is basically abbreviation of **else if.**
- There is no limit on the number of elif statements.
- If there is **else** clause then it should be at the end.

While loop :

- The **while** statement is popularly used for representing iteration.
- **Syntax**

```
while test_condition :
body of while
```

For example -

```
while i<=10:
    i=i+1
```

- The body of a while contains the statement which will change the value of the variable used in the test condition. Hence finally after performing definite number of iterations, the test condition gets false and the control exits the while loop.
- If the condition never gets false, then the while body executes for infinite times. Then in this case, such while loop is called **infinite loop.**

For loop

- The for loop is another popular way of using iteration. The syntax of for loop is
- **Syntax**

```
for variable in sequence:
    Body of for loop
```

- **Example -**

```
for val in numbers:
    val=val+1
```

- The variable takes the value of the item inside the sequence on each iteration.
- Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Fill in the Blanks for Mid Term Exam

Q.1 _____ are those languages in which the basic building block is functions.

Q.2 LISP, Sisal, Haskell, ML, APL, scheme are some commonly used _____ languages.

Q.3 The _____ technique is a technique in which the expression is not getting evaluated until it is bound to a variable.

Q.4 ____ function takes multiple parameters in functional programming.

Q.5 _____means that an expression always evaluates to the same result in any context.

Q.6 The _____ programs consists of facts and rules about some problem domain.

Q.7 Prolog is a based in ____ programming.

Q.8 Javascript, PHP,Perl, VBScript are examples of ____ programing.

Q.9 The principles that a function can always be replaced by its value without changing the meaning is called _____.

Multiple Choice Questions for Mid Term Exam

Q.1 Programming paradigm is also known as _____.

[a] programming methodology
[b] program
[c] software systems
[d] none of these

b) *Explain about the fundamentals of functional programming languages.*

MAY - 2018 [R15]

Principles of Programming Languages (125 AN)

Solved Paper
B.Tech. III[rd] Year
Sem-I (CSE)

Time : 3 Hours] **[Max. Marks : 7**

Part - A : 25 Marks

Q.1 **a)** *In what way do operational semantics differ from denotational semantics ?* **(Refer Q.51 of Chapter - 1)** [2]

b) *A programming languages can be complied or interpreted. Give relative advantages and disadvantage of compilation and interpretation.* **(Refer Q.20 of Chapter - 1)** [3]

c) *Define Union type.* **(Refer Q.37 of Chapter - 2)** [2]

d) *How pointers are represented and used in C and C++ ?* **(Refer Q.43 of Chapter - 2)** [3]

e) *What are the different categories of subprograms ?* **(Refer Q.2 of Chapter - 3)** [2]

f) *Give a brief note on scope and lifetime of vatiable.* **(Refer Q.8 of Chapter - 2)** [3]

g) *What is the purpose of the Java finally clause ?* **(Refer Q.64 of Chapter - 4)** [2]

h) *Explain the applications of logic programming ?* **(Refer Q.46 of Chapter - 5)** [3]

i) *Write about functions in ML and Haskell.* **(Refer Q.10 of Chapter - 5)** [2]

j) *Discuss the storage and control in python.* **(Refer Q.54 of Chapter - 5)** [3]

Part - B : 50 Marks

Q.2 a) *What is need for study of programming languages ? Explain.* (Refer Q.1 of Chapter - 1)

b) *Describe the programming paradigms.* (Refer Q.7 of Chapter - 1) [5+5]

OR

Q.3 a) *Define Ambiguity. Write short notes on attributed grammars.* (Refer Q.48 of Chapter - 1)

b) *Explain briefly about the Denotational semantics with example.* (Refer Q.53 of Chapter - 1) [5+5]

Q.4 *What is an array ? Explain about various array operations with an example.* (Refer Q.30 of Chapter - 2) [10]

OR

Q.5 a) *Explain the syntax of Ada union type.* (Refer Q.37 of Chapter - 2)

b) *List and explain the design issues for all iterative control statements.* (Refer Q.78 of Chapter - 2) [5+5]

Q.6 *Describe the design issues of subprograms and operations.* (Refer Q.8 of Chapter - 3) [10]

OR

Q.7 a) *Briefly explain in the static and dynamic scope.* (Refer Q.46 of Chapter - 3)

b) *What are the general subprogram characteristics ? Explain* (Refer Q.3 of Chapter - 3) [5+5]

Q.8 *Explain how binary semaphore provides solution to the competition synchronization problem with example pseudo code.* (Refer Q.40 of Chapter - 4) [10]

OR

Q.9 *What is exception handling ? Explain about exception handling in Java.* (Refer Q.63 of Chapter - 4) [10]

Q.10 a) *What is the difference between functional and imperative languages ?* (Refer Q.39 of Chapter - 5)

b) *Discuss the data types supported in Python.* (Refer Q.51 of Chapter - 5) [5+5]

OR

Q.11 *Explain the common characteristics of the scripting languages.* (Refer Q.49 of Chapter - 5) [10]

END ...