Principles of Programming Languages [PPL]

PreRequisites:-

1. A Course on "Mathematical Foundations of Computer Science"
2. A course on "Computer programming and Data structures".

Course Objectives:-

1) Introduction to important paradigms of Programming Languages.
2) To provide Conceptual understanding of high-Level Language design implementation.
3) Topics include programming paradigms, Syntax and semantics, data types, expressions and statements, subprograms and blocks, abstract data types, Concurrency, functional and Logic programming Languages and Scripting Languages.
4) To understand the concepts of (object-oriented) OO Languages.
5) To introduce the principles and techniques involved in design and implementation of modern programming Languages and concepts of exception handling.
6) To introduce the concepts of ADT (Abstract Data Type) and OOP (object-oriented programming) for software development.

Course outcomes:-

1. Ability to express syntax and semantics in formal notation.
2. Ability to apply suitable programming Paradigm for the application.
3. Ability to compare the features of various programming Languages.
4. Ability to understand the programming Paradigms of modern programming Languages.
5. Able to understand the concepts of ADT and OOP.
6. Ability to program in different languages Paradigms and evalute their relative benefits.

SYLLABUS :-

Unit 1 - (A) preliminary Concepts

(B) Syntax and Semantics

Unit - 2 - (A) Names, Bindings and Scopes

(B) Data Types

(C) Expressions and Statements

(D) Control structures.

Unit - 3 - (A) Subprograms and Blocks

(B) Implementing Subprograms

(C) Abstract Data types

Unit - 4 - Concurrency

Unit - 5 - (A) Functional programming Languages

(B) Logic programming Languages

(C) Scripting Languages.

Unit-I -(A)

#Preliminary Concepts

(1) Reasons for studying Concepts of programming Languages :-

The Following are benefits of studying Language Concepts:

1. Increased Capacity to Express Ideas.

2. Improves Background for choosing appropriate Language

*3. Increased ability to learn new Languages.

4. Better Understanding of the significance of implementation.

5. Overall advancement of Computing.

(1) ⇒ Those with a limited grasp of natural Languages are limited in the complexity of their thoughts, particularly in depth of abstraction.

⇒ programers can increase the range of their software development thought process by Learning new language constructs.

(2) ⇒ programers were familar with the other Languages available, and especially the

Particular features in those Languages, they could be in a better position to make in formed Language choicely.

③⇒ This makes software Development an exiciting profession, but if also means that continueous learning is essential.

④⇒ Certain kinds of program bugs can only be found and fixed by a programmer who keeps some related implementation Details.

⑤⇒ Finally, there is a Global View of Computing that can justify the study of Programming Language Concepts.

#2) Programming Domains :-
(Area)

In this Section, we briefly discuss a few of the areas of computer applications and their Associated Languages.

1. Scientific Applications
2. Business Applications
3. Artificial Intelligence
4. Systems programming
5. Scripting Languages
6. Special - purpose Languages.

① Scientific Applications :-

→ The first Digital Computers, which appeared in 1940's, used in Scientific Applications.

→ Scientific Applications have simple Data structures but requires large number of Floating-point Arithmatic Compatations.

Data structure Ex :- Arrays & Matrices

Control statements Ex :- Loops & Selections.

Ex. of the Scientific Applications :-

① For-Tran

② ALGOL-60

② Business Applications :-

→ The use of Computers for Business Applications began in the 1950's.

→ Special Computers were Developed for this purpose, along with special Languages.

→ The first successful High-Level languages for Business was COBOL.

Ex :- COBOL

(C)ommon (B)usiness (O)riented (L)anguages.

③ Artificial Intelligence:-

→ (ROBO) - Artificial Intelligence (AI) is Broad Area of Computer Applications.

Ex:- First - AI Languages :- ① LISP

② pro-log (PROgramming in LOGic)

④ System Programming:-

→ The Operation System (OS) and all of the programming support tools of a Computer Stystem are collectively known as its Systems software.

Ex:- C - programming Language.

⑤ Scripting Languages:-

→ Scripting Languages have Evolved over the past 25 - years.

→ Early Scripting Languages were used by putting a list of Commands is called a script, in a File to be Executed.

Ex:- ① Java - Script

② PHP (Hyper - Text Pre - processor)

③ HTML (Hyper - Text Mark-up Language)

⑥ Special - purpose Languages:-

→ A Host of Special - purpose Languages; have appeared over the past 40 - Years.

→ These are Narrow Applicability and the difficulty of comparing them with other Languages.

#3) Language Evalution Criteria:-

⇒ The Underlying Concepts of the Various Constructors and capabilities of programming Languages.

1. Readability
2. Writability
3. Reliability
4. Cost

1) Readability:-

→ perhaps one of the most important Criteria for Judging a programming Language is the Ease with which Programmings can be Read and understand.

2) Writability:- (Type)

→ Writability (Type) is a measure of how easily a language can be used to Create programs for a choosen problem Domain.

⇒ Most of the Language characteristics that affect Readability also affect Writability (Type).

3) Reliability:-

⇒ A program is said to be "Reliable", if it performs to its specifications under All conditions.

⇒ Type checking is Simply testing for type Errors in a given program.

4) Cost:-(Money)

⇒ The ultimate Total Cost of a programming Language is a Function.

⇒ First there is the COST of Training Programmers to use the Language.

⇒ The Cost of writing (Typing) programs in the programming Language.

⇒ The Cost of Computing programs in the Language.

#4) Influences on Language Design:-

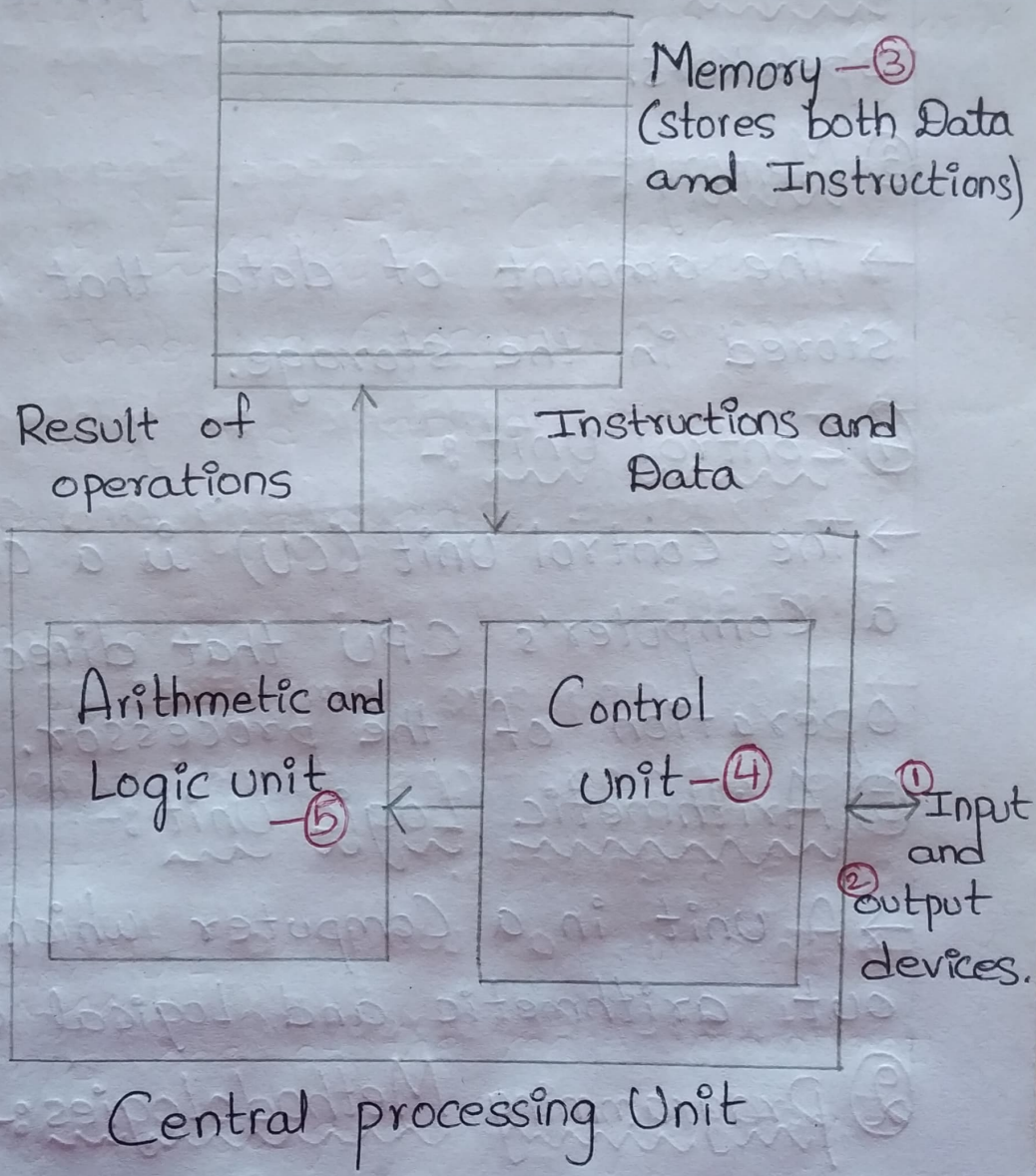
The Most Important Function of "INFLUENCES ON LANGUAGE DESIGN" are

1) Computer Architecture
2) programming Methodologies.

① Computer Architecture :-

⇒ The Basic Architecture of Computer had effect on Language Design.

Fig :- Computer Architecture



① Input Devices :-

→ A piece of equipment used to provide data and control signals to a computer is known as input device.

Examples: Keyboards, mouse, joysticks, scanners, digital cameras and microphones.

② output devices:-

→ An output device is any piece of computer hardware equipment which converts information into human-readable form.

Examples: Monitor, printer, plotters and Speakers.

③ Memory:-

→ The amount of data that can be stored in the storage.

④ Control unit:-

→ The control unit (CU) is a Component of a Computer's CPU that directs the operation of the processor.

⑤ Arithmetic Logic unit:-

→ A unit in a Computer which carries out arithmetic and Logical operations.

② Programming Methodologies:-

⇒ 1940's – Computer Designed

⇒ 1950's – Business Apps (COBOL)

→ The late 1960's and early 1970's brought an analysis, begin in large part by the structured programming movement.

⇒ An important Reason for the research was the shift in the major cost of Computing from Hardware & Software.

⇒ The Latest step in the Evolution of Data-oriented software development which began in the early 1980's is object-oriented programming Language.

Ex:- ① Small talk (Gold berg & Robson, 1989)
② Ada-95 (ARM, 1995)
③ C++
④ Java (James Goliz)

#5) Language Categories:-

⇒ Programming Languages are often catagorized into Four types.

They are
① Imperative
② Functional
③ Logic
④ object-oriented.

Imperative:-

→ The Most popular object-oriented Languages grew out of Imperative Languages.

→ Imperative Languages are composed of step-by-step instructions for the computer.

Ex:- Fortran, C, C++ & Java, etc....

→ The most popular Visual language is Visual BASIC (VB). [Schneiber - 1999].

→ Which is now being Replaced by Visual BASIC.NET.

→ Capability of Drop-and-Drop Generation of Code Segment.

Functional:-

→ Functional Language programs are constructed by applying and composing Functions.

→ This Language empazies on expressions and declarations rather than execution of statements.

→ This is a way of thinking about software construction by creating pure Functions.

Ex:- LISP - LISt Processing,

Phython, Erlang etc-----

Logic:-

→ In Logic Language the program statements express facts and rules about problems within a system of

formal Logic.

→ In this Language, Logic is used to represent knowledge and inference is used to manipulate it.

Example :- PROLOG (programming in Logic)

Object-Oriented :-

→ object-oriented Language is based on the concept of "objects", which can contain data and code: data in the form of fields, andcode in the form of procedures.

→ A feature of objects is that an object's own procedures can access and often modify the data fields of itself.

Ex :- C, C++ (OOPL-oops), Java and Ruby.

#6) Language Design Trade-off :-

* Cost more to ensure Greater Reliability [More Ability/speed/accurate]

Ex :- "C" - Some topic are not there [Index Array Ranges]

→ So executes fast, but it is not so reliable.

→ Other side "JAVA" checks all references to array elements.

⇒ JAVA Executes slower (Less-Time Difference), but it is more Reliable (current Accurate).

#7 Programming Language IMPLEMENTATION Methods :-

→ Two of the primary components of a computer are its main in this implementation. They are ① Internal Memory
② processor

Internal Memory :-

* It is used to store programs (code & Data)

Ex :- ① ROM :- Read only Memory
* It is primary Memory.
* It is a non-volatile storage medium meant for permanent storage.

② RAM :- Random Access Memory.
* It is a volatile memory that is data in RAM are lost when compoter is turned off.
* It is a Computer's short-term Memory.

Processor :-

* It is a collection of circuits that provides a Realization of a set of primitive operations (or) Machine Instructions such as those for Arithmetic & Logic operations.

⇒ The machine Language of the computer is its set of Macro Instructions.

Example :-

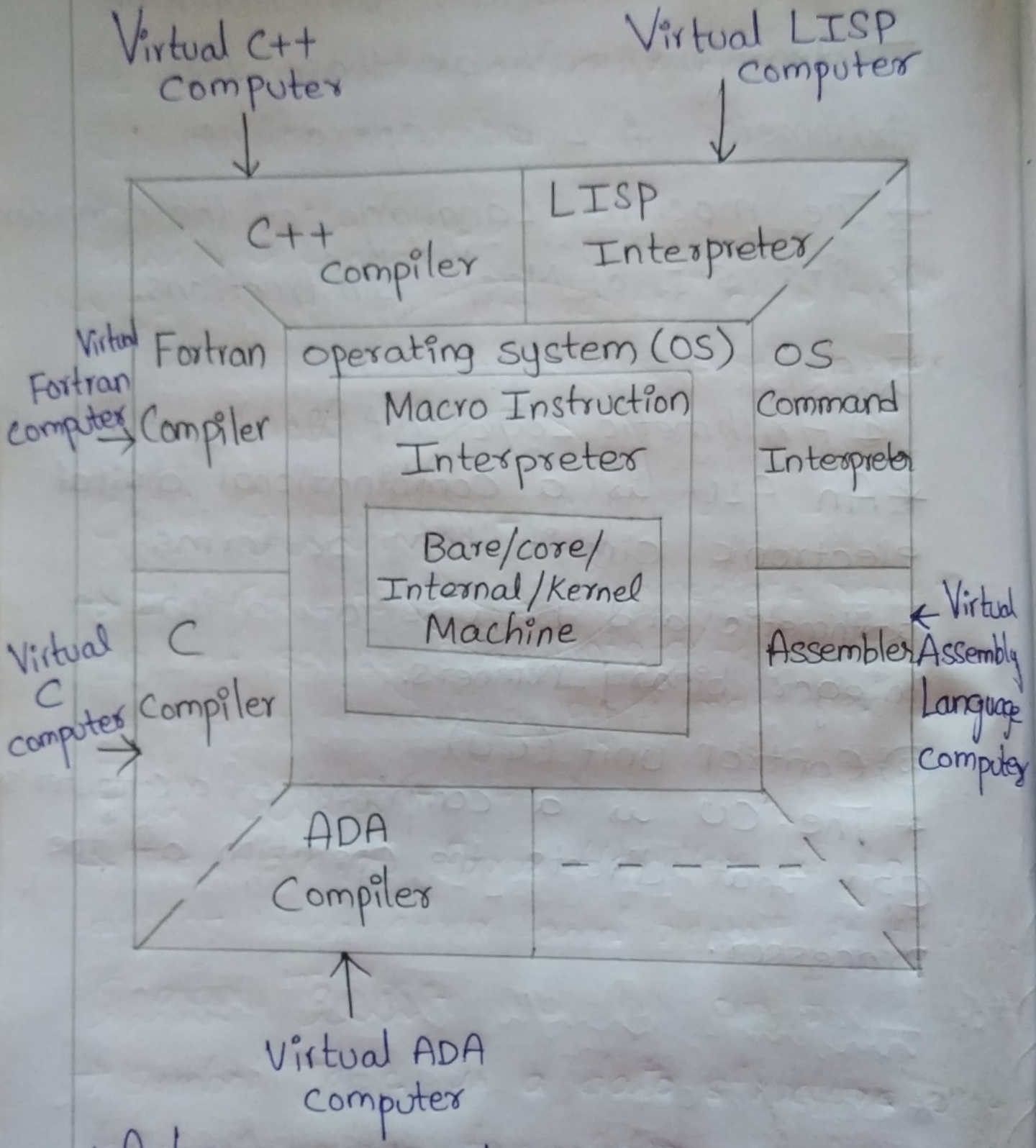
① Arithmetic Logic unit (ALU)

* An ALU is a combinational digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers.

② Control unit (CU)

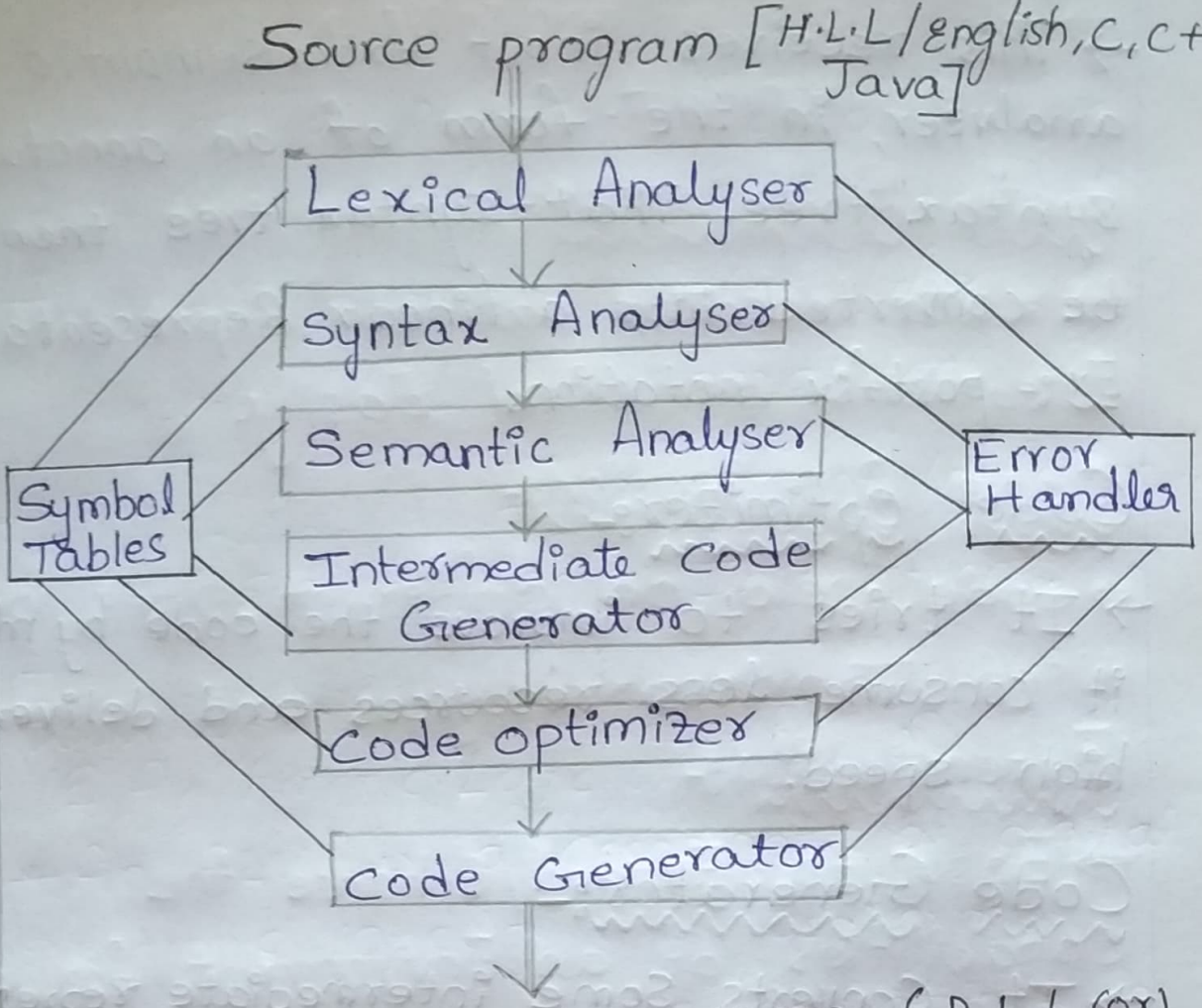
* The CU is a component of a Computer's CPU that directs the operation of the processor.

Layered View Interface of Computer:



→A Language Implementation system Cannot be the only Software on a computer.

The Compilation processes:-



Lexical Analyzer:-

→ It breakes these syntaxes into a series of tokens by removing any whitespace or comments in the source code.

Syntax Analyzer:-

→ The syntax Analyzer analyzes the token stream aganist the production rules to detect any errors in the code.

Semantic Analyzer:-

→ It analyzes the context in the text structure to accuratly disambiguate the proper meaning of words.

Intermediate Code Generator:-

→ This receives input from semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear Representation. Ex:- postfix notation.

Code Optimizer:-

→ It tries to improve the code by making it consume less resources and deliver high speed.

Code Generator:-

→ It Converts some intermediate representation of source code into a form that can be readily executed by a machine.

Error Handler:-

→ The task of Error Handler is to detect each error and report it to user.

Symbol Tables:-

→ Symbol Table is a data structure used by a language translator.

#8) Programming Environments:-

⇒ It refers to the collection of Tools for developing software.

Programming Environment Consists of following Features:-

1. platform Independences
2. User Distribution
3. Distributed processing
4. Data Distribution
5. Code Distribution
6. Extensibility
7. Efficiency
8. Security

Platform Independences:-

⇒ Is the code designed for one platform to be used for other platform.

⇒ Programs can run on any software and Hardware.

User Distribution:-

⇒ Is Remote User access Supported by the environment.

⇒ User can distribute Instructions and Data.

Distributed processing:-

⇒ Distributed processing supported by the environment.

Data Distribution:-

⇒ Data located at different Network Locations be used Easily.

Code Distribution:-

⇒ Code held on different sites be Used Transparently.

Extensibility:-

⇒ Easy to Expand / Enhance the Capabilities of an Environment for a Specific Application.

Efficiency:-

⇒ Efficiently code generated by the Environment.

Security:-

⇒ Security provided by an Environment.

Unit - I - (B)

Syntax And Semantics

Introduction :-

* In Early 1960 Around Algol-60 and algol-68 ["ALGOrithm "L"anguage] were initially Developed.

* Syntax and Semantics are the two - important Aspects of studying the programming Languages.

* Though they are different, they have a close Relationship between them.

#(1) General problems of Describing Syntax and Semantics :-

Syntax :-

→ The syntax of a Language is a set of rules.

→ They define low expressions, sentences, statements and program units are formed by the fundamental units called words / Lexemes.

→ The Lexems of a programming Language refers to its identifiers,

literals, operators and special words.

Ex:- a statement in "C".

```
i = 3*c+15;
```

Table Consisting of Lexemes and Tokens

Lexemes	Tokens
i	Identifier
=	Equal sign
3	Integer Literal
*	Multiply operator
C	Identifier
+	plus operator
15	Integer -Literal
;	Semi-colon.

Semantics :-

→ Semantics of a programming Language refers to the meaning designated to various syntatic constructs such as Expressions, statements and program Units.

Ex :- if (<expression>) <statement>

```
if (condition)
{
  stmt 1;
}
```

#② Formal Methods of Describing Syntax :-

→ There are Basically two Types of Grammar Namely

① Context - Free Grammar

② Regular - Expression (Grammar)

→ These two are useful for Describing the syntax of programming Language.

* Context - Free Grammar :-

→ It has Four Components

① a set of Terminal symbols.
② a set of Non-Terminal Symbols.
③ a set of productions.
④ a The start symbol.

Terminals :-

→ Terminals are nothing but End Points.

→ Terminals are basic symbols from which strings are formed.

→ Terminals are also called as Token Names (Tokens)

Ex :- int a = 5;

<token_Name> <attribute_value>

(<id, 5>)

Non - Terminals :-

→ Non-Terminals are nothing but set of terminal values.

→ Non-Terminals are not End values.

→ Non-Terminals are syntactic Variables that denotes set of strings.

Ex:- stmt → if (expr) stmt else stmt.

(ST)ate(M)ent (or) Sentence

(EXPR)essions (strings)

↳ Equations (Conditions)

→ Here "stmt" and "Expr" are Non-Terminals.

Ex:
```
if (condition)
    stmt 1;
else
    stmt 2;
```

```
if (a>b)
printf("a is big");
else
printf("b>a");
```

Productions :-

* Set of Terminals and Non-Terminals are called production.

String = Terminals (+) Non-Terminals

Start Symbols :-

→ All characters in the keyboard are called start symbols.

Ex:-

Capital letters → (A, B, ---, Z)

Small Letters → (a, b, c, ---z)

Operator → (+, -, *, %, /, etc)

Digits → (0, 1, 2, ---, 9)

Special symbols → (;, #, @, *, $, etc.)

Functions → (F_1, F_2, ----, F_{12}).

* Regular Expressions :-

→ Regular Expressions are Most useful for describing the structure of constructs such as Equations.

→ Regular Expressions are nothing but Common equations.

Ex:-

① $a^2 b^2 = a*a*b*b$

② $ax^2 + bx^1 + cx^0 = ax^2 + bx + c = 0$

We convert

→ Regular Expressions into NDFA [Non-Deterministic Finite Automata]

→ Then NDFA – to – DFA [Deterministic Finite Automata]

** In 1959, John Backus described Algol-58 (Algorithmic Language – 1958) with a new Notation.

* Later it was slightly modified by Petter Naur to describe Algol-60.

* This revised method of syntax description became known as (B)ack Us – (N)aur (F)orm (or) BNF.

* EBNF is Extended-BNF.

→ BNF is identical to context free Languages (Grammar)

→ Context-Free Grammar Can simpley as Grammars.

→ A MetaLanguage is a Language that is used to describe another Language.

→ BNF is a Meta-Language for programming Languages.

→ BNF uses abstraction for syntatic structure.

Ex:- <assign> → <var> = expressions

```
int pageno: = 399;
```

#③ Attribute Grammars:-

→ To overcome the difficulty in Specifying in BNF, many other methods were proposed.

→ One such Mechanism was "Attribute Grammars" designed by "Donald-ku-Nooth" in 1968, to describe both Syntax and semantic.

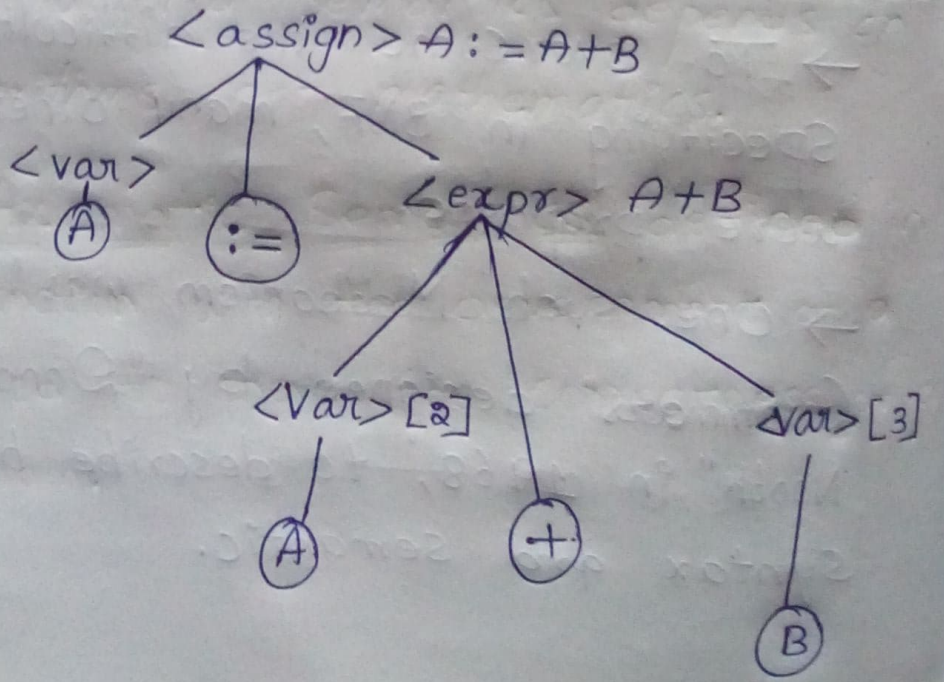
→ For Attribute - Grammars,
① Additionally Attributes,
② Attribute computation functions and
③ predicate Functions are added.

→ Attributes are Associated with Grammar symbols which are similar to Variables.

→ Attribute computation functions (semantic Functions) are associated with Grammar Rules.

→ predicate Functions state some of the syntax and Semantic rules of the Language associated with Grammar rules.

Ex:- a parse - Tree for A:=A+B



Parse - Tree :-

Parser nothing but syntax drawing of a given expression into a tree like structure and checking Errors/Rules of it is called Parse (Syntax) Tree.

Ex:- A:=A+B

#④ Describing the Meaning of the Programs

Dynamic - Semantics :-

→ Dynamic Semantics deals with describing the meaning of the expression statements and program units (small parts).

→ "Dynamic Semantic" can be simply state as "Semantic".

Reasons to know the method of describing Semantics:-

① programmers need to know the meaning (or) the Language statements.

② Compiler writers should know the Semantics of the Language.

③ program correctness proofs are based on formal description of the Language Semantics.

Describing the meaning of Programs:

There is no single widely acceptable notation (or) formalism for describing Semantics.

→ A program is executable software that runs on a computer.

→ programmers create programs by writing code.

→ Computer programming is the process of telling a computer what to do.

Operational Semantics:-

→ Operational semantics deals with describing the meaning of a program by executing its statements on a machine, either real or simulated.

→ operational semantics depends on algorithms, not mathematics.

→ Axiomatic Semantics defined in Conjunction with the development of a method to prove the correctness of a program.

Unit-2 -(A)

Name; Bindings and Scope

Introduction :-

Name :-

Names also known as Identifiers are usually related to the sub-Programs, Labels; Formal parameters and other programming constructs.

⇒ Names be case-sensitive. That is Upper-Case and Lower-case Letters.

⇒ Special words belong to the Reserved words (or) Keywords (or) Library / Inbuilt words.

Ex:- For Case Sensitives

① john
② John
③ JOHN

Three Names are different in Computer Technical programming Language.

⇒ Variable names are the most Common names in a program.

Keywords :-

"A Keyword is actually a programming Language word that has a special meaning".

Ex :- ① auto
② static
③ for
④ if etc--- etc---

⇒ Those Keywords / Reserve words we should not use for Identifiers

---

#③ Variables :-

⇒ Variables are the Abstractions in a Language.

⇒ Variables are Refers to some attributes.

Ex :- a Variable associated to Attributes

① Name ; ② Address ; ③ Value
④ Type ; ⑤ storage value & ⑥ Scope.

Ex :- int i = 5;
float price = 250.50;
↓ Datatype ↓ Variable Name ↓ floating Number (constant)

#④ Concept of Binding :-

⇒ An Association between an Attribute and Entity is called Binding.

⇒ The Time at which binding Takes place is called Binding Time.

⇒ The Values of Attributes must be set before they can be used.

Ex :- int pagenumber;

Binding Occur in Various ways :-

① Language Design Time Binding :-

⇒ When a Language is designed for example "*" operator is bounds to multiplication.

② Language Implementation Time Binding :-

⇒ In most of the Languages a set of Values are bound to an integer type at Language Implementation Time.

Ex :- For Tran; Ada; C and C++

③ Compile-Time Binding :-

⇒ A Variable Type is Bound at Compile Time in For Tran; pascal and C.

④ Linking Time Binding :-

⇒ A Sub-program call is bound to a sub-program code at Link-Time.

⑤ Execution-Time (Run-Time) Binding :-

⇒ In Most of the programming Languages Variables are bound to a value at Execution Time and binding can be modified at Run-Time. [Execution].

⇒ The First Four Categories Refer to binding before Run-Time.

⇒ These categories Refer to static binding.

⇒ Last (5th) Category Refers to Dynamic Binding.

Ex:- float Price;

#⑤ Scope :-

⇒ The Scope (Range) of a Variable is the Range of statement in which the Variable is Visible.

⇒ A Variable is visible in a statement when it can be Refferenced by that statements.

⇒ The scope Rules determined how references to Variables declared outside the currently Executing Sub-program.

There are two-Types of scopes:-

① static scope

② Dynamic scope

① static scope:-

⇒ Static scoping is introduced by Algol-60.

⇒ It implies that the scope of a Variable can be determined statically; before Run-Time.

```
int x, y, sum;
```

② Dynamic scope:-

⇒ Dynamic scoping is based on the calling sequence of sub-programs.

Ex:- Java

⇒ Variable can be declared at any Point of the program.

Ex:- for(int a=0; a<=10; a++)

#6 Scope and Life Time:-

⇒ The scope and Life Time of Variables are Related.

Ex:- a Variable is declared in a JAVA Method, then the scope of the Variable is in the Method only and it ended (Life Time) in the Method only.

Ex:-①
```
import java.io.*;
    class demo
    {
     Public static void main(string args[])
     {
        Void add()
        int a=5;
      {
      System.out.println("a="+a);
      }
     }
}
```

②
```
Void sub()
    {
      int a=5, b=4, Sub;
        Sub a-b;
    }
```

#⑦ Referencing Environments

⇒ The Referencing Environment of a statement is the collection of all Variables that are visible in the statement. (scope/Range).

⇒ The Referencing Environment of a statement in a static-scoped Language is the variables declared in its Local scope.

⇒ Code and Data structures can be created to allow References to Variables from other scopes during Run-Time.

#⑧ Named Constants:-

⇒ A Named Constant is a variable that is bound to a value only once.

⇒ Named constants are useful as Aids (Helps) to Readability and Program Reliability.

⇒ Readability improved by using the name "pi" instead of the constant "3.14159".

$$\boxed{\pi = \frac{22}{7} = pi = 3.14}$$

Unit - ② - Part - ③

DATA TYPES

#① Data Types Introduction:-

⇒ A Data Type defines a collection of Data values and a set of predefined Operations on those values.

⇒ Computer programs produce Results by Manipulating Data.

⇒ A Language support an appropriate Variety of Data types and structures.

Eg:- * "Fortran" - programming Language.
① Linked List
and ② Binary Trees were used.

* "COBOL" - programming Language.
① Integer and
② Floating point data types are used.

* "C, C++ and Java" etc ... onwards
⇒ User defined Data type - enum
[Enumarator]

⇒ Abstract Data types (object etc)

⇒ Arrays; structures; Function & Pointer etc...

#② primitive Data Types:-

⇒ Data Types that are not defined in Terms of other called primitive Data Types.

⇒ Nearly all programming Languages provide a set of primitive Data Types.

Ex:- Integer Values

① Numeric Data Types are
   - ⓐ Integer (Byte; short; int and long)
   - ⓑ Floating-point (float and double)
   - ⓒ Decimal (binary coded decimal [BCD])

② Boolean Types (true [or] false)

#③ character:-

③ Character Types

⇒ character Data are stored in computer as numeric codings.

⇒ 8-bit code - ASCII ("American-standard-code for Information - Inter change).

⇒ ISO (International standard organization) had decided 256- characters ASCII.

#④ character string Types :-

⇒ A character string Type is one in which the values consists of Sequences of characters.

⇒ character string consists are used to label output and input.

⇒ character strings also are an essential type for all programs that do character manipulation.

Eg:- Char c = 'p'; //single character

character string:- Char c[100] = "prasad-sir";

→ string :- copy :- strcpy

Length:- strlen

(Join) Concatination:- strcat

comparing :- strcmp

etc....

#⑤ User-Defined ordinal Types:-

⇒ An ordinal Type is one in which the Range of possible values can be easily Associated with the set of positive Integers.

Ex:- In Java - programming Language
① Integer; ② char & ③ boolean

Some Languages there 2-Types of ordinal Types
① Enumeration Types and
② sub Ranges.

1) Enumeration Types:-

⇒ An Enumeration Type is one in which all of the possible values, which are "Named Constants" are [$Pi = \pi = \frac{22}{7} = 3.14$] provided in the definition.

⇒ Enumeration types provide a way of defining and grouping - collection of Named - constants, which are called Enumeration Constants.

Ex:-

```
enum days {Mon, Tue, wed, Thu, Fri, sat,
           Sun};
```

a) SubRange Types :-

⇒ A subrange Type is a contiguous subsequence of an ordinal Type.

Ex:- 12 --- 14 is a subrange of Integer Type.

⇒ SubRange Types were introduced by Pascal & Ada.

Ex:-

```
Type Days is (Mon, Tue, wed, Thu, Fri,
              Sat, Sun);

Subtype weekdays is Days range
        Mon --- Fri;

Subtype Index is Integer range
        1 --- 100;
```

⑥ Array Types :-

⇒ An Array is a Homogeneous Aggregate of Data Elements in which an individual element is identified by its position in the Aggregate, Relative to first element.

⇒ An Array is a collection of similar data types, which are stored in the sequential memory locations.

Syntax:-

```
int a[5] = { 1, 5, 7, 9, 6 };
```

Generic Syntax:-

Data Type Array Name[Size] → Element.

#⑦ Associative Arrays:-

⇒ An Associative Array is an unordered collection of Data elements that are indexed by an equal number of values called keys.

⇒ In the case of Non-Associative Arrays, the indices never need to be stored.

⇒ In an Associative Arrays, however, the user-defined keys must be stored.

Ex:- In "Perl"-programming Language "Hashes" are Associative Arrays.

#⑧ Records :-

⇒ A Record is a possibly Heterogeneous Aggregate of data elements in which the individual elements are Identified by names.

Ex :- Collection of students (structure).

```
"Struct" student
{
    Student Name;
    Student Rollno;
    Student Marks Average;
}
```

#⑨ Tuple Types :-

⇒ In programming Languages, such as Lisp [LIST processing Language], phython and others. A "Tuple" is an ordered set of values.

Tuple - Data Type Uses :-

① passing a string of parameters from one program to another and

② Representing a set of values Attributes in a Relational Data Base.

⇒ TUPLE is a finite Sequence of terms.

① 1-Tuple is called single (Singleton)

② 2-Tuple is called pair (or) (Couple)

& ③ 3-Tuple is called Triple (Triplet)

Eg:- 6-Tuple is "(5,9,11,3,2,1)"

⇒ Tuple is in mathematics not-Infinite its a finite ordered list (Sequence)

0-tuple reffered as empty tuple.

n-tuple is defined Sequence of n-elements.

#⑩ List Types :-

⇒ Lists are specified in LISP (List processing Language)

⇒ The Elements of simple Lists are Restricted to Atoms (Small element (Item)).

Eg :- (A B C D)

Nested List structures by Parentheses.

Eg :- (A (BC) D (E (F G)))

A List of Four (4) elements

⇒ The first is the Atom "A".

⇒ The second is the subList "(BC)".

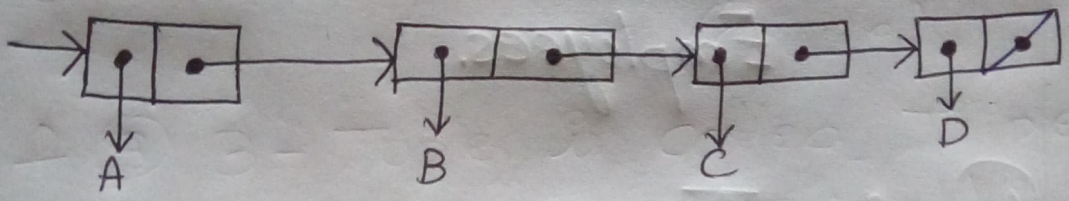
⇒ The Third is the Atom "B".

⇒ The Fourth is the subList "(E (F G))" which has its second Sublist (F G).

⇒ Internally, Lists are usually stored as single-linked List structures.

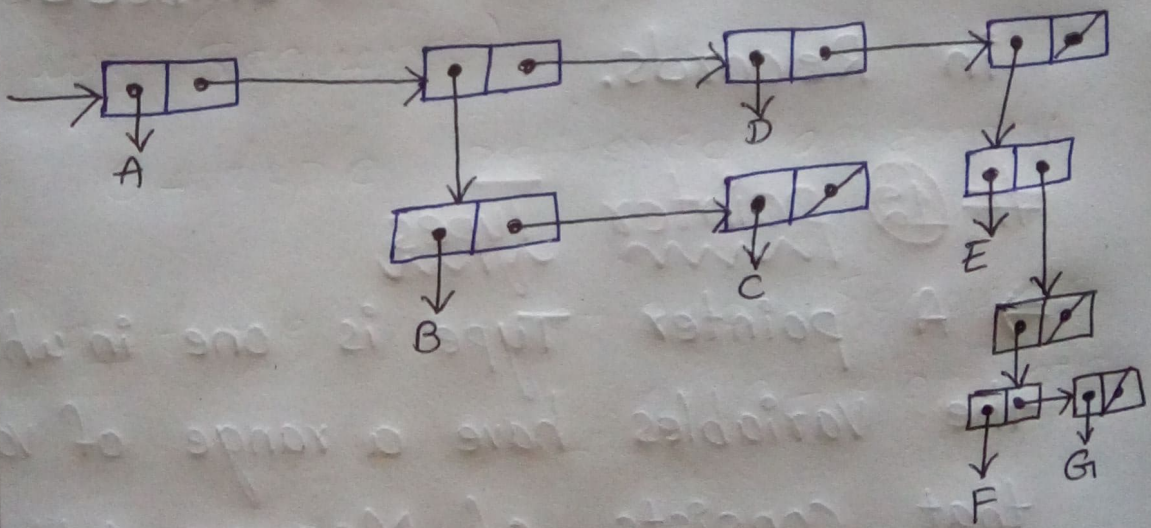
⇒ In which each node has Two-pointer.

Ex:- ① (A B C D)



Note:- The Elements of a LIST are Horizontally shown.

Ex:- ② (A (B C) D(E(F G)))



(11) Union Types:-

⇒ A union is a Type that store different Type values at different Times.

Ex:- Student Group
Books
Employees.

Design Issues:-

⇒ Unions are confined to be Part of Record structures.

⇒ should unions are embedded in Records.

(12) pointer Types

⇒ A pointer Type is one in which the variables have a range of values that consists of Memory Addresses and a special value.

Ex:- int a = 5;
int *a = Address of "a".

Design Issues :-

⇒ what are the scope and life Time of a pointer Variable?

⇒ what is the life Time of a Heap dynamic Variable?

⇒ Should the Language support Pointers Types, Reference Types (or) both?

(13) Reference Types :-

⇒ C++ includes a special kind of pointer Type called a Reference Type.

⇒ Reference Type Used Primarily for formal parameters in function definitions.

Ex :-

```
int result = 0
int & ref_result = result;
- - -
ref_result = 100;
```

⇒ Reference Type Variables are specified in definitions by Ampersands (&).

Type checking:-

⇒ Type checking is the activity of ensuring that the operands of an operator are of compatible Types.

⇒ A Compatible Type is one that is either legal for the operator (or) is allowed under language rules to be implicitly converted by compile - Generated code to a legal Type.

Eg:- int to float

float to int we can change. Type casting (change).

```
int a=5;
float b=6.25;
int sum;
Sum = a+b;
Printf (sum);
```

output = 11.

Type error is changing from one type of Data to other.

Eg:- int to char - Conversion.

#⑮ strong Typing:-

⇒ one of the new ideas in Language design that became prominent in the so-called structured programming Revolution of the 1970's is strong Typing.

⇒ Strong Typing is widely acknowledge as being ahighly valuable concept.

Eg:- 1972 → C-programming Language.

⇒ A "strongly typed" language is one in which each name in a program in the language has a single type associated with it, and that type is know at Compile time.

#16 Type Equivalence :-

⇒ In the "Type Equivalence" objects and contexts can only be identical if they are of the same form.

⇒ where as Type compatibility determines the what information is stored inside the object that are based on the level of compatibility.

Type compatibility :-

⇒ The Idea of Type compatibility was defined when the issue of Type-checking was introduced.

There are Two different Types of compatibility Methods :-

① Name Type compatibility

& ② Structure Type compatibility.

① Name Type Compatibility:-

⇒ It means that two variables have compatible Types only if they are defined in either the same Declaration.

⇒ Name Type Compatibility is easy to implement but is highly restrictive.

② Structure Type Compatibility:-

⇒ It means that two variables have compatible Types if their Type have identical structure.

⇒ There are some Variations of these two approaches and many Languages use combinations of them.

Eg:- in "Ada"- programming Language

```
type Index type is 1 . . . 100;
count : Integer;
index : Index type;
```

Unit ② Part ③

Expressions and Statements.

Introduction :-

Expression Defination :-

"Expressions are the fundamental means of specifying computations in a Programming Language.

⇒ It is crucial for a programmer to understand both the syntax and Semantics of Expressions.

⇒ To understand Expression Evaluation it is necessary to be familar with the orders of operator and operand Evaluations.

#① Arithmetic Expressions :-

In programming languages, Arithmetic Expressions, Consists of "operators" (+, -, *, /, %); Operands; Parentheses; and function cells.

→ The operators can be unary meaning they have a single operand.

→ The operators can be Binary, meaning they have two operands.

→ C- Language include a Ternary operator, which has 3 operators.

→ The purpose of an Arithmetic Expression is to specify an Arithmetic Computation.

It can be Done with Two-Actions:

① "Fetching" the operands, usally from memory and

② "EXECUITING" the Arithmeti operations on those operands.

Precedence Rules Design Issue for Arithmetic Expressions:-

Eg:-

a+b*c ⇒ Result

The a, b, and c are Variables having 3, 4 & 5 Values Respectively.

If Evaluated left to Right

Result = 35

(3+4) * 5 = 7 * 5 = 35.

If Evaluated Right to left
Result = 23.

$$3+(4*5) = 3+20 = 23$$

⇒ BODMAS - Rule means

(B)racket (O)rder (D)ivision (M)ultiplicati
(A)ddition and (S)ubtraction.

#② Over Loaded Operators :-

⇒ Arithmetic operators are often used for more than one purpose.

Eg:-
"+" is used for Integer and floating - point addition.

→ Some Languages "Java" use it for string catenation(Join); this multiple use of an operator is called operator overloaded.

Eg:-
② Ampersand (&) uses as a unary operator AND, and Some time uses as Address

$$x = \&y;$$

#③ Type - Conversions :-

→ Type conversions are either Narrowing (or) widening.

→ eg:-

① Converting (Narrow) a double to a float in 'Java'.

② Converting (widening) Converting int to float in Java.

→ Type Conversions can be either explicit (or) Implicit.

#④ Relational and Boolean Expressions

⇒ In Addition to Arithmetic Expression programming Languages include Relational and Boolean Expressions.

ⓐ Relational Expressions :-

→ A Relational operator is an operator that compares the values of its two operands.

→ A Relational Expression has two operands and one Relational operator.

eg:- (a == b)

Syntax of Relational operators:-

SNO	Operation	Ada	C-language Based	Fortran-95
①	Equal	=	==	==
②	Not Equal	/=	!=	<>
③	Greater than	>	>	>
④	Less than	<	<	<
⑤	Greater than (or) Equal	>=	>=	>=
⑥	Less than (or) Equal	<=	<=	<=

⑥ Boolean Expressions:-

→ Boolean Expressions consists of Boolean Variables; constants; Relational Expressions and Boolean operators.

→ The operators usually include those for the AND, OR and NOT operations and sometimes for

Exclusive OR and Equivalence.

#⑤ short-circuit Evaluation

A short-circuit Evaluation of an Expression is one in which the Result is determined without Evaluating all of the operands and/or operators.

Eg:-

$(13 * a) * (b/13 - 1)$

if $a = 0$ then $13 * a$

$= 13 * 0 = 0$

we need to Evaluate $(b/13 - 1)$ only

```
a=0; // Intiation
a==0; // Assigning.
```

#⑥ Assignment statements

→ The Assignment statements is one of the central constructs in imperative Language.

→ It provides the Mechanism by which the user can dynamically change the bindings of values to variables.

① Simple Assignments :-

The General syntax of the simple Assignment statement is

<target-variable><assignment_operator>
<expression>

C-Language :-

a = = (b+c); (or) a = = b;

ALGOL 60 :-

x := y;

② Conditional Targets :-

C++, Java and C# allow condition Target

Ex:-

```
if(flag)
    count1 = 0;
else
    count2 = 0;
```

③ Compound Assignment operators

⇒ A Compound Assignment operator is a shorth and method of specifying a commonly needed form of Assignment.

Eg:- $\boxed{a = a+b}$

④ Unary Assignment operators:-

The C-Based Languages includes two special unary Arithmetic operators that are Actually abbreviated Assignments.

Eg:- sum = a++;

(or)

a = a+1;

sum = a+1;

#⑦ Mixed-Mode Assignments:-

⇒ Frequently Assignment statements also are mixed mode.

⇒ In a clear departure from C and Java and C# allow mixed-mode assignment only.

→ So, an 'int' value can be assigned to a float variable.

→ In all languages that allows the mixed-mode Assignment.

Eg:-

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

Unit-2 part ④ :-

Control Structures

Introduction :-

→ Computations in Imperative Languages programs are accomplished by evaluating expressions and assigning the Resulting values to Variables.

→ Statements that provide these kinds of capabilities are called control statements.

→ Flow of control (or) Execution Sequence in a program by operator and precedence

Rules :-

* A control structure is a control Statement and the collection of Statements whose execution it Controls.

#② Selection statements :-

* A selection statement provides the means of choosing between two (or) more Execution paths in a program. (statement).

* Those statements are fundamental and essential parts of all Programming Language.

Selection statements are of Two-Types :-

① Two-way selection

② multiple (n-way / Three)-way selection.

① Two-way selection Statements :-

→ The two-way selection statements imperative Languages are quite similar,

Ex:-

```
if control expression
  then clause
else
    clause
      (statement;);
```

Design Issues:-

* The form and Type of the expression that controls.
* The 'then' and 'else' clause specified.
* The meaning of nested selectors be specified.

② Multiple selection statements:-

(n-Type/way)

* The multiple-selection construct allows the selection of one of any number of statements (or) statement Groups.
* If is, therefore, a generalization of a selector.
* The need to choose among more than two control paths in a program is common.

* A multiple - sectors can be built from Two - way selectors and gotos.

Eg:-

```
Switch (expression)
{
  case constant_expression_1: Stmnt 1;
    :
  case constant_expression_n: Stmnt_n;
  [default : statement - n+1]
}
```

Design Issues :-

* The Form and Type of the expression that controls.
* The selectable segments Specified.

#③ Iterative - statements :-

* An Iterative statement is one that causes a statement (or) Collection of statements to be executed Zero, one (or) more items.

* Iteration is the very essence of the power of the computer (programming Language).
* Iteration will reduce program code, complexity save time and memory also.

Design Issues:-

* The Iteration controlling.
* The control mechanism appear in the Loop.

#④ Unconditional Branching:-

* 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.

#⑤ Guarded commands:-

* Guarded commands are alternative control constructs with positive theoretical characteristics.

Eg:- Dijkstra's selection construct

form:-

if < Boolean expression → <statement>

[] < Boolean expression → <statement>

⋮

[] < Boolean expression → <statement>

fi ⇒ The closing Reserved word "fi" is the opening Reserved word spelled Backward.

⇒ Like Gaurded Code (covered).

Unit-3 Part ①
Sub programs

Introduction :-

→ Two Fundamental abstruction included in a programming Language.
① Process Abstraction.
and ② Data Abstraction.

→ In the early history of high Level programming Languages, only Process Abstraction was Recognized and included.

→ In the 1980s, however, many people began to believe that data Abstraction was Equally Important.

→ The methods of object-oriented Languages are closely related to the subprograms.

#① Fundamentals of subprograms :-

→ Definitions :- "A subprogram definition describes the interface to and the actions of the subprogram abstraction.

* A subprogram call is the explicit request that the called subprogram be executed.

#② Design Issues for subprograms

→ Subprograms are complex structures in programming Languages.

→ One obvious-issue is the choice of one (or) more parameter-passing methods

→ The most important Question here is whether local variables are statically (or) dynamically allocated.

→ There is the Question of whether subprogram definitions can be nested.

→ what parameter-passing method (or) methods are used?

→ Are Local Variables statically (or) dynamically allocated?

→ can subprogram definitions appear in other subprogram definition.

→ Can subprograms be overLoaded?

#③ Local Referencing Environments

→ subprograms can define their own variables, thereby defining Local Referencing Environment.

→ Variables that are defined inside subprograms are called Local variables.

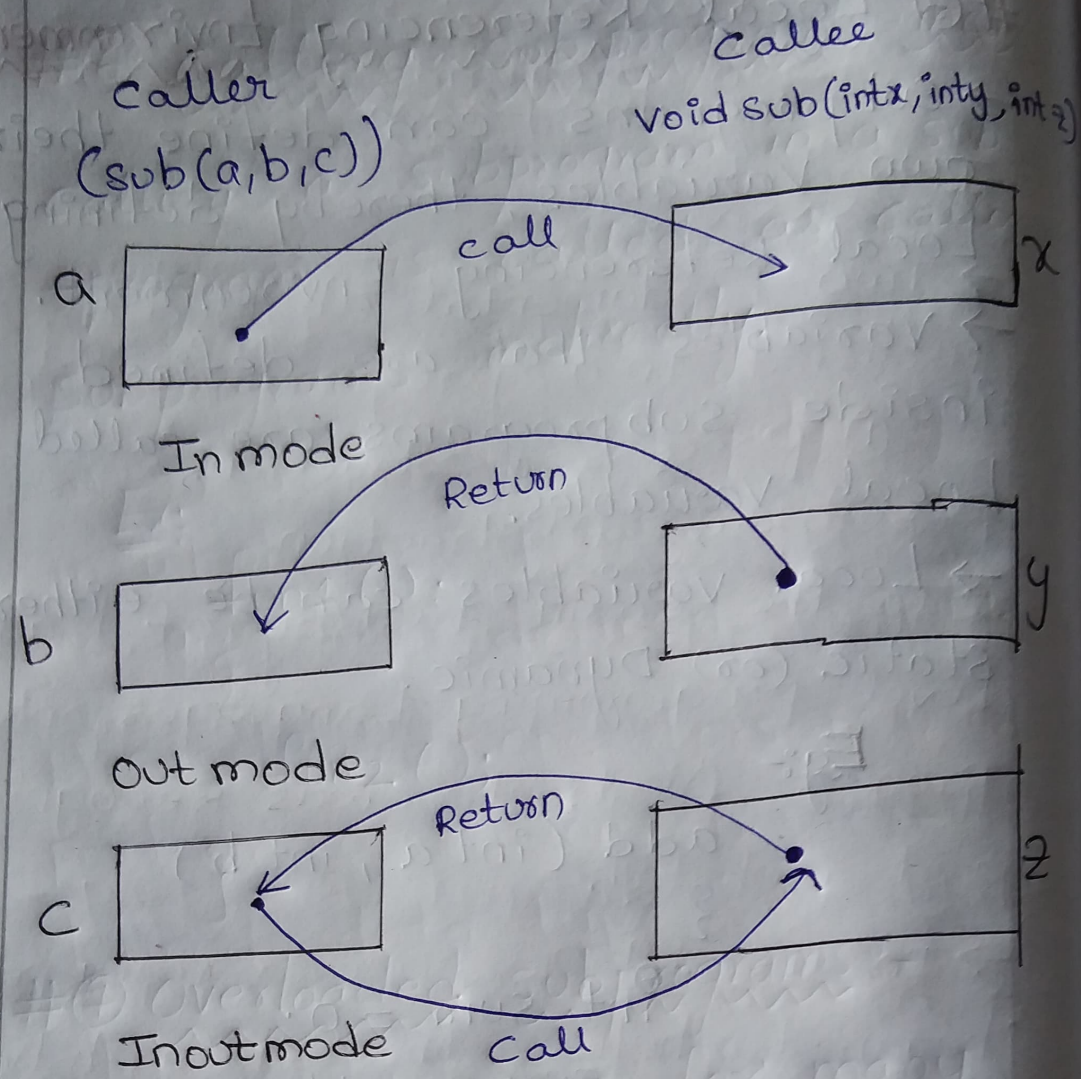
→ Local variables can be either static (or) Dynamic.

Eg:-

```
add (int a , int b)
{
    int sum = 0;
    sum = a+b;
}
```

#④ parameter - passing Methods:-

→ parameter - passing Methods are the ways in which parameters are transmitted to and from called subprograms.



Three - parameter passing Methods.
when physical moves are used.

#⑤ parameters that are subprogram

→ In Languages that allow nested Subprograms, such as JavaScript, there is another issue related to Subprogram names that are passed as parameters.

→ The question is what referencing environment for executing the passed subprogram should be used.

→ The three choices are:

① It is the environment of the call statement that enacts the passed subprogram" Shallow Binding".

② It is the environment of the definition of the passed subprogram "Deep Binding".

③ It is the environment of the call statement that passed the subprogram as an actual parameter "Ad hoc binding"; has never been used.

→ The parameters of subprograms are generic. The parameters are described as formal and actual parameters.

Formal :- Has dummy variable listed in subprogram (called function).

Actual :- It represents value or address listed in the subprogram header and used in the header.

#⑥ Calling subprograms indirectly

Calling a subprogram in a sub-program is called as calling subprograms indirectly.

Ex:-

```
fun1 ();
{
   fun2 ();
}
```

#⑦ Overloaded subprograms:-

Using same name for many subprograms is called overloaded subprograms.

Eg:- In Java

```
Class Hello
{ ≡
Hello()  ⟶ constructor overloaded
{
}
}
```

#⑧ Generic Subprograms:-

These are the template functions in C++

Ex:- Template < class Type >
Type fun (Type a, Type b)
{
return a > b ? a : b;
}

i.e., float fun (float a, float b)
{
return a > b ? a : b;
}

#⑨ Design Issues for functions:-

The functions are designed by

a) No side effects for the function.

b) Return types should be same.

c) functions passes the parameter as objects.

d) functions can pass the addresses.

#⑩ User-defined overloaded operators :-

The operator overloaded is performed in C++

a) It is a type of polymorphism in which an operator is overloaded to give user defined meaning.

b) Overloaded operators is used to perform operation on user-defined data type.

Eg: ++, +, --, etc.

#⑪ Closures :- The scope of the variable holds on to local variables even after the code execution has moved out of that block.

Ex:- Nested Functions.

```
void log (struct s)
{
  int a;
  Void print (struct s)
  {
    a++;
    =
  }
```

closure

#(12) Co-routines :-

It is a kind of sub programs that contains multiple entry points which are under the control of other co-routines.

Ex: RPC (Remote procedure calls).

Unit-3 part②

Implementing Subprograms

Definition:-

The Subprograms call and returns operations of a language is called as Implementing sub-Programs.

General Semantics of calls and Returns:-

* The Subprogram call and return operations of a language are together called its Subprogram Linkage.

* A sub program call has numerous actions associated with it

→ parameter passing methods
→ static local variables
→ Execution status of calling program
→ Transfer of control
→ subprogram nesting.

CALL :-

→ It is a mechanism for parameter passing.
→ Allocates local variables
→ Arranges the transfer of control.
→ It access non-local variables.
→ It executes the unit of subprograms.

RETURN :-

→ It copies automode parameter.
→ Deallocates the storage used for local variables.
→ Disable the access of non-locals.
→ It returns the control.
→ Restore the execution status of the calling program unit.

Implementing Simple sub-programs :-

→ The simple subprograms are implemented by Local variables, Parameter passing and Return Statement of the subprogram.
→ Subprograms cannot be nested.
→ All local variables are static.

Ex :- Subprograms in early versions of Fortran.

Implementing Subprograms with Stack-Dynamic Local Variables

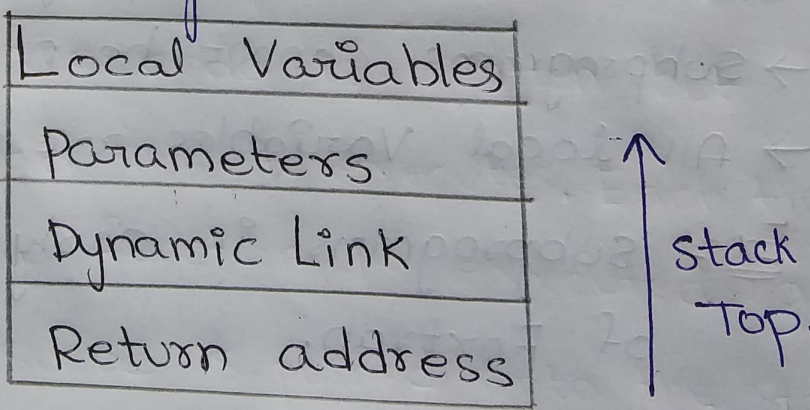
→ The subprograms are implemented by the compiler generating code to allocate and deallocate of Local Variables.

→ Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram).

→ Record format is static. But its size may be dynamic.

→ The dynamic links points to the top of the stack.

→ The return value of the function call should contain space in the memory.



Nested Subprograms :-

A subprogram in a subprogram is called Nested subprogram.

→ some non C-based static-scoped languages (eg., Fortran 95, Ada, JavaScript) use stack-dynamic Local variables and allow subprograms to be nested.

→ The major challenge in implementing nested subprograms is to provide access to non-local variables.

* Two steps are involved:

① Find the instance of the activation record in the stack in which the variable was allocated.

② Use the local offset of the variable (within the activation record instance) to access it.

Blocks :-

→ Blocks are User-specified local scopes for variables.

→ It contains one or more declaration and statements.

→ These are the fundamental to structured programming.

→ The control structures are formed from the blocks.

→ The Blocks can perform

a) Creating a block

b) Nested block

c) It is a section of code

d) It is an algorithm.

Implementing Dynamic Scoping :-

→ Dynamic scoping is a programming Language paradigm that you don't typically see.

→ The scoping that most programmers are used to is called lexical scoping.

→ Dynamic scoping doesn't care how the code is written, but instead how it executes. Each time a new function is executed, a new scope is pushed onto the stack.

→ There are two distinct ways in which Local variables and non-Local references to them can be implemented in a dynamic-scoped Language:

- Deep access.
- shallow access.

Deep access :-

The Dynamic chain links all subprogram activation record instances in the reverse of the order in which they were activated.

→ This method is called deep access, because access may require searches deep into the stack.

Shallow access :-

Put locals in a central place

→ one stack for each variable name.

→ Central table with an entry for each variable name.

Unit-3 part ③

Abstract Data Types

The Concept of Abstraction :-

→ Data abstraction refers to Providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

→ Data abstraction is a programming technique that relies on the separation of interface and implementation.

Introductions to Data Abstraction :-

It is a methodology that enables to isolate, how a compound data object is used.

→ The data abstraction is the process of Taking away or removing characteristics from Something in order to reduce it,

to a set of essential characteri-stics.

Design Issues:-

* The first design issue for abstract data types is the form of the container for the interface to the type.
* The second design issue is whether abstract data types can be parameterized.
* The third design issue is what access controls are provided and how such controls are specified.

Abstract data types:-

→ The representation of objects of the type is hidden from the program units that use the type.

→ The declarations of the type and the protocols of the operations on objects of the type.

Language Examples:-

Modern object-oriented languages, such as C++ and Java support a form of abstract data types.

Parameterized ADT :-

* It is a type or class for objects where behaviour is defined by a set of values and a set of operations.
* parameterized ADTs allow designing an ADT that can store any type elements – only an issue for static typed languages.
* Also known as Generic classes.
* C++, Ada, Java 5.0, and C# 2005 provide support for parameterized ADTs.

Ex :-
Black Box Testing → which hides the inner structure and design of the data type.

Encapsulation Constructs :-

It is one of the fundamentals of oops. It is the binding of data with the methods that operates on that data.

* It is used to hide the values or state of a structured data object inside a class, preventing unauthorized parties to access the data.

Naming Encapsulations :-

* A Naming Encapsulation is used to create a new scope for names.
* Large programs define many global names need a way to divide into logical groupings.
* C++, C#, Java, Ada and Ruby Provide naming Encapsulations.

P.P.L - ④ - Unit :- ① Part :-

Part ①: Object Oriented Programming :-

#① Design Issues for OOPs :-

The OOPs are Design by

① Initialization of objects
eg:- Constructors.

② Nested classes.
③ It is exclusively for objects.
④ Sub classes and Sub Types.
⑤ Polymorphism.
⑥ Type checking.
⑦ Single & multiple Inheritance.
⑧ object allocation & De-Allocation.
⑨ Binding
⑩ Static & Dynamic Binding.

#② OOPs in Small Talk :-

⇒ It is a Part of Object Oriented Programming.
~~It is Explained by.~~
① Every thing is an objects.
② objects have Local Memory.

③ Computations are Performed by objects sending (or) Messages to objects.
④ All objects are allocated with memory
⑤ The De-Allocation is Implicit.
⑥ No Multiple Inheritence.
⑦ The sub class inherits all the instance of the variables.

eg:- C++; Java; Ada-95; OOPs concepts-Net also.

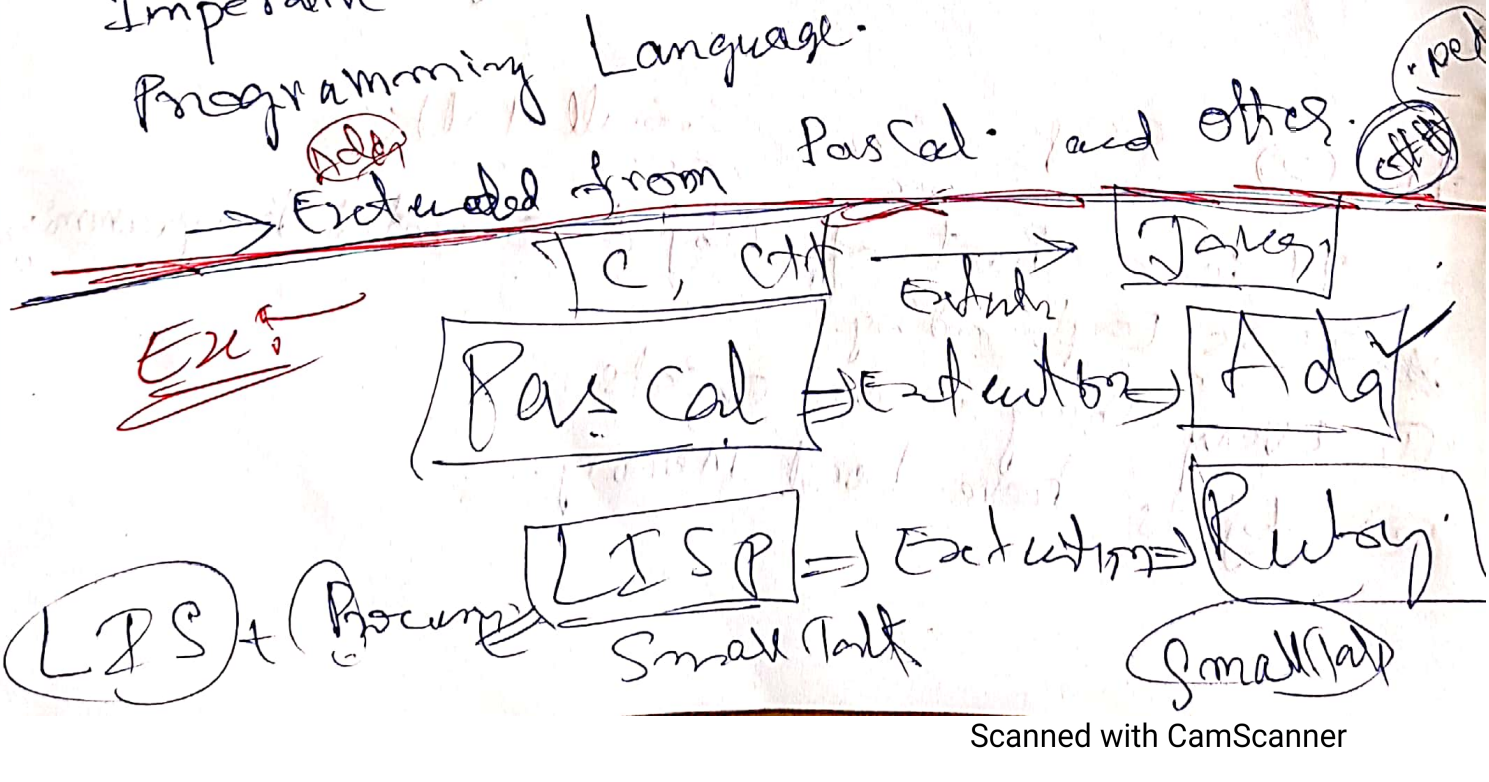
⇒ C++ is semi OOPs concepts Programming Language

⇒ JAVA is Completly OOPs Concepts Programming Language.

⇒ Ada-95:-(1995)

⇒ Ada is a structured; statically Typed; Imperative and object-oriented: high level Programming Language.

→ Extended from Pascal and other.



⑨ Ruby :-

⇒ It is a Open Source and Object oriented Language.

⇒ It sources on simplicity & Productivity.

⇒ The Ruby is created from LISP, Small Talk and Perl.

⇒ The Ruby ~~is created from~~ applies procedural and functional Programming styles.

⇒ It is used in NASA - Research Centers.

⑤ Implementation of object-oriented constructs :-

⟹ Implementation are class; object; Method; Message; Instance variables and Inheritence of the OOPs-Technology in the language like C++; Java; Python etc.

Unit - ④

Concurrency

Introduction :-

* In simple words, Concurrency is the ability to run several programs (or) several parts of a program in parallel.

* Concurrency enable a program to achieve high performance and throughput by utilizing the untapped capabilities of underlying operating system and machine hardware.

* The backbone of Java concurrency are threads.

Introduction to subprogram Level Concurrency :-

* A task or process or thread is a program unit that can be in concurrent execution with other program units.

* The Large programs are divided into subprograms in concurrency level.
i.e., Execution of 2 or more programs simultaneously the execution of program occurs either physically on Seperate processor (or) logically in time sliced fashion on a single processor (or) execution of two or more subprograms simultaneously on a single processor.
* Tasks differ from ordinary subprograms in that:
① A task may be implicitly started.
② when a program unit starts the execution of a task, it is not necessarily suspended.
③ when a task's execution is completed, control may not return to the caller.
* Tasks usually work together.

Semaphores :-

* Dijkstra - 1965.
* A Semaphore is a data structure consisting of a counter and a queue for storing task descriptors.

{* a task descriptor is a data structure that stores all of the relevant information about the execution state of the task.}

* Semaphores can be used to implement gaurds on the code that accesses shared data structures.
* Semaphores have only two operations, wait and release (originally called P and V by Dijkstra).
* The semaphores are widely used to control access to files and shared memory.
* Semaphores can be used to

provide both competition and cooperation synchronization.

Monitors :-

* Ada, Java, C#
* The idea: encapsulate the shared data and its operations to restrict access.
* A Monitor is an abstract data type for shared data.
* It is synchronization construct, that were created to overcome the problems caused by semphores such as timing errors.

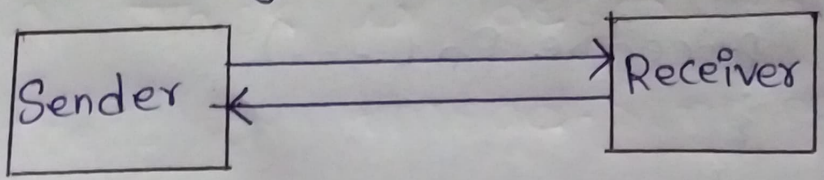
Ex:-

```
monitor monitorname
{
  data variables
  Procedure P1(---)
  {
  }
  Procedure P2(---)
  {
  }
  Procedure Pn(---)
  {
  }
}
```

* only one process can be active in monitor at a time.

Message passing :-

* It is a type of communication between objects, processes (or) other resources used in OOP, inter - process communication and parallel computing.
* Message passing is a general model for concurrency
  - It can model both semaphores and monitors.
  - It is not just for competition Synchronization.
* In the synchronous method, the message passing system requires the sender and receiver to wait for each other while transfering the message.



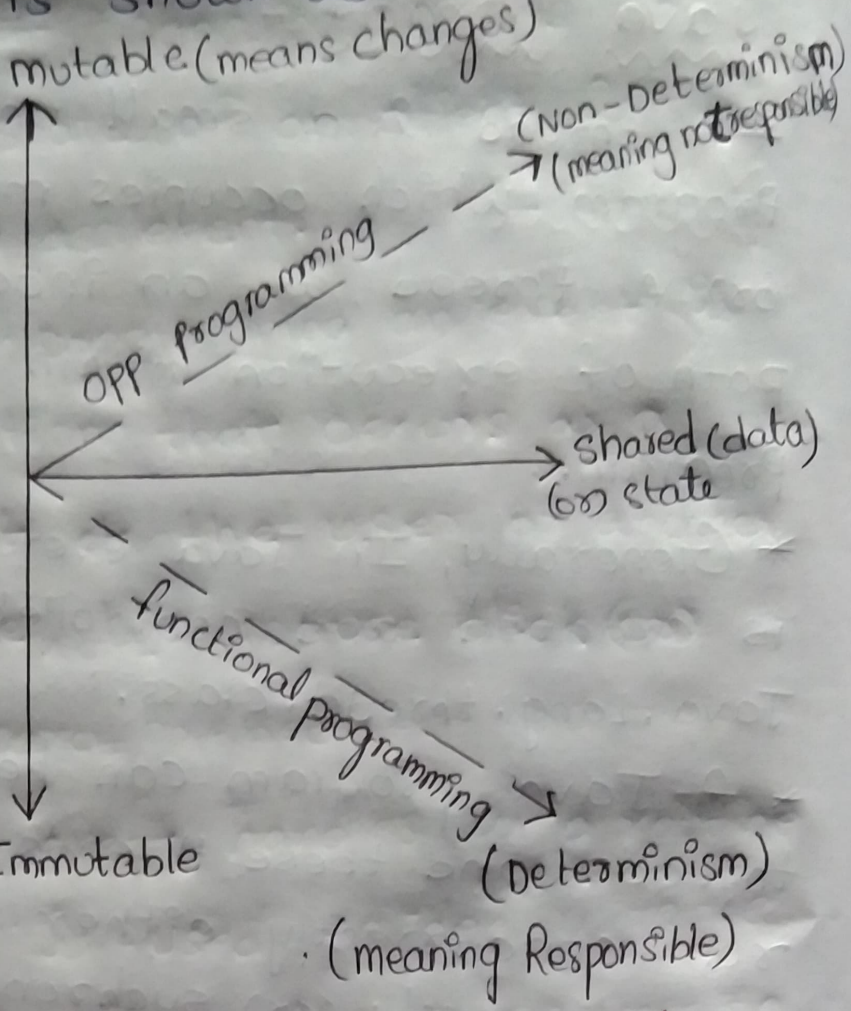
Java threads :-

* A thread is a small unit of program which occupies memory.
(or) A thread is an independent path of execution within a program.
* Generally, In Java every thread is created and controlled by the Java.Lang.thread.class.
* A Java program can have many threads and these threads can run concurrently either asynchronously (or) sychronously.

Concurrency in Function Languages

* The functional languages which are based on mathematical functions.
* The functional programming Languages are LISP, Python, Eslang etc. These Languages supports functional paradigm.
* The functional paradigms supports the concurrent programming Generally the concurrent programming guarentees deterministic execution.

It is shown as



Statement Level Concurrency

* It describes how data should be distributed over multiple memories & statement can be executed concurrently.
* Minimize communication among Processors and the memories of the other processors.
* A Task is described as a Process or thread running on a processor.

* A Thread (or) Task can communicate with other threads by shared variables (or) by through message passing Technique.
* The above description is called as statement level concurrency.

Exception Handling & Event Handling

Introduction :-

* Exception Handling means, a process of responding to exceptions or errors, when a program is running.
* Generally, the exception occurs when an unexpected event happens that requires a special processing.

Ex :- Array overflow,
Array underflow,
$\frac{1}{0}$, etc...

* Event handling means changing the state of an object is known as event.

* In programming an event is an action that occurs as a result of the user or another source, like mouse click (or) Keyboard pressed.

* An event handles is a routine that is used to deal with the event allows the programmes to write code and executed when the event occurs.

Exception Handling in Ada, C++, Java

* The exceptions in Ada is simple compared to C++ and java.

* Generally the exceptions in Ada is an object where as in C++ and java are classes.

* In Ada, if the exception is raised we can't return and resume computation.
* The computations that raised the exception is terminated permanently.
* In c++, the exception is a problem that arises during the execution of a program.
* The c++ exceptions handling in built upon three keywords, try, catch and throw.
* In Java, the exception is similar to the exceptions of c++ but handled by five keywords try, catch, throw, throws and finally.

Event Handling with Java and C#

* In Java and C#, the event handling is the mechanism that controls the event and decide the action to be performed, if an

event occurs.

* This mechanism have the code called as event Handling which will be executed when an event occurs.
* Java uses the class Event to handle the events.
* In C# the event handles as used in GUI (Graphical user Interface) application to handle events such as button clicks and menu selections.
* In C# the event handler is a method that contains the code, gets executed in response to a specific event.

Unit - 5 Part 1

Functional programming Languages.

Introduction :-

⇒ Functional programming Languages are specially designed to handle symbolic computation and list processing applications.

⇒ Functional programming is based on mathematical functions.

⇒ some of the popular functional Programming Languages include: Lisp, Phython, Erlang, Haskell, clojure etc.

Mathematical Functions :-

⇒ The Functions in programming Languages are square roots, Logarithms exponentials and trignometry.

Fundamentals of Functional programming Language :-

⇒ In FPL, variables are not necessary, as is the case in mathematics.

* A purely functional programming Language doesnot use variables (or) assignment statement, thus freeing the programmer from concerns about the memory.

* Repetition must be done by recursion rather than by iteration.

* programs are function definitions and function application specifications, and execution consists of evaluating the function applications.

* A functional Language provides:
  - → a set of primitive functions.
  - → a set of functional forms to construct complex functions.
  - → a function application operation and,
  - → some structure (or) structure for data representation.

*These structs are used to represent the parameters and values computed by functions.

LISP :-

* LISP is an acronym for "LIST Processing Language" - so named because the list is one of the primary data structures in the Language.
* Symbolic AI regards symbolic lists as being a key part of the way intelligent beings and systems actually store and manipulate information.
* LISP is the second oldest higher-Level language.

Support for Functional programming in primarily Imperative Languages

* These are described as
  1) Pure functions
  2) Recurssion
  3) Referential transparancy
  4) Variables
  5) functions
  6) member functions.

Comparision between functional and Imperative languages

Functional	Imperative
1) Declarative programming	1) Algorithmic programming
2) pure functional	2) step by step
3) The execution has Low importance	3) Execution is very important.
4) Function calls	4) Loops, conditions, methods exist.
5) It is procedural	5) It is OOP.
6) Classes are not created	6) Classes exist
7) NO objects	7) objects exist
8) NO Security	8) High Security
9) Readability	9) Tough to Read.

Unit - 5 Part - 2

Logic Programming Language

Introduction :-

It is a type of Programming paradigm largely based on formal logic.

* It is a set of sentences in Logical form expressing facts and rules about the problem domain.

* The Logic rules are written in the form of clauses.

Ex:- $H : B_1, ---, B_n$

i.e.,

H if B_1 and :--and B_n

where H is called Head of the Rule $B_1 -- B_n$ called as Body. Written in the simplified form.

Overview of Logic programming

→ The overview of Logic programming are

* The differences between solving the goals should be done by theorem Proving and model generation.

* The top-down and bottom-up approaches should be solved.
* The relation should be maintained between declarative and procedural representations.
* It should assign Algorithemic Program debugging.
* It should focus on the branch of AI (Artificial Intelligence)

Basic Elements of prolog

* A prolog term is a constant, a variable, or a structure.
* A fact statement is simply a proposition that is assumed to be true.

Note:- Every statement is terminated by a period.

* Rule statements state rules of implication between propositions.
* A goal statement is one that which requests an answer; the

syntactic form of fact statements and goal statements are identical.

Applications of Logic programming

* It is used in parsing techniques.
* It Uses methods and techiques in software development.
* So many software tools developed by Logic programming.
* De-buggers, compilers, Editors, uses the Logic programming.
* It is used in AI and knowledge representations.

Unit-5 Part-③

Scripting Language

Introduction :-

It is a language of commands in a file and capable of being executed without being compiled.

Ex :- perl, PHP and python.

At client side javascript.

Pragmatics :-

* It is a study of a language. i.e how words are used.
* The study of signs and symbols also called as pragmatics.

Ex :- Learning Grammar in English.

key Concepts :-

The key concept explains syntax and semantics of the

The pragmatics explains the meaning of different words and expression which are suitable and approximate for the situation.

Case study: Python - Values and types

* Values() is an inbuilt method in Python that returns a list of all the values available in a given dictionary.

Data types in python

1) Integers
2) Floating point Numbers
3) Complex Numbers
4) strings
5) Built in functions (Math type)
6) Classes, Inheritance, Input/output functions.

Variables:-

It is a symbolic name associated with a value and associated value may change.

* It is also a symbol represents a quantity in mathematical Expression.

Storage and Control

* The assignment of particular areas of a magnetic disk to Particular data (or) instructions are called as storage.

Ex:- assigning an address.

* The control refers to user actions and logic that initiate, interrupt (or) terminate transactions.

Binding and Scope:-

Binding → Declaration have Limited area

Eg:- Variable.

Scope → It depends on the calling sequence of Sub programs.

Procedural Abstraction:-

It is an idea that each method should have clear (or) coherent Conceptual descriptions, that seperates implementation from the users.

* you can encapsulate behaviour in methods that are internal to an object or methods that are widely usable.
* Methods should not be too complex or too long.

Data Abstraction:-

Hiding Unnecessary data and viewing the relevant data inorder to reduce complexity and increase effeciency.

Separate Compilation:-

*It allows the program to be compiled and tested.
One class at a time, even built into Libraries for later use.
* In this, each class is placed in separate file and taken for compilation.

Module Library :-

* A Module is a collection of variables and objects.
* Every module has a diagram view, displaying the objects in the form of nodes, arrows indicating dependencies.
* A module may contain hierarchy of modules.