Theory and design

OF PROGRAMMING LANGUAGE

- Introduction:

Basically, how languages have evolved in the recent times: (Terms discussed)

- Parallelism
- Agility
- Increased speed, decreased delay
- Modularity
- Interdependent/ independent modules.
- Better throughput
- Decentralized setup
- The 80-20 rule => Research and work models.
- waterfall model + sequential approach vs. Agile framework → sprint/non-sequential
- Integration ↳ Modularity; CBT → component Based Tech

- Author: Ravi Sethi

Unit I: Imperative Programming Paradigm:

The Imperative Programming Paradigm: Basics of Programming Paradigms, Expression Semantics, Data Semantics, Imperative Design principles – Structured, Turing Complete, Modularity, Procedural abstraction, Structured Theorem - Sequencing, Selection, Iteration and Recursion.

- Need of the study:
  - Understand obscure features
  - Choose the best one among alternatives
    Eg. threads/ executive in Java
  - Good use of debuggers, linkers, tools, etc.
  - Simulate useful features in lang. that lacks them
    ↳ compatability
  - Better use of technology

- Programming:
  - It is a medium with which you communicate with the computer.
  - Medium must be readable and flexible
  - Natural: Humans write it :)
    Formal: computer system for communication
  - Essentially, it is algorithm + data
    ↳ Element/ sentence/ text with which algorithm

and data of given task is rep.

- Basic building block
- Specification of computation and notation of writing programs.
- Theory is equally powerful for product base.
- Practical is not equally powerful in case of delivery

- Basics of programming paradigms:

- way to organise programming languages based on their features
- Ways/ styles using which programming languages can be organised.
- It is more of an organisation tool that houses many languages.
- Naturally, It is neither a language/ tool in itself.
- there are many PL, each with its own strategy or methodology → which is known as "paradigm".

- Common programming paradigms:

- Imperative and declarative → the most primitive types

(i) **Imperative**:

- Oldest one
- Lang. like C, Fortran and basic
- Eg. used in von-Neumann level architecture
- Sequential: procedural and OOP
- Parallel: concurrent
  ↳ Akka
- Scripting lang.: subset of von Neumann lang.
  Eg. csh, cmd, etc.

(a) **Sequential programming**.

(1) Procedural:

- Based on a step-by-step process → series of computational steps.

Eg. C
- Code reusability is supported

(2.) Object Oriented Programming:
- Data is in the form of fields and code is in the form of procedures.
- Eg. C++, Java, Pascal

(b.) Concurrent Programming:
- Parallel processing system is employed
- Overall computation is factored into sub-computations → executed concurrently instead of sequentially
- Eg. C, Java, Python.

Note:
- Concurrency vs. parallelism:
  Both give extensive ans. of throughput.
  Concurrency → broken into pieces, then executed
  Parallelism → executed as a whole (x multithread)

(ii.) Declarative:
- Express logic without computation's control flow;
- focuses on "what" rather than "how".
- Eg. Math, data, known facts, prolog, XSLT
- Logical, Functional and Database → types

(a) Logic Programming: (Depend on DB)
- With its knowledge base, the program provides an answer to the given task/qn.
- Eg. AI features. → Ada, prolog languages, XSLT
- Based on a set of "logic rules"
- Models comp. as an attempt to find values

(b) Functional Programming:
- Abstraction is the central model
- Data are loosely coupled with functions
- fun^s become data @ the time of LOC
- perl, javascript → static to dynamic webpage
- Hyper calls too:)
- communication bet user and kernels → sys. calls
- computational model based on recursive defn. of functions.
- Take inspiration from lambda calculus
- Eg. LISP, ML → model / framework

Cohesion refers to the degree to which the elements within a module or component of a software system are related and work together to achieve a common purpose. On the other hand, coupling refers to the degree of interdependence between different modules or components in a software system.

(c) Database Programming: (↔ Declarative)

- Based on data and its movement
- Eg. SQL
- Data flow languages: form model comp. as the flow of info. among primitive functional nodes.
- Eg: Id, Val

  ↓

  Row and column attributes (field attr.)
- XML → DB that is used in web designing (Makes it portable)

Compilers and Interpreters:

(i) Lexical Analyser → converts words to tokens

(ii) Syntax Analyser

(iii) Semantic Analyser

(iv) Intermediate code generator
→ temporary code with redundant lines

(v) Optimizer → optimizes the code and reduces unwanted lines.

(vi) Code generation → executable code

compiler → converts to .exe file while interpreter does not. (Compiler is better this way)

compilation vs. interpretation → not opposites, but no clear cut distinction either.

compiler:

* Source prog. → compiler → target prog.

Input → target prog → output

* Better performance

Interpreter:

* Source
  Input → Interpreter → Output
  - Not a completely exe. prog.
  - process must be re-done in a diff. machine

* Greater flexibility ; better diagnostics.

Translator:

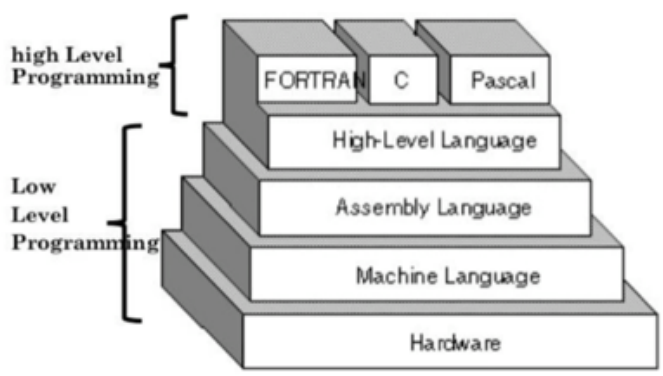
* Source → Translator → Intermediate

  Intermediate
  Input → virtual machine → output

* Note: virtual machine is an emulation of our original machine

* System software : 3 types → int., comp., assembler.
  ↓
  Embedded in kernel space

*



Computer languages:

- Can be high level or low level

- (i) High-level :

  - Unambiguous → common gd. betn client and dev.
  - Readable
  - Easy to understand

- User-friendly
- Mostly imperative
- Portable and machine independent
- Eg. C/C++, Java, python, Fortran
- maintenance and modification is easier.

- (ii) Low-level:

- Can be Assembly (middle) / Machine code (low)

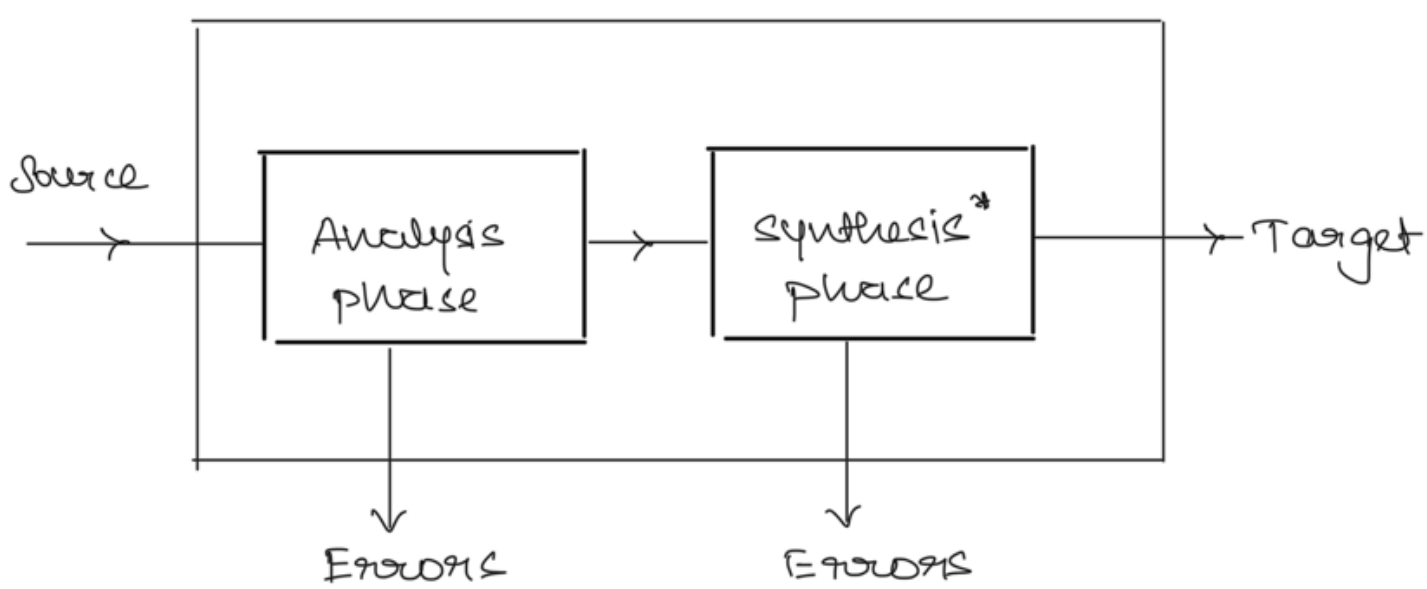
- Can prove to be a bit difficult for the user. Mostly declarative.
- Assembly lang: has certain mnemonics.

Machine code lang: In terms of 0s and 1s (CPU dependent)
↓
Direct relation with CPU.

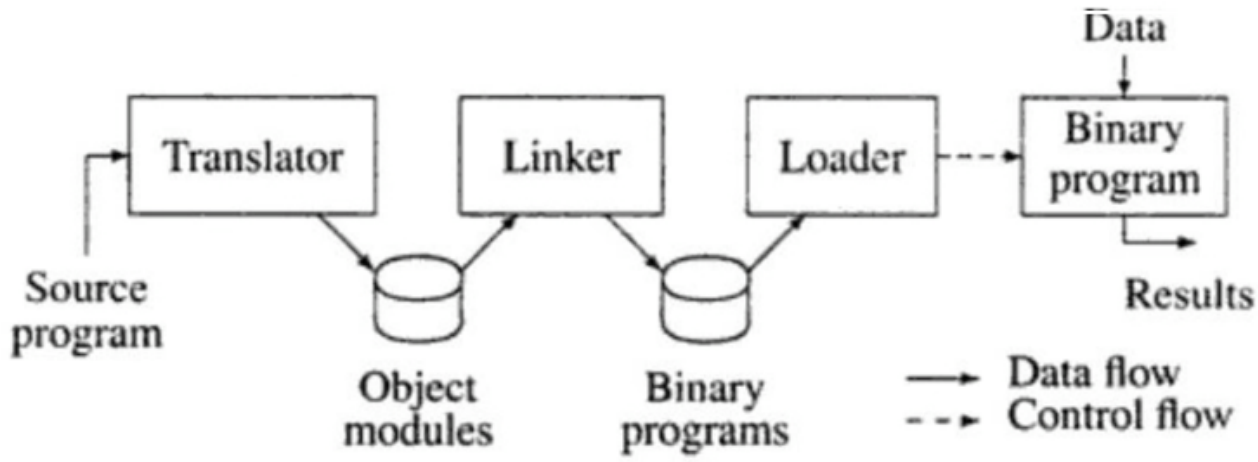
- Translators: Assemblers, compilers and interpreters

	Assembler	Compiler	Interpreter
Description	• Translates assembly language into machine code. • Takes basic commands and operations from assembly code and converts them into binary code that can be recognised by a specific type of processor. • The translation process is typically a one-to-one process from assembly code to machine code.	• Translates source code from high-level languages into object code and then machine code to be processed by the CPU. • The whole program is translated into machine code before it is run.	• Translates source code from high-level languages into machine code, ready to be processed by the CPU. • The program is translated line by line as the program is running.
Advantages	• Programs written in machine language can be replaced with mnemonics, which are easier to remember. • Memory-efficient. • Speed of execution is faster. • Hardware-oriented. • Requires fewer instructions to accomplish the same result.	• No need for translation at run-time. • Speed of execution is faster. • Code is usually optimised. • Original source code is kept secret.	• Easy to write source code, as the program will always run, stopping when it finds a syntax error. • Code does not need to be recompiled when code is changed. • It is easy to try out commands when the program has paused after finding an error – this makes interpreted languages very easy for beginner programmers to learn to write code.
Disadvantages	• Long programs written in such languages cannot be executed on small computers. • It takes lot of time to code or write the program, as it is more complex in nature. • Difficult to remember the syntax. • Lack of portability between computers of different makes.	• Source code is easier to write in a high-level language, but the program will not run with syntax errors, which can make it more difficult to write the code. • Code needs to be recompiled when the code is changed. • Designed for a specific type of processor.	• Translation software is required at run-time. • Speed of execution is slower. • Code is not optimised. • Source code is required.

Language Process:



* synthesis phase: creation of data structures (Memory allocation) + code generation.



Every language requires the two:

(i) Syntax (ii) Semantics } Grammar + Logic

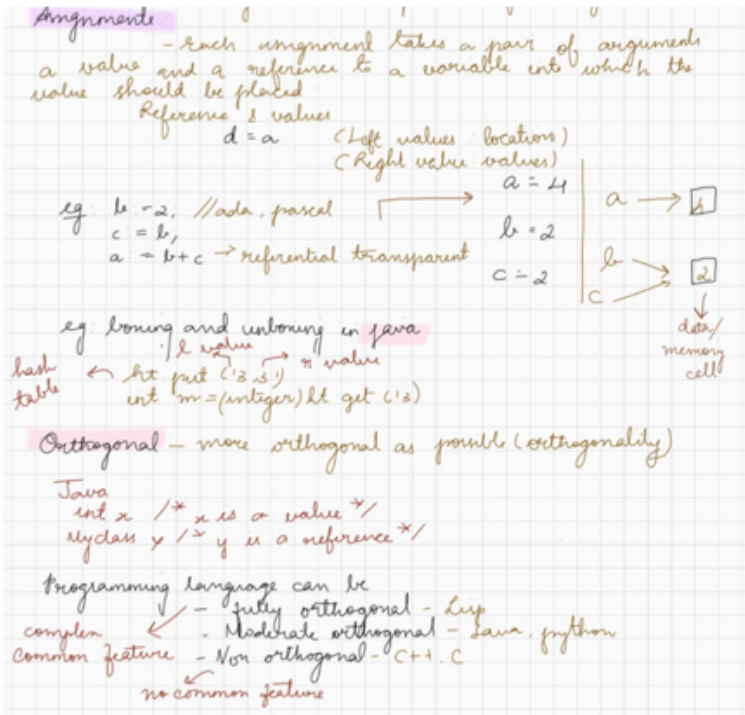
Evaluate, reveal meanings

↓

Based on diff. languages.

L: Memory loc.

R: Data cell value



Orthogonal:

↓

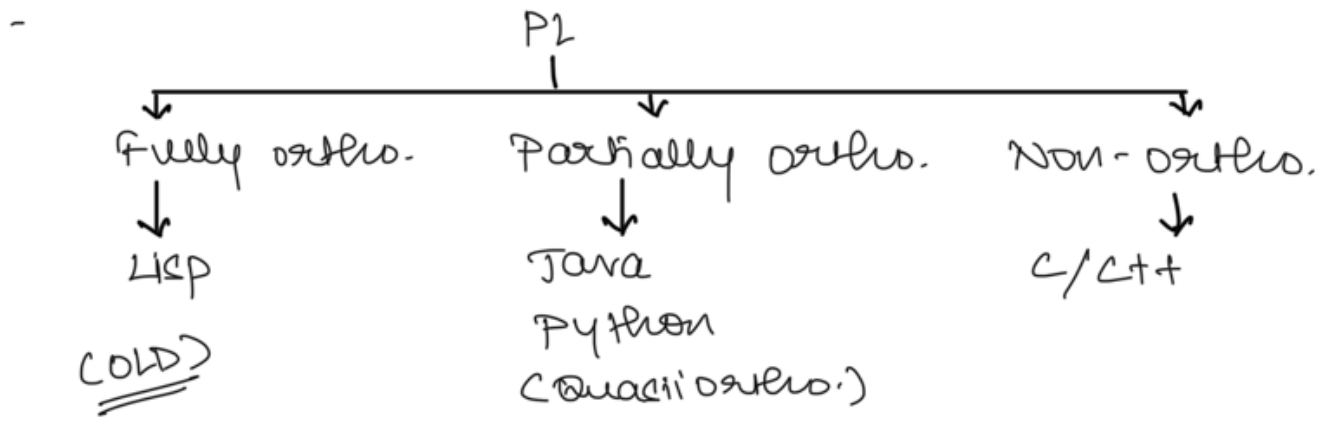
Orthogonality - more orthogonality as possible

↳ we must analyse if it is useful / good

Eg. Java:

int x; → value (x)

Myclass y; → ref (y)



Language	Imperative	Declarative	Logical	Functional	Procedural	Object-Oriented
C	90%	5%	0%	0%	100%	0%
C++	85%	5%	0%	10%	90%	70%
Pascal	80%	10%	0%	0%	90%	0%
Java	70%	10%	0%	10%	60%	80%
Fortran	90%	0%	0%	0%	100%	0%
Python	60%	20%	0%	20%	60%	40%
Ada	80%	5%	0%	0%	90%	10%

Expression semantics → using Java :

* Note:
  Ut. put (13, 21)
  ↓
  Hashtable : Method in Java
  Get L value from 13th loc and store in 21st loc.

  the above is known as boxing method in Java

  Semantics represents the logics/ correlation

Whenever you pick a PL, the features inside the LOCs must be independent
↓
Identify the +ves and -ves → features of LOC must not be dependent / non-correlated.

Orthogonality : LOCs are independent of each other
↓
Orthogonality is +ve, but overall quality of code -ve.

Eg. if, switch, case, conditional structs, loops.

Here, Orthogonal > Quasii > Non-orthogonal

↓

Can be manipulated to favour either side

* Fully orthogonal: more expressive and flexible & programmers have greater freedom

* Non-orthogonal: can be more complex to learn; more rules and exp. to remember.

* Quasii orthogonal: strike a balance bet" orthogonal and practicality.

↓

Good compromise between expressiveness and complexity.

Lisp → fully orthogonal

Eg.

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))

(print (factorial 5)); //output is 5! → 120
```

Quasi → python

Primitive types, mutable and immutable objects, dynamic typing, built-in functions and methods

map function, filter predicate

Non-orthogonal → C

Reasons: memory management, Data redundancy, inefficient storage → both extremes

* combination Assignment Operators:

- Updating assignment: a=a+1 → clutter free

- compound assignment : x += 1 // x = x+1

- pre post increment

* multiway assignment:

a,b := c,d; & a:=c; b:=d; // clu

used in NL - Pattern matching mechanisms.

* Data semantics
↓
Meaning and use of data in computer

Types: int, float, Boolean, Complex, decimal, char, str. -> arr/pri

- static: COBOL, java

- Lim dyn. length: C/C++

- Dyn. Perl, JS, SNOBOL4

- Ada: all three :)

* Implementation:

- Static length: compile-time descriptor

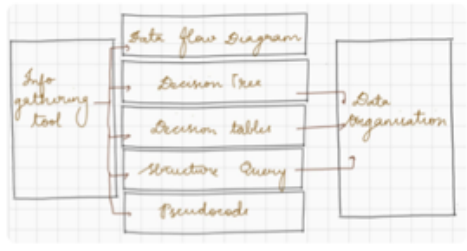
- Limited dynamic length: may need run-time desc.

- Dynamic length: run-time desc.

* User-defined ordinal types:

- Eg. enum days {mon, tue, wed, thu, fri, sat}

* Structured program:



- Graphic presentation
- Logic rather than physical
- Clear picture of data/system flow
- High level overviews to low level c

Turing completeness:

Alan Turing: a limited set of states, an infinite amount of storage and transition function.

Turing-computable machine

working principle ∴
↓
with the state transition diagram

Turning completeness → describes a prog. system
⇒ stores any comp. problem.

Eg. C/C++, Java, python, Go, Visual Basic.

*Modularity:

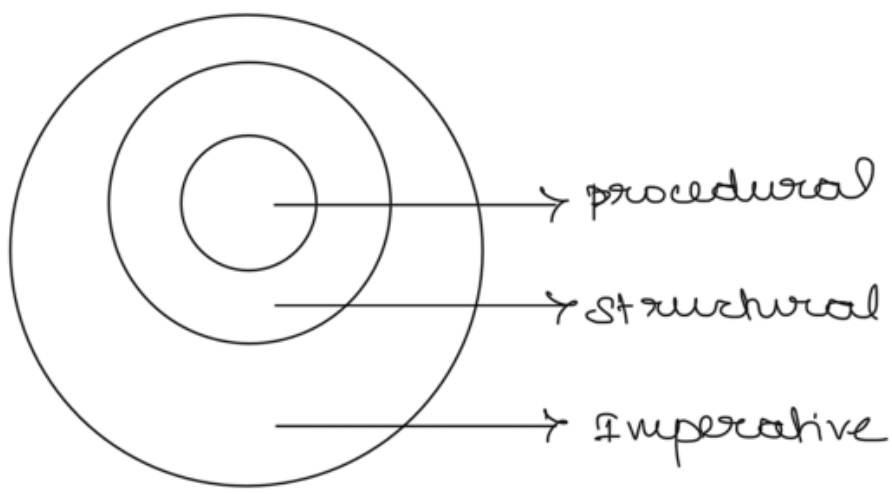
- Advantages of using functions:
  - ↳ Separate pieces of LOC
  - ↳ code reuse / maintenance
  - ↳ Modification / maintenance
  - ↳ more understandable prog.
  - ↳ plug-in plug-out

Rapidness,
Agile
shorter deadline

* Procedural Abstraction:

x ← square the Num (3)
// x now holds value 9

Note:



* Algorithms:

↳ Divide and conquer
↳ Brute Force

Recursion → Activation stack that can be compartmentalized.

Turing machine → mathematical abstract model for computation.

Used in proving correctness of computation in a program. Turing completeness.

---

* Procedural Abstraction:

Ravi Sethi - chapter 5 (upto 181)

- Examples in C can also be written.

- "Procedure": piece of code / program.

  - Functional (return a value) — Fruitful fn.
  - Proper (do not return any value)

> No return values in procedure + compartmentalised & modularised

- name — occurence — Declaration — Location values (r-value to r-value)
  - scope (Acc. and vis.)
  - Activation
  - States

- Example - function definition:

```
void swap (int a, int b)
{
  int z;
  z = a;
  a = b;
  b = z;
}
```

Executed when fn. is called or is 'activated'.

- Elements of procedure:
  - name
  - Body
  - Parameters → formal (defn.) / actual (passed)
  - return (optional).

- Programming process: Understand → outline → decompose gen. soln. → develop specific case → crcd. → Implement

- SRS delivery → Functional/Non-functional → What → How

- Parameter passing value: call by value and call by reference.

Let us see an example in Pascal:

```
Function square <x.Integer> : <Integer>
                 input data      return data
var sq. integer
          passing
begin

      sq = x * x

end
```

- Parameter passing methods:

↳ call by value (data) → Impermanent changes.

↳ call by reference (address) → permanent changes.

↳ call by name

Lazy execution

Example:

```
#include <stdio.h>

int i;

int main()
{
   i=4;
   int a[10] = {0,1,... 9}

   fun. (i, a[i])

}
```

```
                    i     a[i]
void fun (int a, int b)
{
    a = a+b     | A+a[4] = 4+4 = 8
    a = a-b     | 8 - a[8] = 0
    b = a+a     | 0+0 = 0
    b = a-b     | 0-0 = 0
}
```

Recursion Multiple Activation (fun. call)

function factorial <n: Integer>; <Integer>

begin

if (n == 0 || n == 1)

then fact = 1

else

fact = n * fact(n-1)

end

- Features of procedural programming and their benefits :-

- Limitations:

  ↳ No data security
  ↳ Focused on operations
  ↳ No portability
  ↳ cannot model the real world
  ↳ Unsuitable for complexity

- Use cases:

  • Any setup that is machine dependent
    ↓
    creation of compilers, OS, kernels, etc.

  • Also in those cases where state is left unaltered

- No real world execution → as it does not support change of states.

- Optimization process: recursive program
  ↓
  compiler knows the syntactic queue → reduced time complexity.

Note: call by value: formal is converted to actual
↳ Both actual values and dummy parameters (the opposite in call by ref.)

Qns: C program for call by value and call by ref → use global and local variables.

Here, we shall implement call by value and call by reference in a C program.

The C code:

```
#include <stdio.h>
int a=1, b=2; // global variable.

void swap (int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
    print ("%d %d", a, b);
}

void sum ( int *n , int * m)
{
     *m = *m + *n ;
      print ("%d", *m); // call by reference
}

// main program.
```

Justify procedural activation in the code
(can be same/ diff. code)

Note: Dangling pointers

Conclusion: (Parts A and B)

- Represent whether program is holding circular dependency

- Is the program recursive?

- Count the no. of times fn. is activated (numeric info.)

- Does the program have req. pointer?

- can the program be modular?

- Does the program support top-down development?

- Procedural / Functional / OOPs → Simi / diff.

- Why is data mutable / immutable.

- Define dangling / dancing pointers.

Date: 30/9/24

write a C program which comprises of call by value and call by reference (C global and local variable)

```
#include <stdio.h>   → directives
int globalVar = 100
void callbyvalue (int a){
    printf("Value before modfication /d\n", a);
    a = 20
};  printf("Value after modfication /d\n", a);
void callbyreferencecint *b) {
    printf("Value before modfication /d\n", *b);
    *b=30;
};  printf("Value after modfication /d\n", *b);
void demonstratevar () {
    int localvar = 10;
    printf("local variable /d", localvar)
    printf("global variable /d", globalvar);
};
int main(){
    int x = 10;
    int y = 40;
    callbyvalue(x);
    callbyreference(&y);
    demonstratevar();
    return 0;
};
```

Answers:

- circular dependency occurs when two or more modules depend upon each other, creating a loop.

- A recursive program is a program that calls itself
(Time complexity is significantly).

- Referential pointer is used to store address of another variable

```
int a = 5;
int * ptr = &a;
```

- A modular program is a program that can be broken down into simple, independent modules.
(can be stored in separate files as well).

- Top down approach: program that is designed starting from highest level of abstraction and gradually breaking into smaller, specific components.

- Mutable: data can be changed after initialization.
Eg. array

Immutable - data cannot be changed after initialization.
Eg. Tuples.

- Functional: call by value; returns the value
↳ does not change state of system.

Procedural: call by reference; does not return any value.
↳ may/may not modify state of the system.

- Scope rules for names: lexical and dynamic

Macro expansion dynamic scope:
actual parameters are textually sub. for formals.
+
resulting procedure body is textually substituted for the call.

Name → Declaration → Location → value } procedural.

Lexical comes under static

- does not even have renaming options.
- remains static up till the end.

Eg. # define MAXBUFF 4

- Lisp → Mc Carthy → dynamic scope → bug.

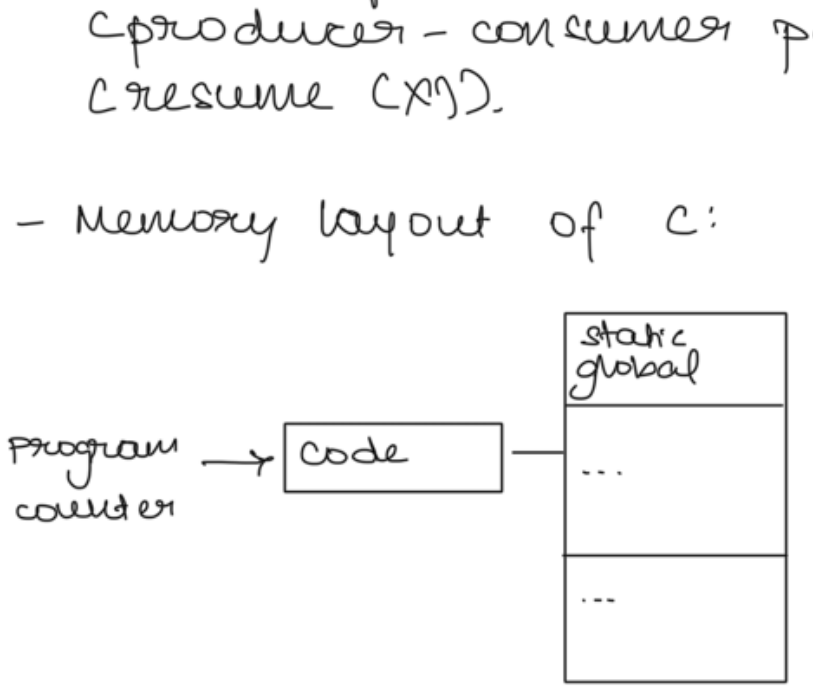
- Renaming a lexical scope has no effect.

- Call by name resolves naming conflicts by renaming locals.

- variable storage → global / static / init.

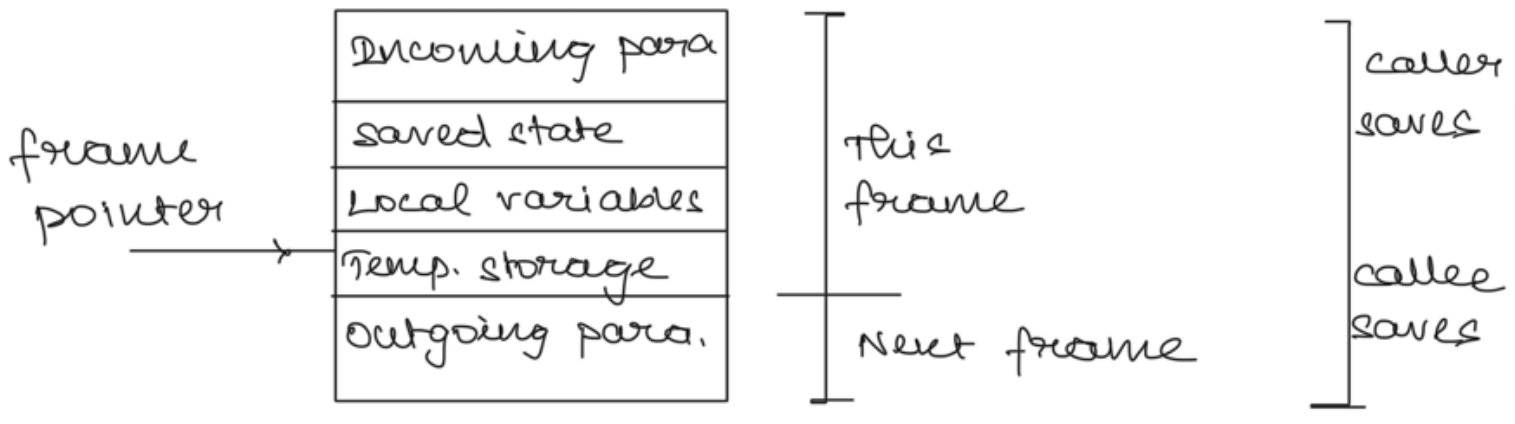
- Last-in first-out manner → coroutines (producer-consumer pairs) and simula 67 (resume (X)).

- Memory layout of C:



frame pointer moves inside the entire network of the stacks.

Each activation: Ready, run, wait, pause, resume
↓
5 states in any activation.



This frame: current execution;
next frame: outgoing of current, incoming of next.

Callee: no overwritten information is allowed.

- Dangling pointers: hang freely.

  ↳ To avoid we use malloc(), realloc() or calloc().

- Benefits: data semantics + compiler implementation.

- calls are dep. on stack memory

  ↓ Avoid fragmentation

  Heap memory.

Background motivation:

A process / thread is a potentially active execution context.

Classic von Neumann: model of computing.

Process can be abstraction of a physical process. Here, one process writes multiple threads.

Can run in parallel set ups, interleaved and run-until-block.

Low coupling and high cohesion

- Concurrency with interleaving process.

- What is currency, features of concurrent prog. paradigms, modularity and race cond^n

- Parallel prog. + deadlock.

- concurrency pictorication

Layer concurrency: It marks

OOP design principles – SOLID principles.

Advantages : maintainability, testability and flexibility.

S- Single

O- Open / Closed principle
↓ ↓
Extension Modification

Liskov : Uncle Bob - Li Principle

I: Interface Segregation Principle

D: Dependency Inversion Principle
↳ Loosely coupled, flexibility and testability.

Questions on determining the program flow.

Defns. of OOPs concepts - Qn.

1. Encapsulation:
Encapsulation is achieved by keeping class variables private and providing public getter and setter methods.

```
class Person {
    private String name; // Encapsulated field
    private int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter and Setter methods
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
```

```
}
```

2. Inheritance:
Inheritance allows a class to inherit fields and methods from another class.

```
class Employee extends Person {
    private String jobTitle;

    public Employee(String name, int age, String jobTitle) {
        super(name, age); // Call to parent class constructor
        this.jobTitle = jobTitle;
    }

    public String getJobTitle() {
        return jobTitle;
    }

    public void setJobTitle(String jobTitle) {
        this.jobTitle = jobTitle;
    }

    public void displayInfo() {
        System.out.println(getName() + " is a " + jobTitle + " and is " + getAge() + " years old.");
    }
}
```

3. Polymorphism:
Polymorphism allows one method to perform different functions based on the object that invokes it.

```
class Manager extends Employee {
    public Manager(String name, int age, String jobTitle) {
        super(name, age, jobTitle);
    }

    @Override
    public void displayInfo() {
        System.out.println(getName() + " is a Manager in " + getJobTitle() + ".");
    }
}

public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee("Alice", 30, "Engineer");
        Employee mgr = new Manager("Bob", 40, "IT");

        emp.displayInfo(); // Outputs: Alice is an Engineer and is 30 years old.
        mgr.displayInfo(); // Outputs: Bob is a Manager in IT.
    }
}
```

4. Abstraction:
Abstraction hides complex implementation details and exposes only the necessary parts.

```
abstract class Vehicle {
    abstract void start(); // Abstract method

    public void stop() {
        System.out.println("Vehicle stopped.");
    }
```

```
}

class Car extends Vehicle {
    @Override
    public void start() {
        System.out.println("Car started.");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.start(); // Car started.
        car.stop();  // Vehicle stopped.
    }
}
```

Part-C:
SOLID Qns, OOPs principles.
4th unit → concurrency + interleaving

Michael: Exp. and data semantics. } PPT Too
Others: Ravi Sethi .ie. } Unit two → PPT will do

Unit-3 ⇒ Java concepts → rej, OOPs notes.

- What are procedures? } part A
- Modularity ?
- Define OOP.

SOLID principles .ie.

concurrency, parallel, functional, OOP, multithread, multilevel job execution
↓
synchronisation issues } concurrency technique + syn. problem
⇒ Deadlock, race conditions.
↓
Techniques in Ravi Sethi book + PPT

Swapping techniques in Unit 4
↳ Interleaving mechanism.

structure of imperative design principles.
Turing complete machine - Tape machine
(Bloom's Taxonomy)

Mindmap structure ⇒ comparison study → paradigms

Unit-IV: Concurrent Programming

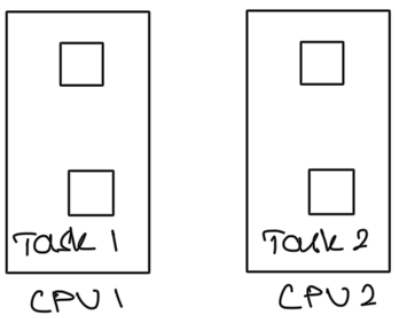
Defn. It is a technique in which two/more processes start, run in an interleaved fashion.
→ through context switching.

Focusses upon communication (IPC) and synchronization (control exchange).

Parallelism:

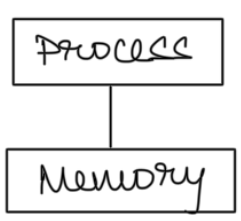
- Achieved true hardware parallelism → 2 processes on 2 separate CPU cores.

Machine Organization:

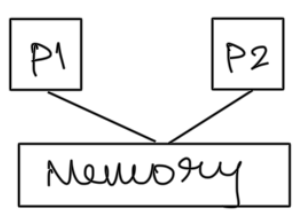


3 types:
Single processor,
Shared memory and
Distributed memory.

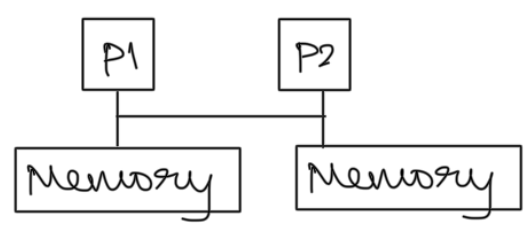
(i.) Single:



(ii.) Shared



(iii.) Distributed

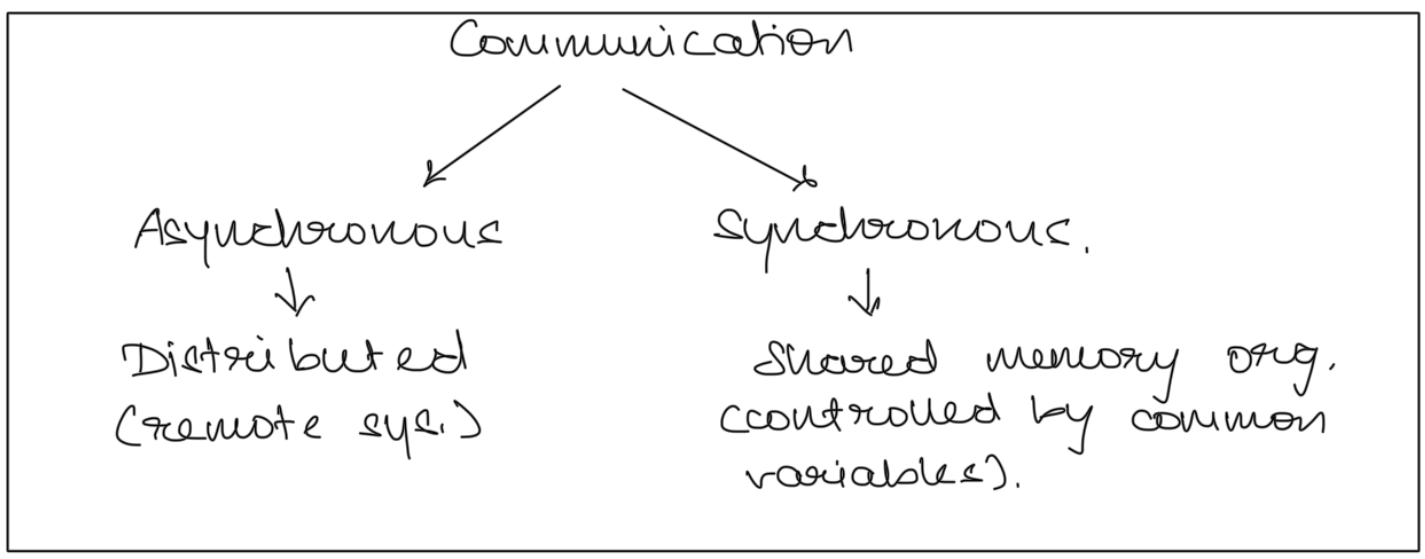


Interrupt:

Can be from system → system interrupts (OR)
from user → user interrupts.

Priority is higher for system interrupts.

Multiprocessor Organization:



Interactive System:

Defn.: System works based on user interaction.

Also known as reactive systems.
↳ Majorly used in gaming applications.

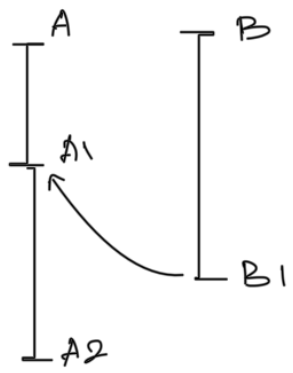
Streams:

Sequence of values that flows from one process

to another.

Involves the concept of pipelining.

Eg.



Streams (4 marks).

Process Network: pipe construct (Sequential / concurrent).

Questions:

1. Concurrency vs. Parallel programming

2. Advantages of: concurrent programming, multi threading, multiprocessing, cache coherence, race conditions, deadlock → characteristics.

3. How imperative and functional programming help in concurrency.

4. Parallelism in hardware + Challenges.

5. Solution to defend parallelism in hardware.

6. Pipeline construct and synchronization
   ↳ Implicit vs. Explicit synchronization

7. Process interleaving

8. Two modes of communication in concurrent programming paradigms (Syn / Asyn.)

9. Message passing + communication channel and process networks

10. What are streams

11. What are co-routines

S.NO	Concurrency	Parallelism
1.	Concurrency is the task of running and managing the multiple computations at the same time.	While parallelism is the task of running multiple computations simultaneously.
2.	Concurrency is achieved through the interleaving operation of processes on the central processing unit(CPU) or in other words by the context switching.	While it is achieved by through multiple central processing units(CPUs).
3.	Concurrency can be done by using a single processing unit.	While this can't be done by using a single processing unit. it needs multiple processing units.
4.	Concurrency increases the amount of work finished at a time.	While it improves the throughput and computational speed of the system.
5.	Concurrency deals with a lot of things simultaneously.	While it does a lot of things simultaneously.
6.	Concurrency is the non-deterministic control flow approach.	While it is deterministic control flow approach.
7.	In concurrency debugging is very hard.	While in this debugging is also hard but simple than concurrency.

Mutual exclusion
Hold and wait
Non preemptive
Circular wait

5- Cache coherence:

Multiple processes share same memory => way to organize so that the process is co-ordinated.

semaphores => Binary and mutex are utilized.

7. Deadlock => condition where processes wait only due to improper resource allocation.

↓

Mutual exclusion
Hold and wait
Non preemptive
Circular wait

---

2. Need efficiency, throughput => many programs executed in a single unit of time

3. Multi threading:

Handling multiple threads simultaneously
↳ >1 user can utilize → support multitasking.

one - one ; one - many ; many - many

Improves performance + responsiveness.

4. Multiprocessing: many processes simultaneously.

Concurrent Programming:

(i.) Co Begin

(ii.) Parallel Loops

(iii.) Launch at Elaboration

(iv) Fork / Join (primitive constructs)
↳ x in Unix

(v) Implicit Receipt

(vi) Early Reply

↳ can be based on software / hardware → Parallelism } throughput ↑s

↳ can be sequential / functional / Imperative
↓
traditional

ADA: Adaptive Differential Analyzer

Objective: logical structure → extra processes → separate physical devices.

Concurrency as interleaving → threads / events

Process: running instance and task is said to be workable unit

Program (os)
↓ Running instance
Process (PL)
↓ Submit
Threads
↓
Single / Multi processor

} slow concurrency

Schedulers / Dispatchers → put a thread / process to sleep.

Long term vs. Short term

Parallel loops → loops called recursively.

Launch - at - elaboration → subroutine + no para.

Fork → 'join' is optional

checks if the processes have been closed.

Implicit receipt → thread in response to an incoming request

Early Reply → for scheduler to respond to concurrency

Busy waiting, spin lock, polling, dead/ live lock + interrupt, fairness.

12th chapter in Ravi Sethi

Answers:

1. Polling is a technique where a program repeatedly checks the status of an external device.

It is the process of repeatedly sampling the status of external device by a client program.

2. Busy Waiting:
   - it is a technique where a process repeatedly checks if a condition is true.
   - It is an efficient soln when waits are short and can be wasteful when waits are long.
   - It is a synchronisation technique.

3. Spin Lock:
   - A synchronisation technique where a thread continuously polls the lock until it becomes available
   - A low-level synchronisation mechanism suitable primary for use on shared memory multiprocessor.

4. Dead Lock:
   - It is a situation where multiple threads or processes are unable to proceed because they are waiting for each other to release a resource.
   - Can occur in multiprocessing / parallel / distributed system
   - It can freeze, crash or consume too many resources.

5. Live lock:
   - A situation where multiple program / threads are able to execute but the system as a whole is unable to progress.
   - Happens when threads are stuck in a loop.

- Dining Philosophers' problem → helps in concurrency by giving soln + synchronization.
- Think / Hungry / Eat 3 states of a philosopher
- Helps in resource partitioning
  ↳ But might lead to deadlock, livelock and fairness.

Monitor abstract, Binary semaphore and mutex -> assist in solving the problem.

Livelock: resources reallocation -> but no real progress made.

Sol access to shared data -> nondeterministic process.

Syn. primitives: locks, mutex, semaphores, cond" vari.
↳ Eg, locks and mutex → in order to enter critical section.

Also, semaphores and condition variables
↳ global with one variable

Two constraints to flush in/out the intermediate result and also the final result.
↳ close the result and make an exit point
Binary: decrement or negative answer.

caller is waiting and callee has closed 3
↳ lock and unlock
↳ Monitor DS

Arguements into abstract medium ⇒ monitor execution.

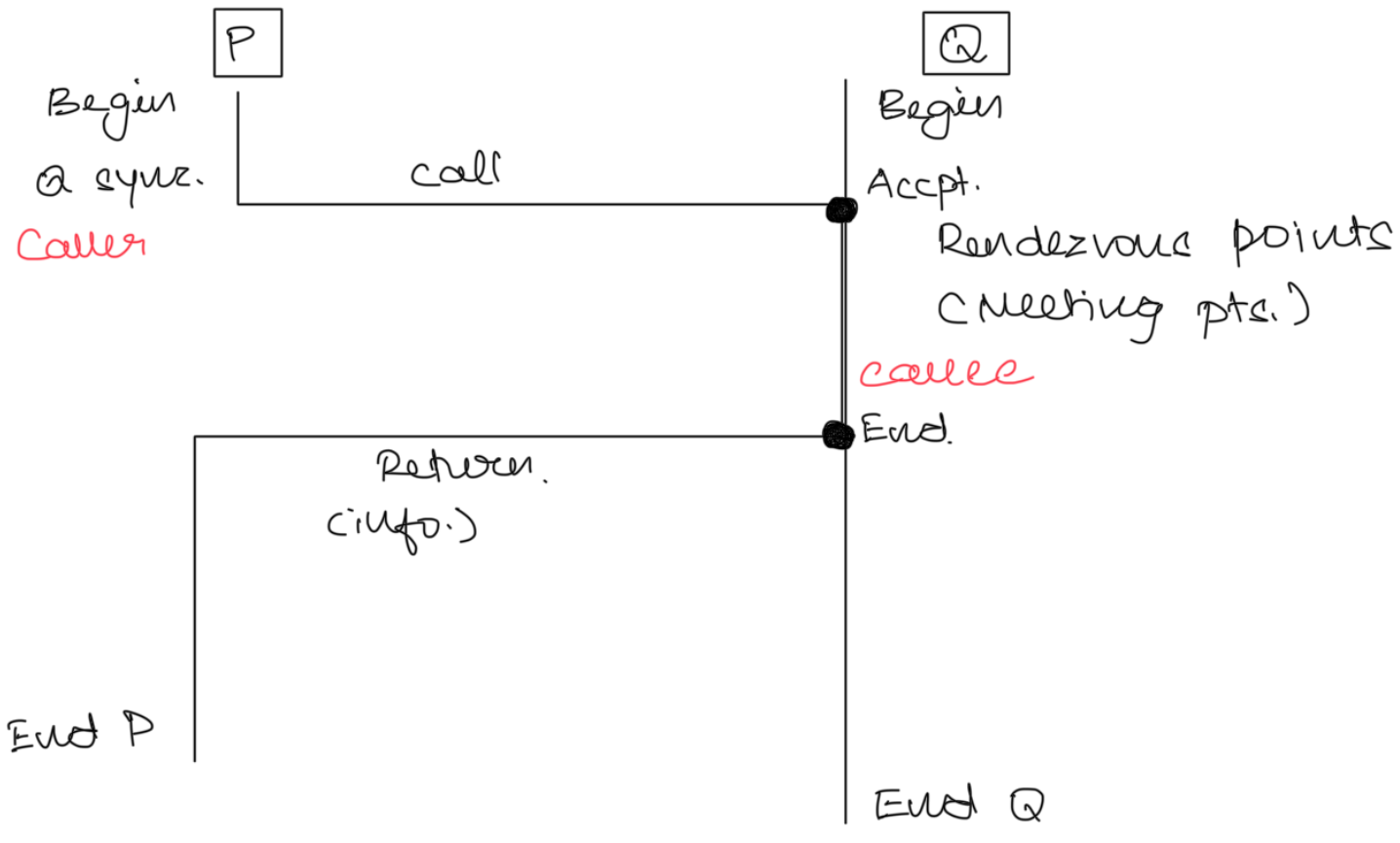
Safety as serializability ⇒ OS Book (Ravi Sethi)

Syn. by rendezvous → eg. to sequence diagram
↳ UML diagram

UML diagram

caller is the server ↳ client-server model.

Usually in Ada. (From caller to callee)



Concurrency in Ada:

- Locks/Unlocks (or) Wait/Signal	Sync in Rem.
- Accept <entry name> do ... (Body) End <entry name>	Rem. (procedure in call in ADA)

If Rendezvous is performed correctly, program runs smoothly
↳ P and Q process that communicate
↳ keywords and primitives.

Different Scenarios:

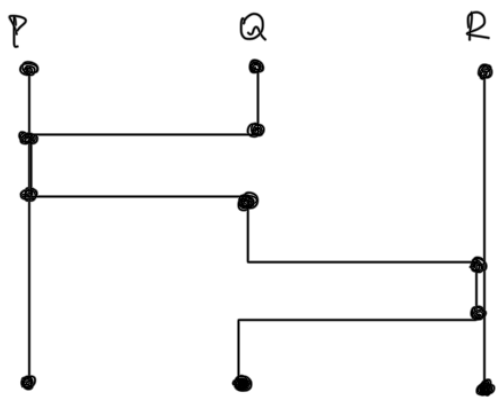
- If the client calls before the server is ready to accept, then the client waits until the rendezvous can occur.

- If the server reaches an accept statement before the client calls, then server waits till rendezvous can occur.

Synchronized Communication:

- Sync. and communication during a ren.
  ↳ Example Ada code.

Flow diagram:



Task init → init
Dyn. created Tasks:
p := new emitter
q := new emitter

```
select
    accept deliver_milk do
    ...
    end deliver_milk;
or
    accept deliver_juice do
    ...
    end deliver_juice;
end select;
```

Buffer as a flow:

```
select
when not full => accept enter (c: in character) do
...
end enter;
or
when not empty => accept leave
```

↳ Buffer as a process

```
task body buffer is
begin
    loop
        select
            when not full =>
                accept enter(x: in integer) do
```

```
            ...
            end entry
        or when not empty =>
            accept leave (x: out integer) do
            ...
            end leave
        end select
    end loop
end buffer.
```

& ppt + Chap-12 (Ravi Sethi)

Module 5: Functional Programming

Module 6: Logical Programming

Module 7: Scripting Structures

- Basics of Web design and Scripting Language
- Case study: Java Script

No Textbook

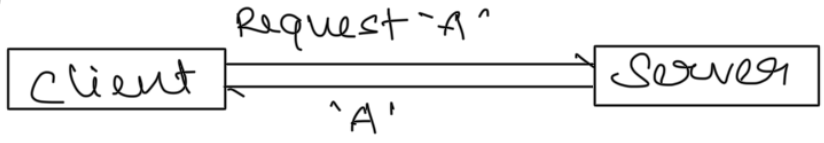
- scripting only in front end

↓ components:

Web browser arch, web servers, appn. servers, types of webpages, web essentials.

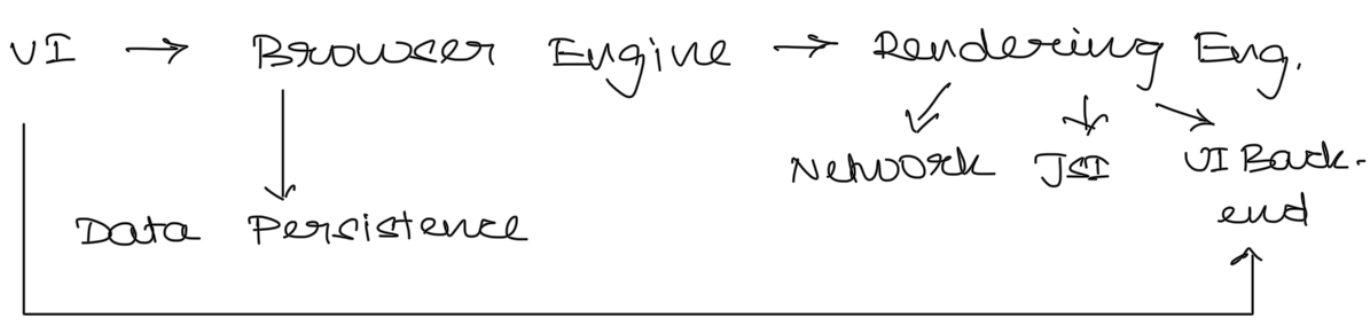
Web Essentials:

- Client: web browsers.
- Servers: systems used for info. supply.
- Computer networks: browser - server comm^n.



HTTP URL: Host (FQDN) + Port + Path + Query + Fragment

(Authority and only then Request URL)



Web server: (Not the same as domain server)

- Directly linked with web clients.

- URI: Uniform Resource Identifier
  URL: Uniform Resource Locator.

Web Architechure and Web client/ server
↳ primitive.

Now: package, compartment and model
↳ UI.

MVC - Model - View - Controller

Eg. command version, URL, HTTP.

↳ Full stack and minstack architechure.

XML: Extensive Markup Language
Decentralized DB for data exchange.

HTTP Response from Web server

HTTP Request Types:

- GET (default) and Post do the same task essentially.

  Send data from client to server

  Yellowpages → predefined set up repository
  ↓
  ≃ Telephone directory replacement
  (for business services)

  content mgmt. system.

All URLs are URIs, but the converse is not true

Web Client ⟶ Web Server
Rendering Engine

Application Server

Multi-tier application ⇒ n-tier application

DTD: Doc Type Design
↳ with the strict tabs

Exam: 50-60 tags, inline, internal and external css
↳ Embed html and css files.

DTD: Document Type Declaration

Creation of a simple webspage + authentication from client's side.
(without DB / Backend).

From exam pov, structure matters.
↳ web app to demonstrate navigation

CSS: design and layout of webpages.

Benefits: less coding, more styling, std., Better performance.

XML is NOT a replacement for HTML.
Also, XML is not a Language in itself.

Well-formed XML Documents and XML Namespaces

XML namespace:

<element xmlns: name = "URL">
↳ name space

Avoid element name conflict

Namespace Prefix: cont

Syntax: DTD → Doctype

<! DOCTYPE element DTD identifier>

[
  declaration 1
  declaration 2 ...
]

Internal DTD & Elements within XML files

External DTD, DTD file content → student.dtd as follows

#PCDATA

```
<BookList>
<Book Genre = "Science">
  :
<!ATTLIST BOOK FORMAT>
```

Elemental Declarations in DTDs

↳ Attribute declaration

Write a simple implementation of a mgmt. system html, CSS, XML, DTD & use the above

Provide front end and so html tags and CSS files: inline, internal and external

↳ 5x3 matrix

Simple elemental defn.