UNIT – II

Introduction to Problem Solving

FYBTECH

Subject : Programming and Problem Solving

Unit 2 Contents

- Problem solving process/framework, Algorithms, Pseudo-code and flowchart, Case study for Algorithm, flowchart and pseudo code: calculate slope of a line, Factorial, Fibonacci, snake and ladder, tic-tac-toe. Top down and Bottom up design approach, Software Development life cycle, Programming paradigms: Imperative, object oriented, functional and logic programming. Role of programming languages, need to study programming languages, Characteristics of Programming Languages.

Computer Programming– Three Stages

- Computer programming can be divided into three distinct areas of activity:
  1. Problem definition and problem solving
  2. Creating a structured solution (or algorithm)
  3. Coding (e.g. C, Java, C++)
- Consider an example as case of customer while marketing
- Specify the problem
- Solve the problem
- Specify solution in structured format

Problem Solving, Algorithm Development and Coding

- In problem solving, algorithm development and coding we tend to view problems at different levels of abstraction.
  - **Problem solving** is done at a high level of abstraction. Much of the detail is omitted at this stage in order that the problem can be more easily specified and solved.
  - **Algorithm development** works at a lower level of abstraction than problem solving. It contains much of the detail that will eventually become the program. However, it still omits the finer detail required in the final compute program
  - **Coding** works at a very low level of abstraction. Programming code must be written with every single detail a computer needs to carry out a task included.

Contd…

- Problem solving, algorithm development and coding are also different in terms of the language that is used to complete the activity:
  - Problem solving is generally done using **use natural, everyday language**
  - Algorithm development is done using a specialist, semi-structured language (**pseudo code**)
  - Coding is done using a highly-structured (**programming**) language that is readable by machines

What is a Problem?

- There are many different kinds of problems:
  - How do I get to Oxford from London?
  - Why is it wrong to lie?
- For a problem to be solvable through the use of a computer, it must generally:
  - Be technical in nature with **objective** answers that can be arrived at using **rational** methods
  - Be **well-structured**
  - Contain **sufficient information** for a solution to be found
  - Contain little, if any, **ambiguity.**

When Can You Use a Problem Definition?

- Starting Point
- Expected Outcome
- Possible Procedure

How do We Solve Problems?

- We need to THINK!
- We need to engage in one or more of the following types of thinking:
  - **Logical reasoning**
  - **Mathematical reasoning**
  - **Lateral thinking**
- Problem solving is easier if we employ a **problem solving framework** and an appropriate **problem solving strategy to** aid us.

A Problem Solving Framework

- An outline framework for problem solving:
  1. Understand the problem
  2. Devise a plan to solve the problem
  3. Carry out the plan
  4. Assess the result
  5. Describe what has been learned from the process
  6. Document the solution.

- In Vickers, this framework is called the **How to Think Like a Programmer** (HTTLAP) approach to problem solving.

Understanding the Problem

- To understand a problem
  - We need to read and reread it till we understand every detail
  - We need to dissect the problem into its component parts (e.g. problems and **sub-problems**)
  - We need to remove any **ambiguity**
  - We need to remove any information that is **extraneous** to the problem
  - We need to determine our **knowns** and our **unknowns**
  - We need to be aware of any **assumptions** we are making.

Devise a plan to the solve the problem

- If a problem contains a set of sub-problems, in what order are you going to solve them?
- How are you going to represent the problem:
  - Numerically?
  - Graphically?
  - Tabular data?
  - Natural language?
- Does the problem lend itself to a particular problem solving strategy or strategies:
  - Working backwards?
  - Logical reasoning?
  - Finding a pattern?
  - Accounting for all possibilities?

Carry Out the Plan

- Consider the following problem:

  In a room with ten people, everyone shakes hands with everyone else exactly once. In total, how many handshakes are there?

- How would you represent and solve the problem?
- Different strategies to be used

Assessing the Results

- It is very unusual when solving complex problems to achieve the correct result first time round.
- To verify our solutions are correct, we need to take a few steps backwards:
  - Was our understanding of the problem correct?
  - Did we overlook anything?
  - Did we choose the correct strategy?
  - Did we employ that strategy correctly?
  - Have we made any incorrect or unwitting assumptions?
- However, it is often very difficult to spot our own mistakes. It is often better, therefore, to have somebody else verify our solutions for us.

Contd…

- Sometimes solutions appear correct, but are in fact wrong, due to an initial misunderstanding of a problem (What Vickers calls, **errors of the third kind**).
- If you have misunderstood a problem, it does not matter how good a coder you are, your program will not work as it is supposed to.
- Therefore getting the problem-solving part of programming right is absolutely essential if we are to build programs that work as they are supposed to work.

Describing What you have Learned

- You can only become a good problem solver by reflecting on your experiences of problem solving.
- Keeping a record of problems you have attempted, your success, failures, the approaches you have used, etc. will:
  - Broaden your problem solving repertoire
  - Help you to recognize similarities/patterns in problems
  - Help you identify and fix logical or implementation errors
  - Help you to solve problems faster and more effectively

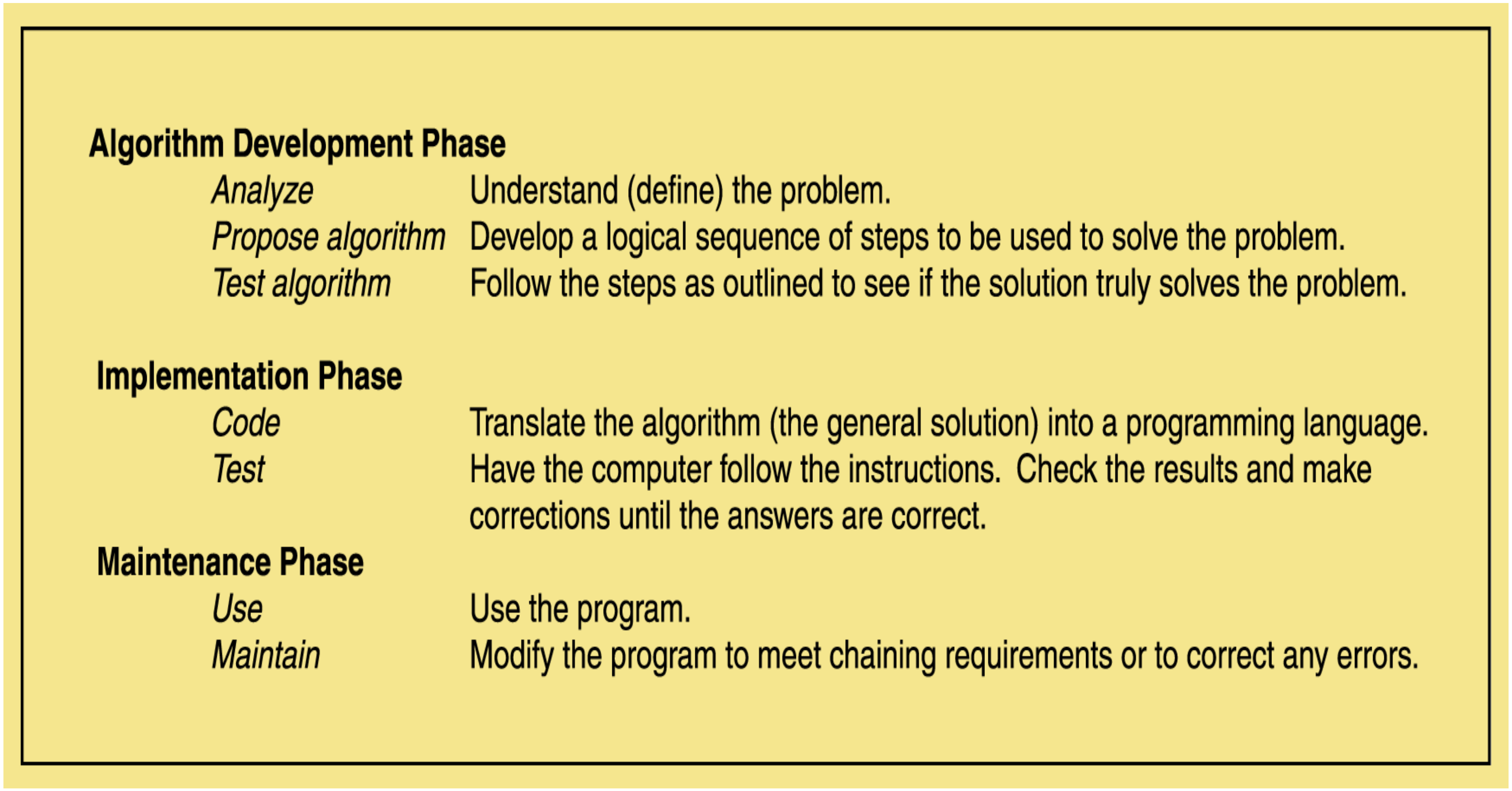
Documenting the Solution

- Documenting a solution will help you to **generalize** your approach to other similar problems.
- Do you recognize the following type of problem? How would you solve it?

Carlos and his friends have a fantasy football league in which each team will play each other three times. The teams are Medellin, Cali, Antigua, Leon, Juarez, Quito and Lima. How many games will be played in all?

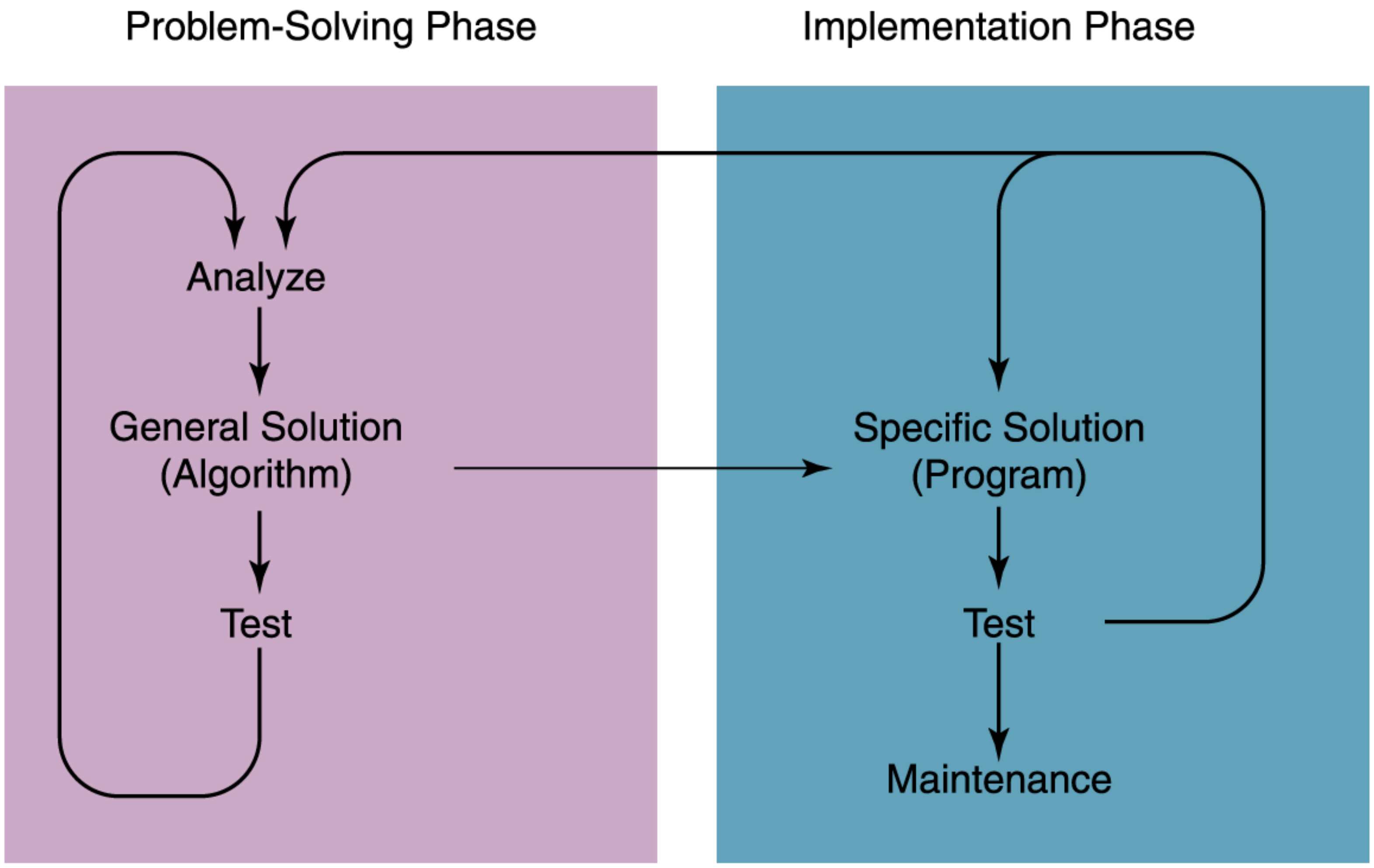
- It will also prevent you forgetting how you arrived at your solutions.
- In the long run, it will make you a better problem solver and eventually a better programmer.

Computer Problem-Solving



Algorithm Development Phase		
	Analyze	Understand (define) the problem.
	Propose algorithm	Develop a logical sequence of steps to be used to solve the problem.
	Test algorithm	Follow the steps as outlined to see if the solution truly solves the problem.
Implementation Phase		
	Code	Translate the algorithm (the general solution) into a programming language.
	Test	Have the computer follow the instructions. Check the results and make corrections until the answers are correct.
Maintenance Phase		
	Use	Use the program.
	Maintain	Modify the program to meet chaining requirements or to correct any errors.

The Interactions Between Problem-Solving Phases



Conveying solution in a formal language

Purpose of program planning

- To write a correct program, programmer must write each and every instruction in the correct sequence.
- Logic (instruction sequence)of the program can be very complex.
- Hence program must be planned before they are written to ensure program instructions are:
- Appropriate for the problem
- In the correct sequence

Algorithm, Flowchart and Pseudo Code

- A typical programming task can be divided into two phases:

1. Problem solving phase

- **produce an ordered sequence of steps** that describe solution of problem
- this sequence of steps is called an ***algorithm***

2. Implementation phase

- implement the program in some programming language

Algorithm, Flowchart and Pseudo Code

- Algorithms, Flowcharts and Pseudo Codes are different tools used for creating new programs, especially in computer programming
- Algorithm
  - Set of step-by-step instructions that perform a specific task or operation
  - It is a Natural language and NOT programming language
- Flowchart
  - Visual program design tool
  - Semantic symbols describe operations to be performed
- Pseudo code
  - Set of instructions that mimic programming language instructions

What is an Algorithm?

- Step-by-step description of how to achieve a solution for a given problem
- refers to the logic of the program or sequence of instructions
- It is defined as a sequence of instructions that when executed in the specified sequence, the desired results are obtained.

Algorithm

- Algorithm is a representation of a solution to a problem
- It is commonly used for data processing, calculation and other related computer and mathematical operations
- To be an algorithm, a set of rules must be unambiguous and have a clear stopping point

Characteristics of an Algorithm

- Should be precise and unambiguous
- Should be executed in a finite time
- Should not be repeated infinitely. algorithm will ultimately terminate.
- Desired results are obtained
- Must have start and stop steps

Example

- Algorithm to add two given numbers:

```
Step 1: START
Step 2: Read two numbers A and B
Step 3: Add numbers A and B and store result  in C
Step 4: Display C
Step 5: STOP
```

Example

Write an algorithm to find the largest among three different numbers entered by user.

```
 Step 1:START
Step 2:Declare variables a,b and c.
Step 3:Read variables a,b and c.
Step 4:If a>b
              If a>c
                   Display a is the largest number.
            Else
                  Display c is the largest number.
```

Contd...

```
Else
        If b>c
            Display b is the largest number.
    Else
            Display c is the largest number.
Step 5:STOP
```

Generalized Algorithm

- Logic written in universal or general terms
- Algorithms are written in general languages which can be used by any one
- Knowledge of programming language is not required
- Only solution of problem is given, implementation details are not required
- Becomes easy to find solution
- Anyone can refer this algorithms

How to Make Algorithms Generalized

- Solve problem by person who will not be expert in implementation
- Use generalized logic to solve problem

Infinite Loop

- Loop mean repeat set of instruction till specific condition is true
- Infinite loop is endless loop, i.e. it never ends
- Infinite loop occurs when condition remains always true or no termination condition exists
- Due to infinite loop, program never ends

Avoiding Infinite Loops

Avoided by limiting the number of repetitions of loop

Two ways:

- By counting fixed count is used for repeating loop
- By using sentinel value It is special value whose presence guarantees termination of loop

Different Ways to Represent an Algorithm

- As a flowchart
- As a pseudocode
- As a program

Program Planning Tools

- Flowchart
- Structure charts
- Pseudo codes

Flowchart

- Flowchart is a tool developed in the computer industry, for showing the steps involved in a process

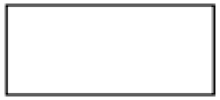
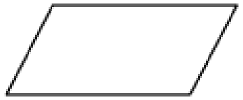
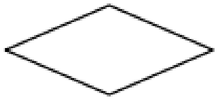
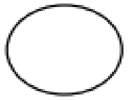
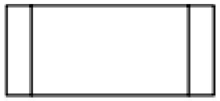
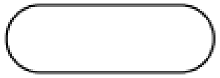
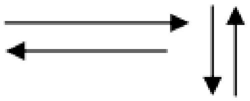
- **A flowchart is a diagram made up of boxes, diamonds and other shapes, connected by arrows** - each shape represents a step in the process, and the arrows show the order in which they occur

- Flowcharting Symbols:
  - There are 6 basic symbols commonly used in flowcharting of assembly language programs: Terminal, Process, input/output, Decision, Connector and Predefined Process

Flowchart

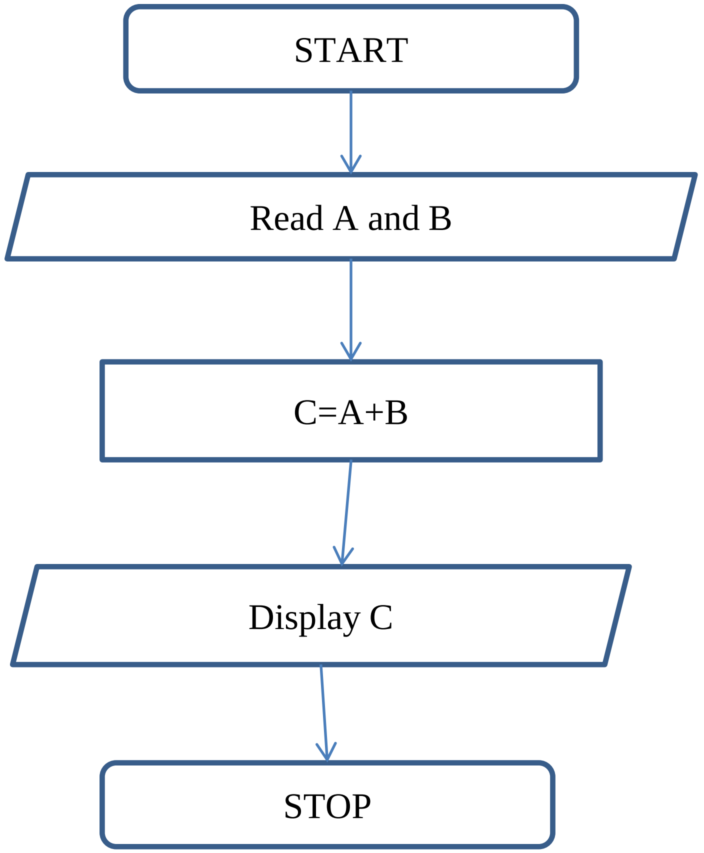
- Pictorial representation of an algorithm
- Steps of an algorithm are shown in different shapes and logical flow is indicated by arrows
- Boxes represent different operations
- Arrows shows sequence of operations
- Helps to understand logic of program

Flowchart Symbols

Symbol	Name	Function
	Process	Indicates any type of internal operation inside the Processor or Memory
	input/output	Used for any Input / Output (I/O) operation. Indicates that the computer is to obtain data or output results
	Decision	Used to ask a question that can be answered in a binary format (Yes/No, True/False)
	Connector	Allows the flowchart to be drawn without intersecting lines or without a reverse flow.
	Predefined Process	Used to invoke a subroutine or an interrupt program.
	Terminal	Indicates the starting or ending of the program, process, or interrupt program.
	Flow Lines	Shows direction of flow.

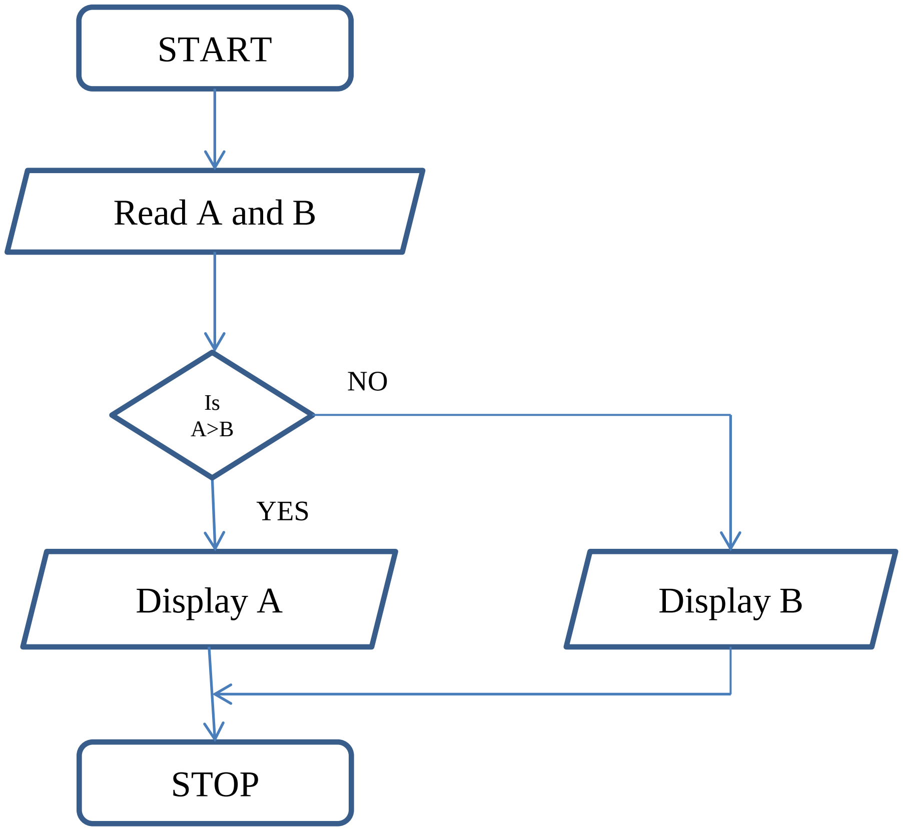
Flowchart Examples

Add two numbers and display result

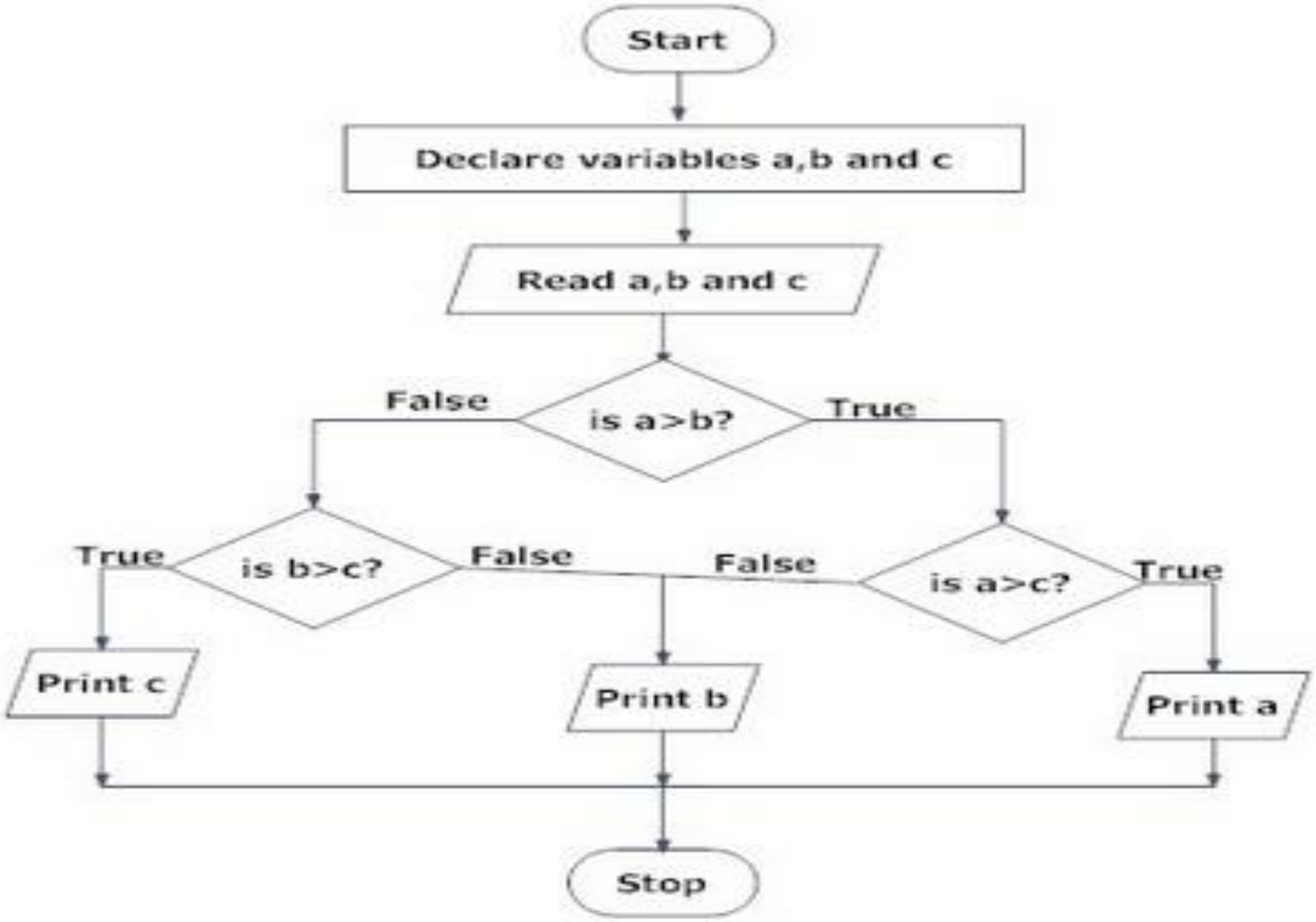


Flowchart Examples

Find the largest of given two numbers



Draw flowchart to find the largest among three different numbers entered by user.



Advantages and Limitations

- Advantages:
  - Better Communication
  - Proper Program Documentation
  - Effective Coding
  - Systematic Debugging
  - Systematic Testing
- Limitations:
  - Time consuming
  - Difficult to modify
  - No update
  - No standards

Pseudocodes

- “Pseudo” – imitation or false
- “Code” – instructions written in a programming language
- Combination of English and Programming Language
- Focuses on developing the logic of the program without worrying about the syntax

Pseudo Code

- Pseudo code is one of the tools that can be used to write a **preliminary plan** that can be developed into a computer program

- Pseudo code is a generic way of describing an algorithm without use of any specific programming language syntax

- It is, as the name suggests, pseudo code i.e. **it cannot be executed** on a real computer, but it models and resembles real programming code, and is written at roughly the same level of detail

- Computer science textbooks often use pseudo code in their examples so that all programmers can understand them, even if they do not all know the same programming languages

Example

Accept two numbers :-

```
if first number > second number
    display "First number is greater"
else
    display " second number is greater"
```

Advantages and Limitations

- Advantages
  - Focuses on logic of program without worrying about syntax
  - Language independent; can be translated to any computer language code
  - Easier to develop program from pseudocode than flowchart
  - More concise, readable, and easier to modify
- Limitations
  - No visual representation of program logic
  - No standards
  - Cannot be compiled and executed hence correctness cannot be verified

Example:
Algorithm, Flowchart and Pseudo Code

- Determine a student's final grade indicating whether he/she is passing or failing. The final grade is calculated as the average of four marks.

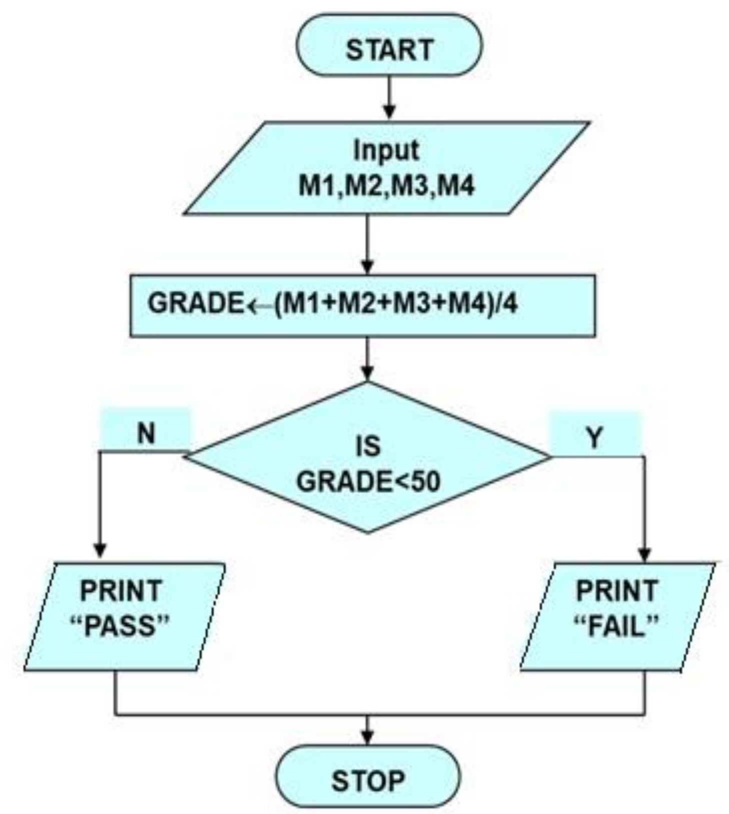
Algorithm:

```
Step 1: Start
Step 2: Input a set of four marks
Step 3: Calculate the grade by adding and dividing by 4
Step 4: if grade is below 50
            Print "FAIL"
        else
            Print "PASS"
Step 5: Stop
```

Pseudo code:

```
Read M1,M2,M3,M4
GRADE ← (M1+M2+M3+M4)/4
 if (GRADE < 50) then
     Print "FAIL"
  else
     Print "PASS"
  endif
```

Flowchart:

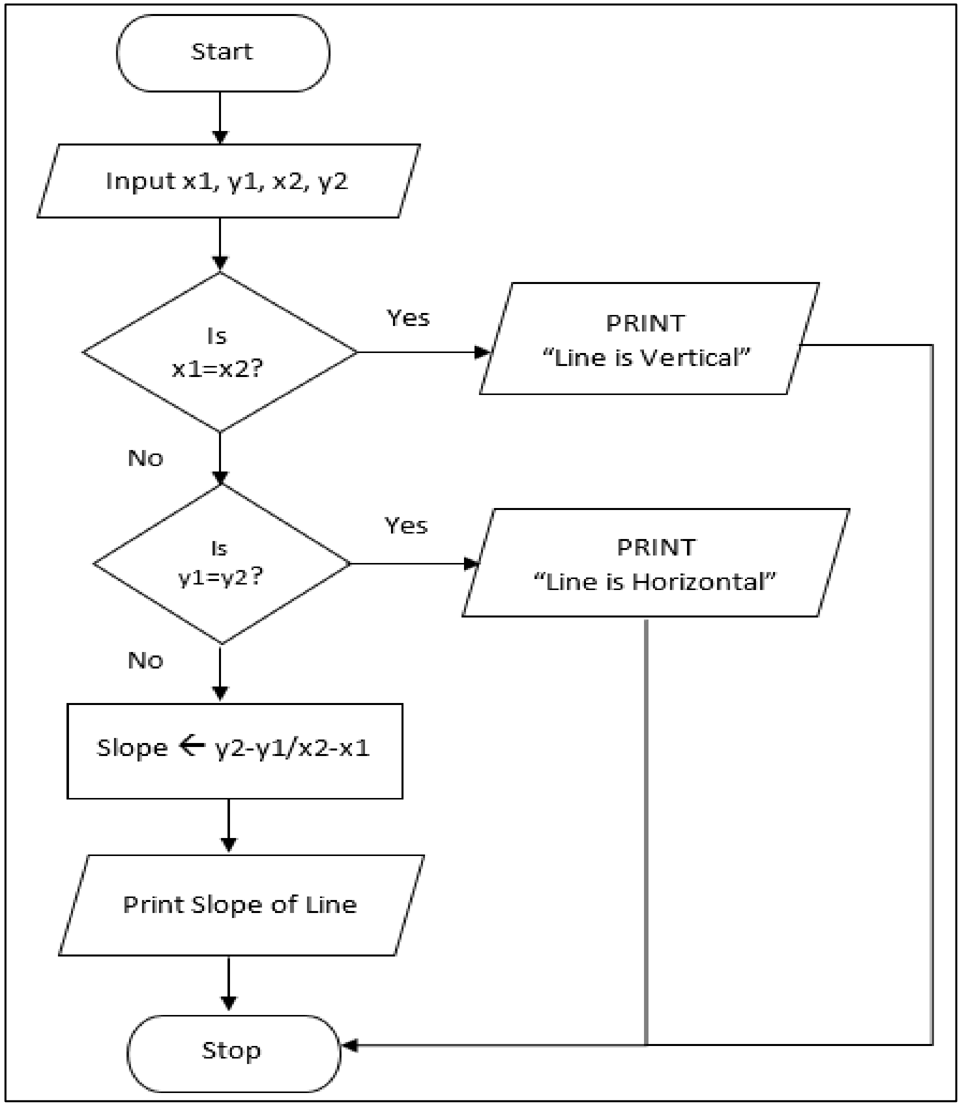


Case Study

Algorithm and Flowchart to calculate slope of a line

Algorithm:

```
Step 1: Start

Step 2: Input x and y coordinates of line as
        x1, y1, x2, y2

Step 3: if x1 and x2 are equal then
            Print “Line is Vertical“
            Go to Step 6
        else if y1 and y2 are equal then
            Print “Line is Horizontal“
            Go to Step 6
        else Go to Step 4

Step 4: Find Slope as y2-y1/x2-x1

Step 5: Print Slope of Line

Step 6: Stop
```



Case Study

Algorithm and Flowchart to find factorial of a number

Algorithm:

Step 1 : Start

Step 2 : Read n

Step 3 : Initialize counter variable i to 1 and fact to 1

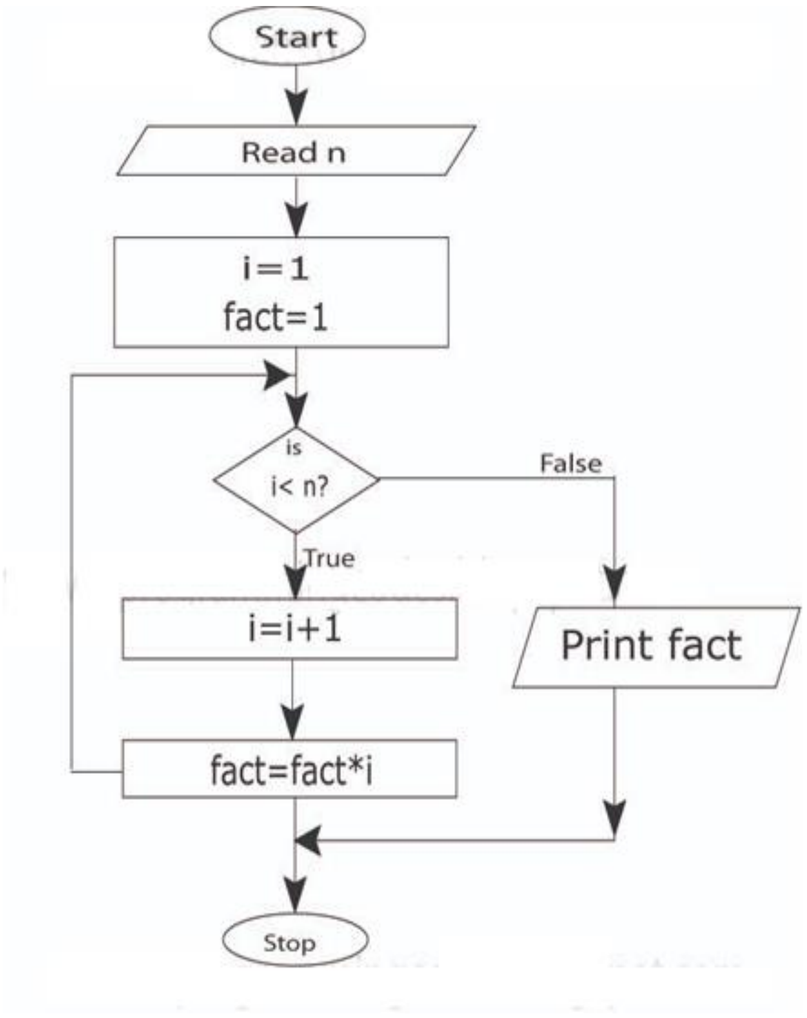
Step 4 : if i < n go to step 5 otherwise goto step 7

Step 5 : increment counter variable i

Step 6 : calculate fact = fact * i and goto step 4

Step 7 : Print fact

Step 8 : Stop



Case Study

Pseudo code to find factorial of a number

Pseudo code:

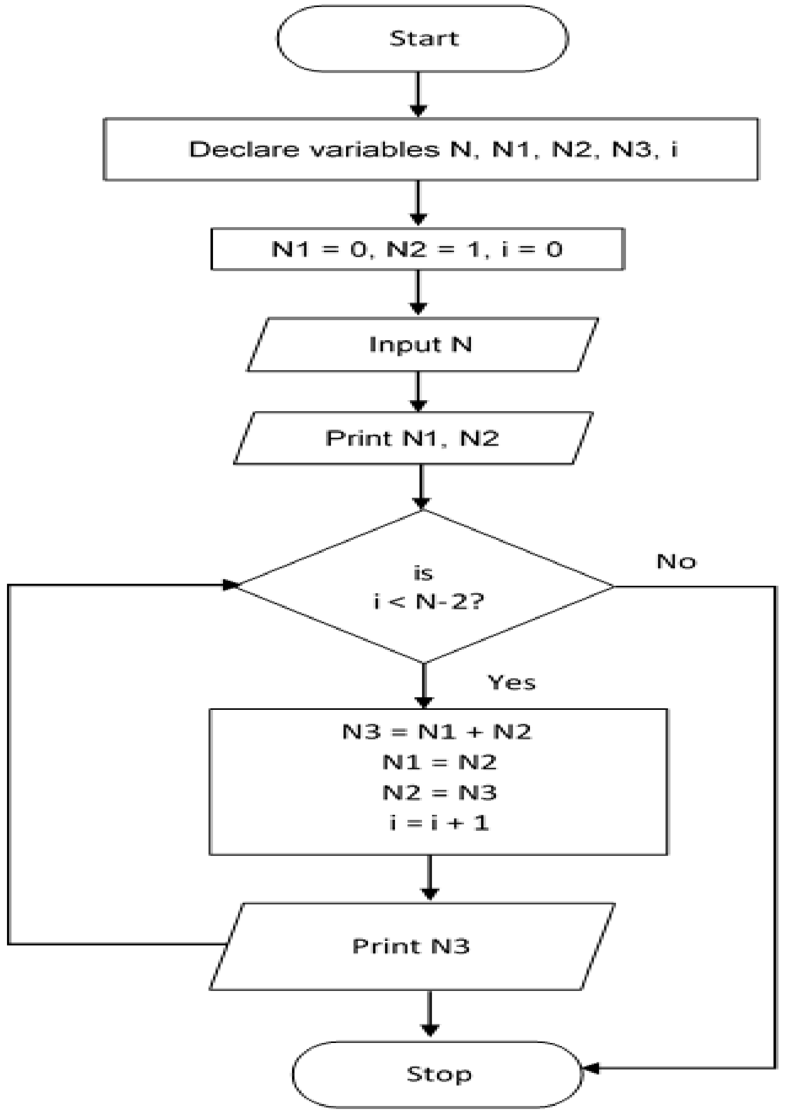
```
□ Input a number n
□ Initialize counter variable i =1 and factorial=1
□ while i <= n
        factorial = factorial * i
        i=i+1
□ Print factorial of a number
```

Case Study

Algorithm and Flowchart to find Fibonacci Series

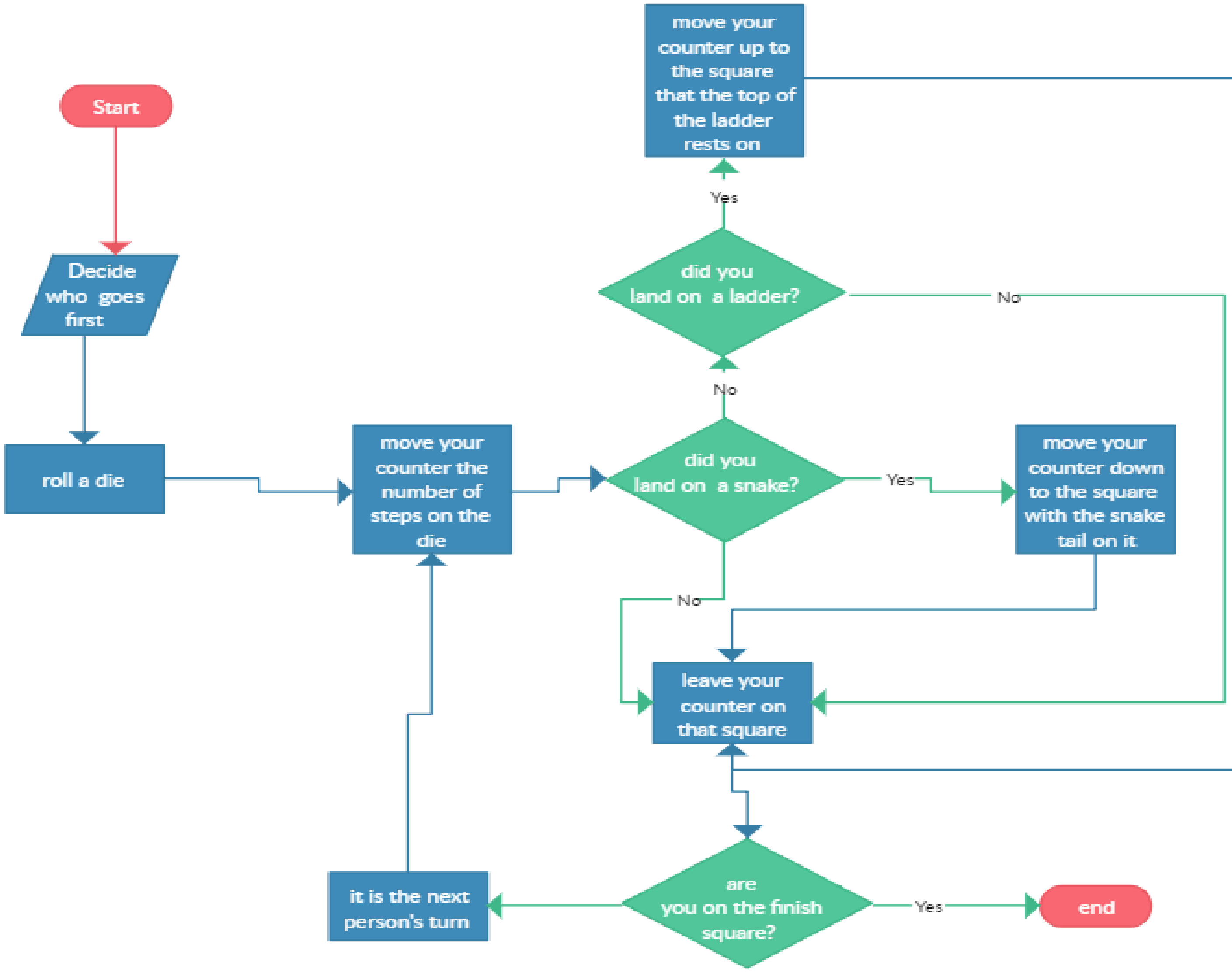
Algorithm:

```
Step 1: Start

Step 2: Declare variables N, N1, N2, N3, i

Step 3: Initialize variables N1 = 0, N2 = 1, i = 0

Step 4: Read value of N from user

Step 5: Display N1, N2

Step 6: Repeat following statements until i <  N - 2
            Compute N3 = N1 + N2
            N1 = N2
            N2 = N3
            i  = i + 1
            Display N3

Step 7: Stop
```



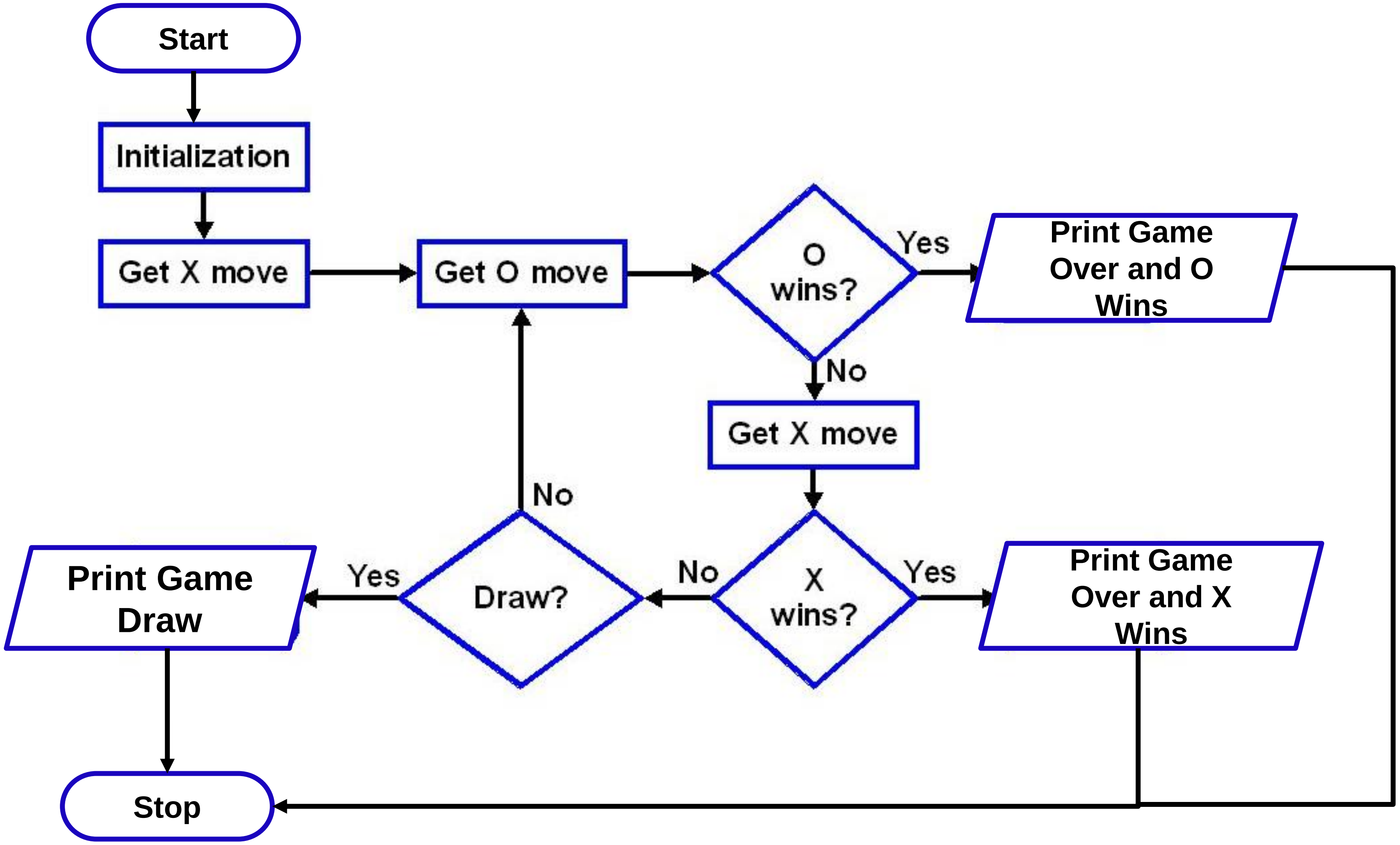
Case Study

Flowchart for Snake and Ladder


Start
Decide who goes first
roll a die
move your counter the number of steps on the die
did you land on a snake?
Yes
move your counter down to the square with the snake tail on it
No
did you land on a ladder?
Yes
move your counter up to the square that the top of the ladder rests on
No
No
leave your counter on that square
are you on the finish square?
Yes
end
it is the next person's turn

Case Study

Flowchart for Tic-Tac-Toe



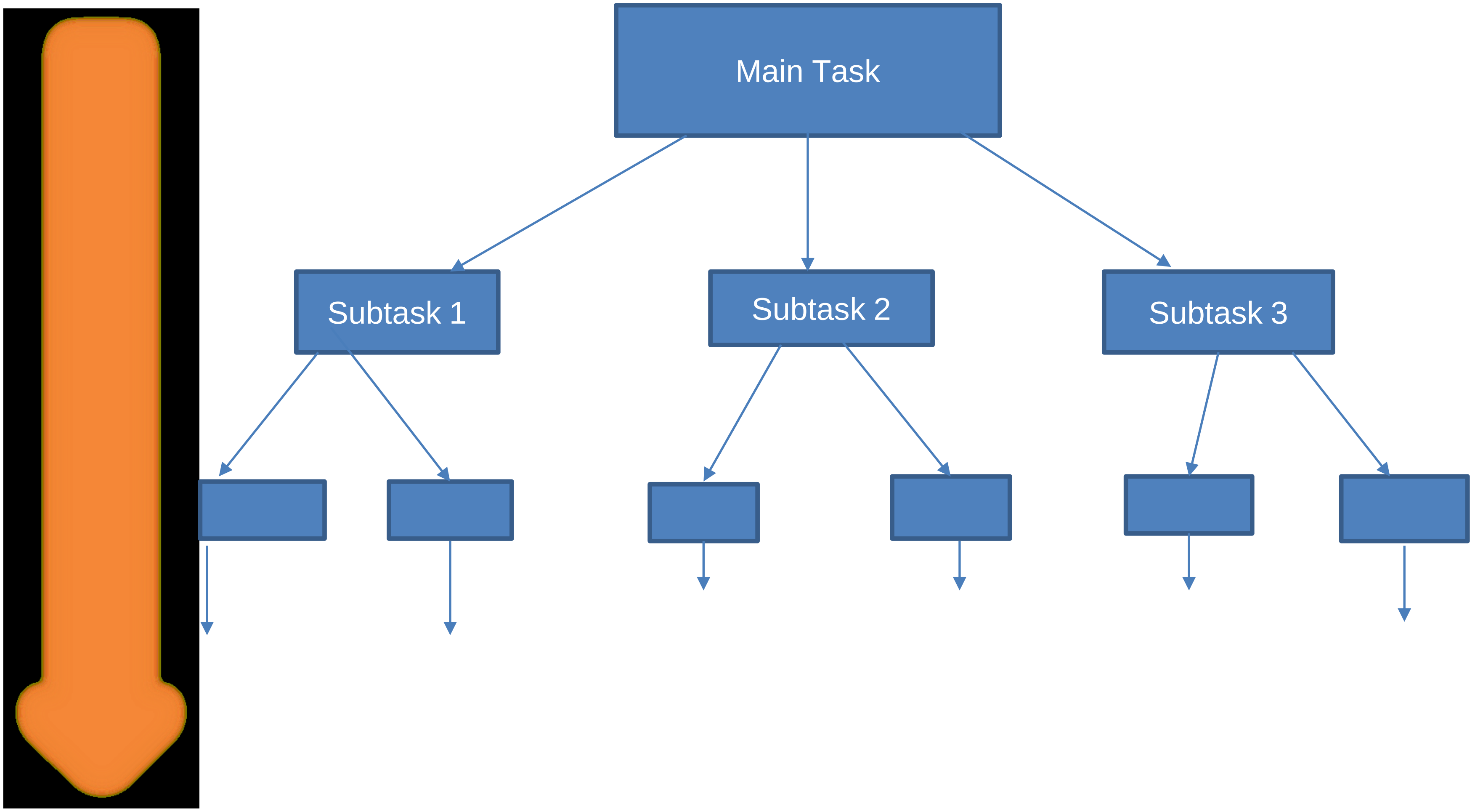
Top down Design Approach

- **Top down Design Model**
- In top-down model, an overview of the system is formulated, without going into detail for any part of it.
- Each part of the system is then refined in more details.
- Each new part may then be refined again, defining it in yet more details until the entire specification is detailed enough to validate the model.

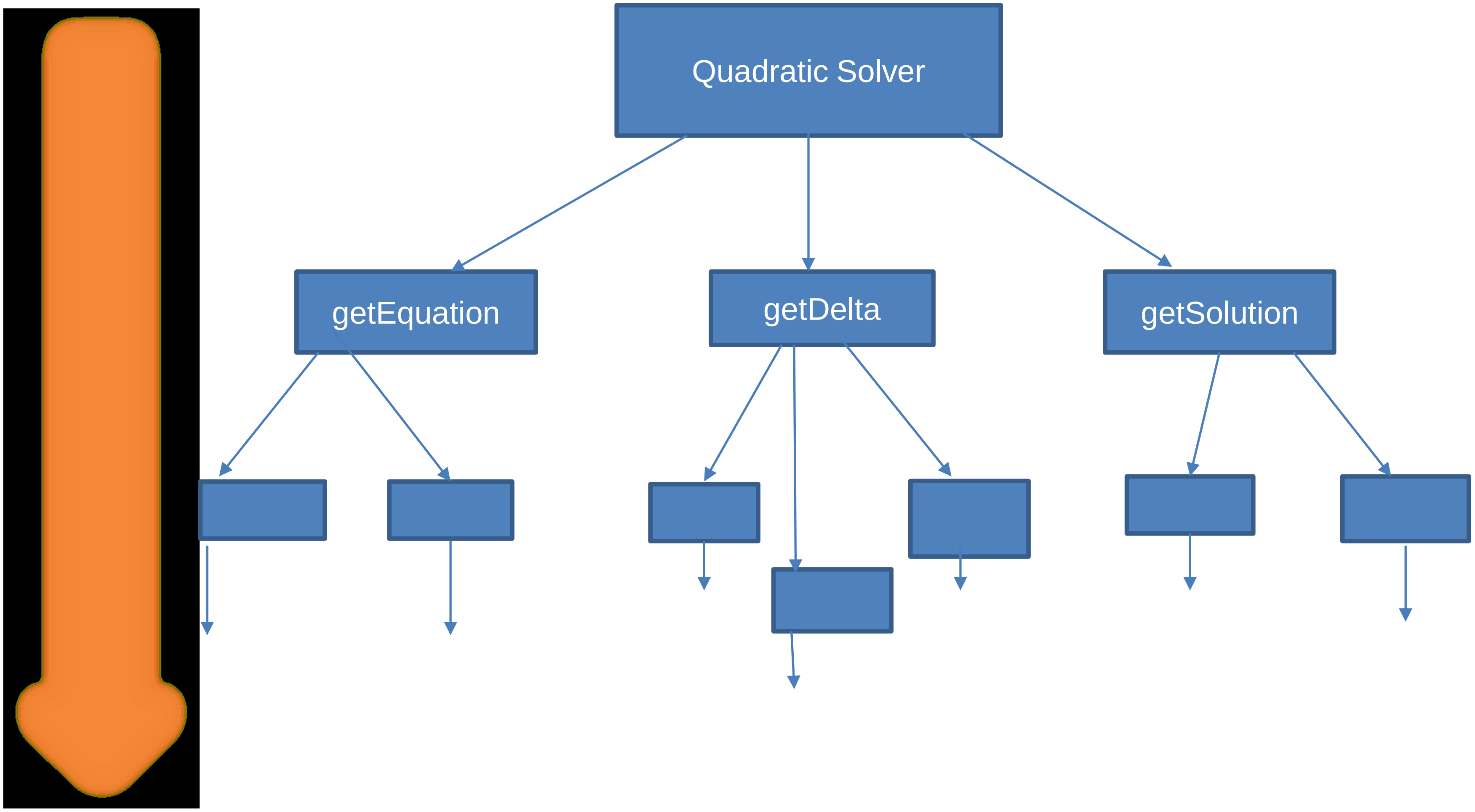
Contd…

- **Top down Concept in Problem Solving**
- If we look at a problem as a whole, it may seem impossible to solve because it is so complex. Examples:
- writing a University System program
- writing a word processor
- Complex problems can be solved using **top-down design,** also known as **stepwise refinement**, where
- We break the problem into parts
- Then break the parts into parts
- Soon, each of the parts will be easy to do

Top down Design



Top down Design



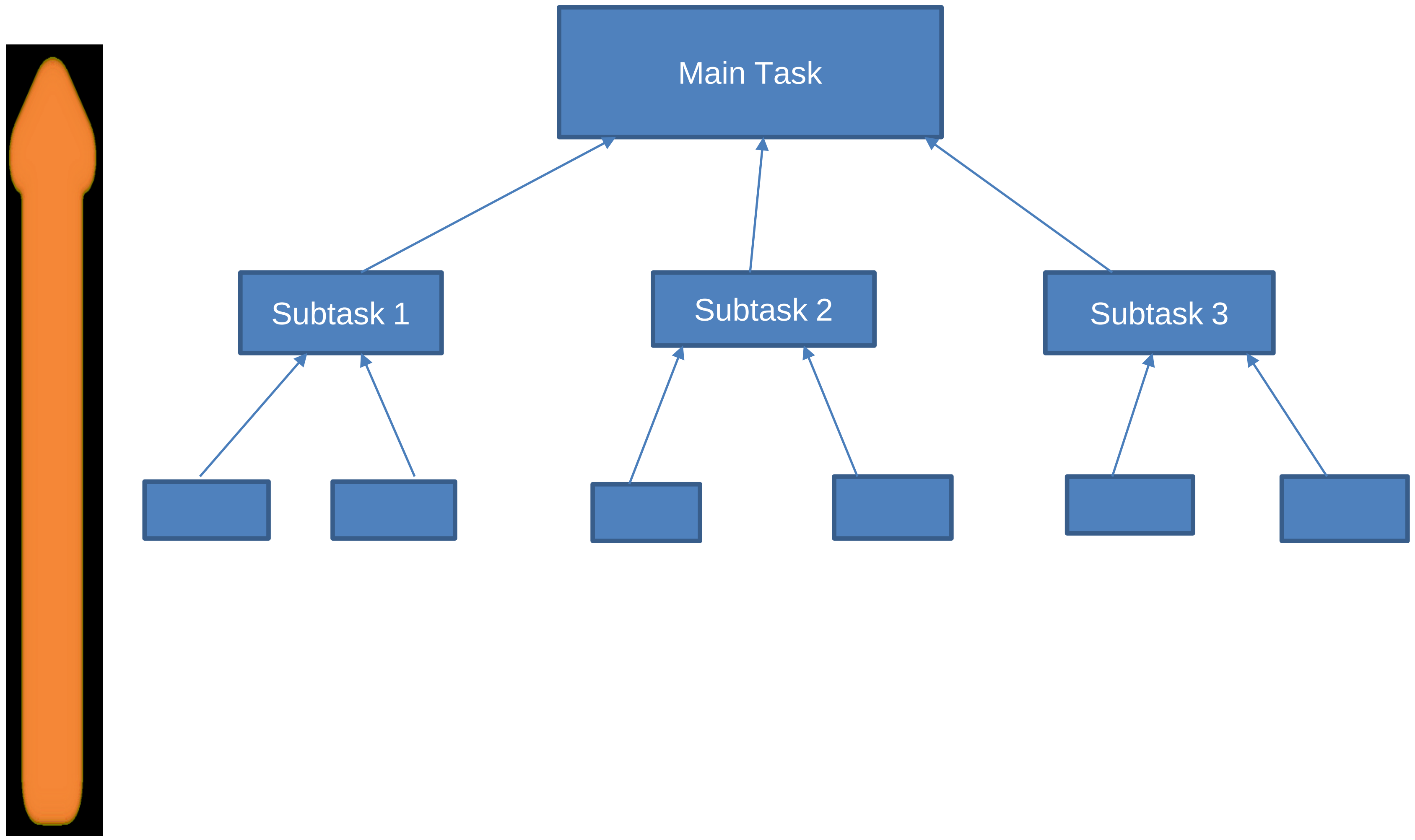
Advantages of Top-down Design

- Breaking the problem into parts helps us to clarify what needs to be done.
- At each step of refinement, the new parts become less complicated and, therefore, easier to figure out.
- Parts of the solution may turn out to be reusable.
- Breaking the problem into parts allows more than one person to work on the solution.

Bottom-up Design

- In **bottom-up** design individual parts of the system are specified in details.
- The parts are then linked together to form larger components, which are in turn linked until a complete system is formed.
- Object-oriented languages such as C++ or JAVA use bottom-up approach where each object is identified first.

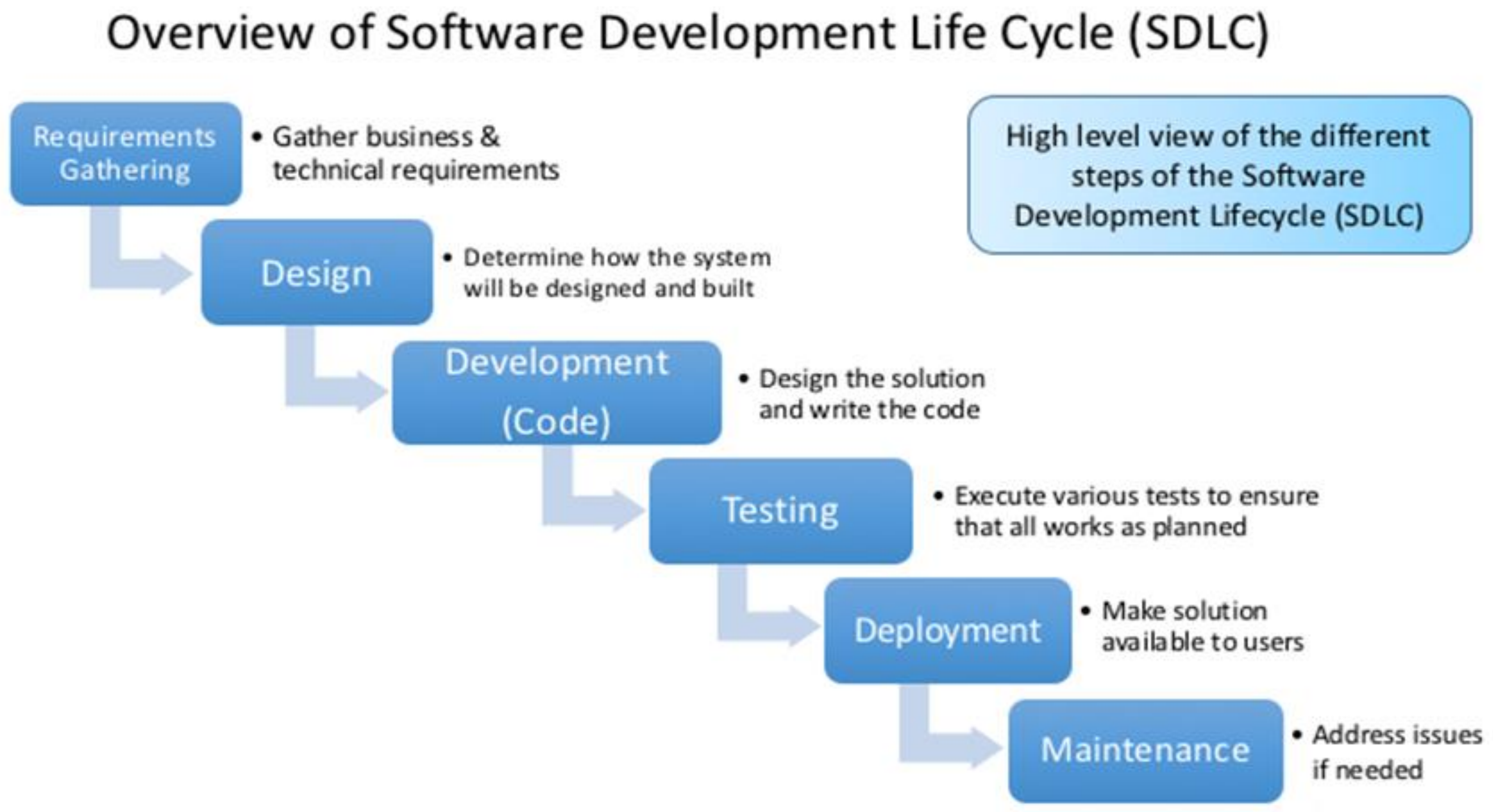
Bottom up Design



Software Development Life Cycle (SDLC)

- The software development life cycle (SDLC) is a framework defining tasks performed at each step in the software development process
- SDLC is a structure followed by a development team within the software organization
- It consists of a detailed plan describing how to develop, maintain and replace specific software
- The life cycle defines a methodology for improving the quality of software and the overall development process

Software Development Life Cycle (SDLC)


Overview of Software Development Life Cycle (SDLC)
Requirements Gathering
• Gather business & technical requirements
Design
• Determine how the system will be designed and built
Development (Code)
• Design the solution and write the code
Testing
• Execute various tests to ensure that all works as planned
Deployment
• Make solution available to users
Maintenance
• Address issues if needed
High level view of the different steps of the Software Development Lifecycle (SDLC)

Programming Paradigms

A programming *paradigm* is a pattern of problem-solving thought that underlies a particular genre of programs and languages.

There are four main programming paradigms:

- ☐ Imperative
- ☐ Object-oriented
- ☐ Functional
- ☐ Logic (declarative)

Imperative Paradigm

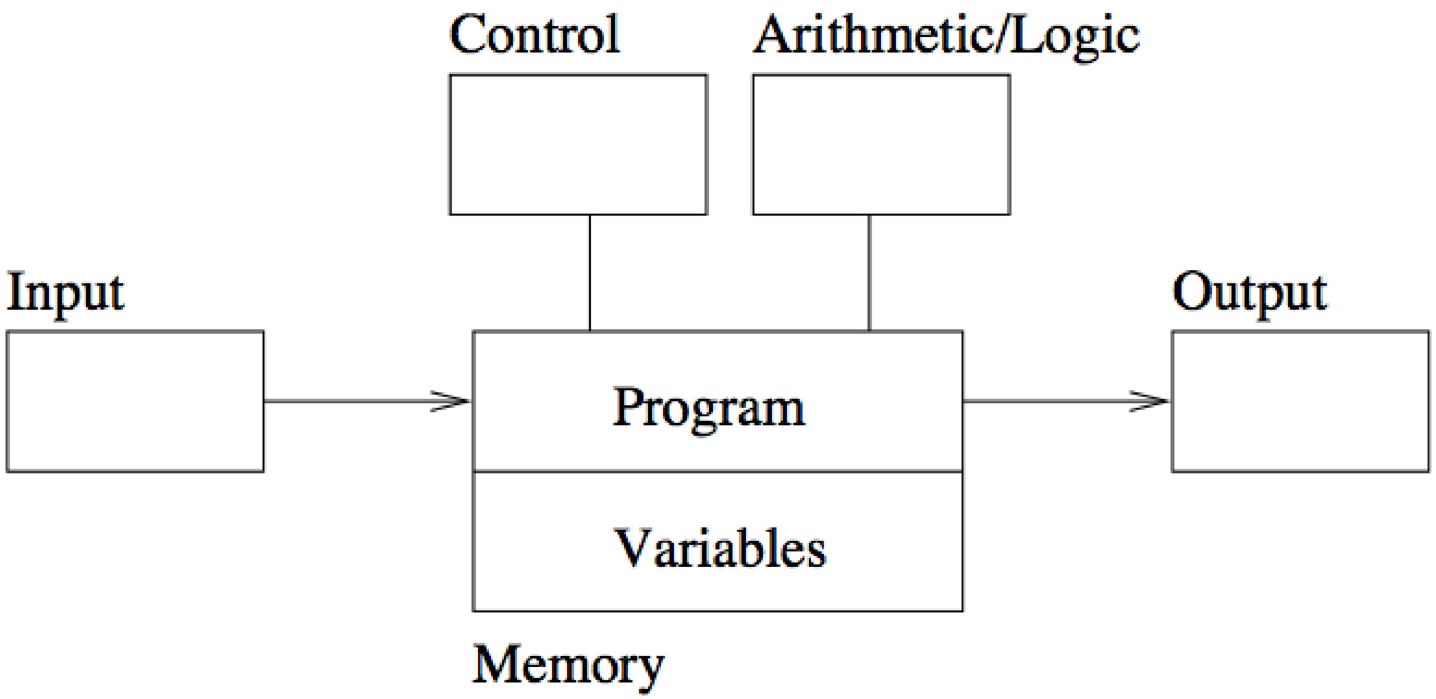
Follows the classic von Neumann-Eckert model:

- Program and data are indistinguishable in memory
- Program = a sequence of commands
- State = values of all variables when program runs
- Large programs use procedural abstraction

Example imperative languages:

- Cobol, Fortran, C, Ada, Perl, …

The von Neumann-Eckert Model



Object-oriented (OO) Paradigm

An OO Program is a collection of objects that interact by passing messages that transform the state.

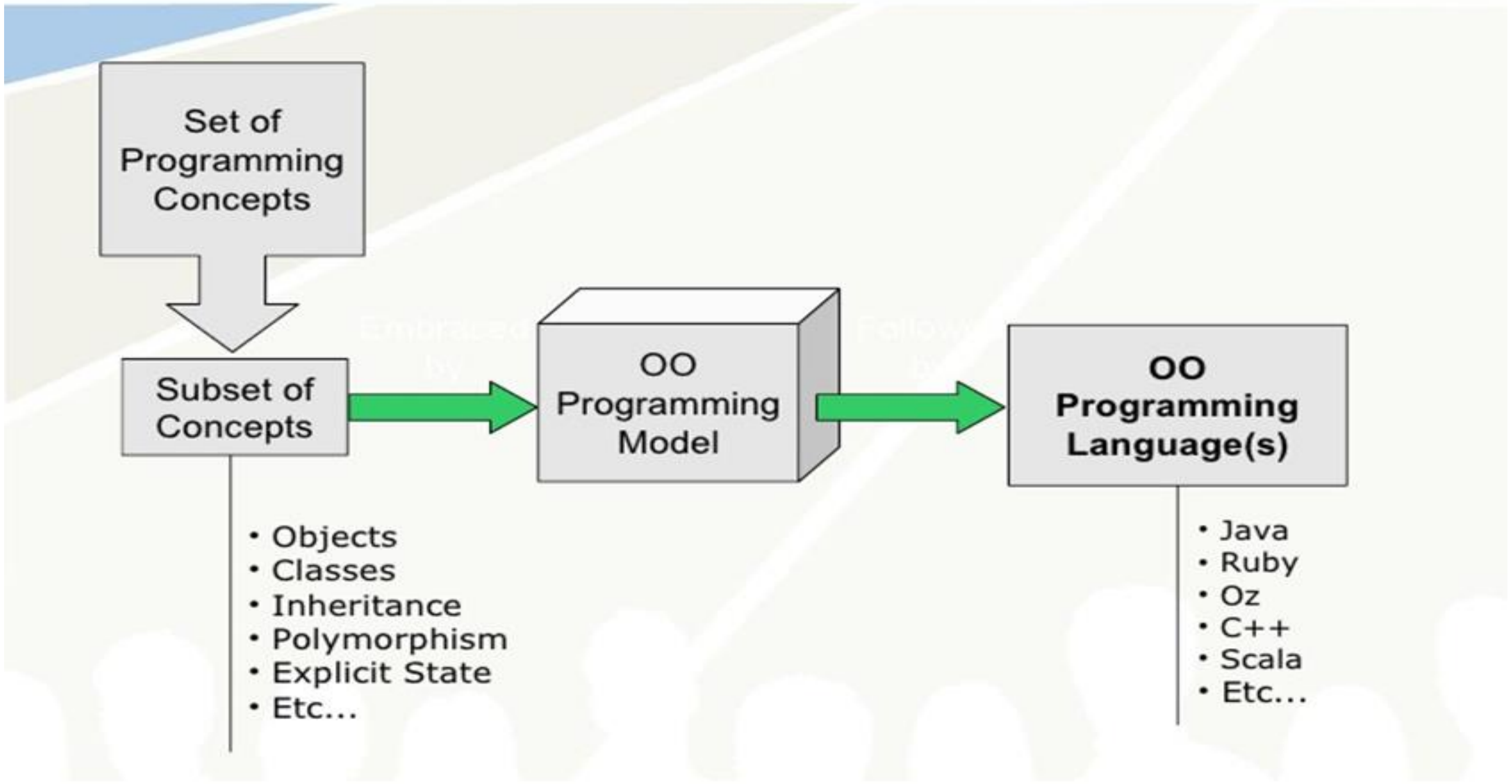
When studying OO, we learn about:

- Sending Messages
- Inheritance
- Polymorphism

Example OO languages:

Smalltalk, Java, C++, C#, and Python

Contd…


Set of Programming Concepts
Subset of Concepts
OO Programming Model
OO Programming Language(s)
• Objects
• Classes
• Inheritance
• Polymorphism
• Explicit State
• Etc…
• Java
• Ruby
• Oz
• C++
• Scala
• Etc…

Functional Paradigm

Functional programming models a computation as a collection of mathematical functions.

- Input = domain
- Output = range

Functional languages are characterized by:

- Functional composition
- Recursion

Example functional languages:

- Lisp, Scheme, ML, Haskell, …

Functional vs Imperative

- Compute Factorial Function
  - In Imperative Programming Languages:

```
fact(int n) {
  if  (n <= 1)    return 1;
  else
       return n * fact(n-1);
}
```

- In Functional Programming Languages: Scheme

```
(define (fact n)
  (if (< n 1) 1 (* n (fact (- n 1)))
))
```

Logic Paradigm

Logic programming declares what outcome the program should accomplish, rather than how it should be accomplished.

When studying logic programming we see:

- Programs as sets of constraints on a problem
- Programs that achieve all possible solutions
- Programs that are nondeterministic

Example logic programming languages:

- Prolog

Role of Programming Languages

- A **programming language** is formal language that specifies a set of instructions that can be used to produce various kinds of output .
- Programming languages generally consist of instructions for a computer .
- Programming languages can be used to create programs that implement specific algorithms.

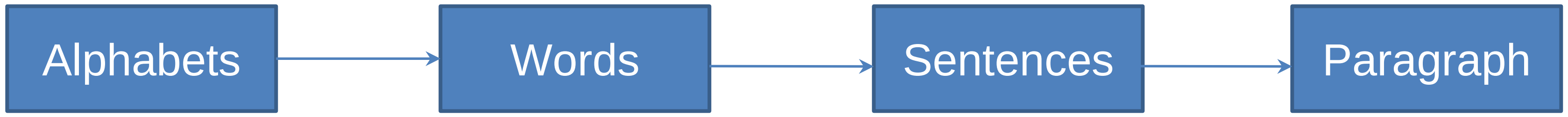
What is Programming Language

- A programming language is a notational system for describing computation in machine-readable and human-readable form.
- A programming language is a tool for developing executable models for a class of problem domains.

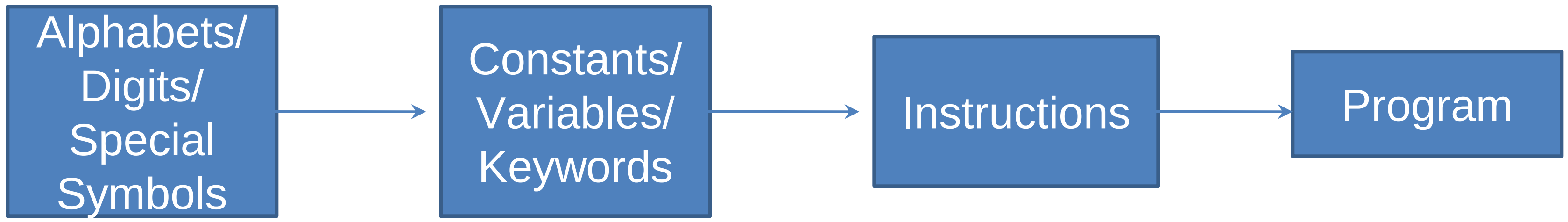
What is Programming Language

- English is a natural language. It has words, symbols and grammatical rules.
- A programming language also has words, symbols and rules of grammar.
- The grammatical rules are called syntax.
- Each programming language has a different set of syntax rules.

Steps in learning English Language-



Steps in learning C-Language-


Alphabets/ Digits/ Special Symbols
Constants/ Variables/ Keywords
Instructions
Program

Need to Study Programming Language

- To improve your ability to develop effective algorithms.
- To improve your use of existing programming languages.
- To increase your vocabulary of useful programming constructs.
- To allow a better choice of programming language.
- To make it easier to learn a new language.
- To make it easier to design a new language.

Characteristics of Programming Languages

- The language must allow the programmer to write simple, clear and concise programs.
- It must be simple to use so that a programmer can learn it without any explicit training.
- It must be platform independent. That is, the program developed using the programming language can run on any computer system.
- The Graphical User Interface (GUI) of the language must be attractive, user-friendly, and self-explanatory.
- The function library used in the language should be well documented so that the necessary information about a function can be obtained while developing application.

Characteristics of Programming Languages (Cont…)

- Several programming constructs supported by the language must match well with the application area it is being used for.
- The programs developed in the language must make efficient use of memory as well as other computer resources.
- The language must provide necessary tools for development, testing, debugging, and maintenance of a program. All these tools must be incorporated into a single environment known as Integrated Development Environment (IDE), which enables the programmer to use them easily.
- The language must be consistent in terms of both syntax and semantics.