# CSIT113

# Computer and Information Technology

L-T-P 2-0-2

MODULE 3

## Outline of Topics

- Hardware/Software interface
  - Layers of the Machine
  - Kinds of Software
- Computer Languages
- Syntax, Semantics, Grammars
- What happens to your program?
  - Compilation, Linking, Execution
  - Program errors
- Compilation vs. Interpretation etc.

## A Layered View of the Computer

**Application Programs**

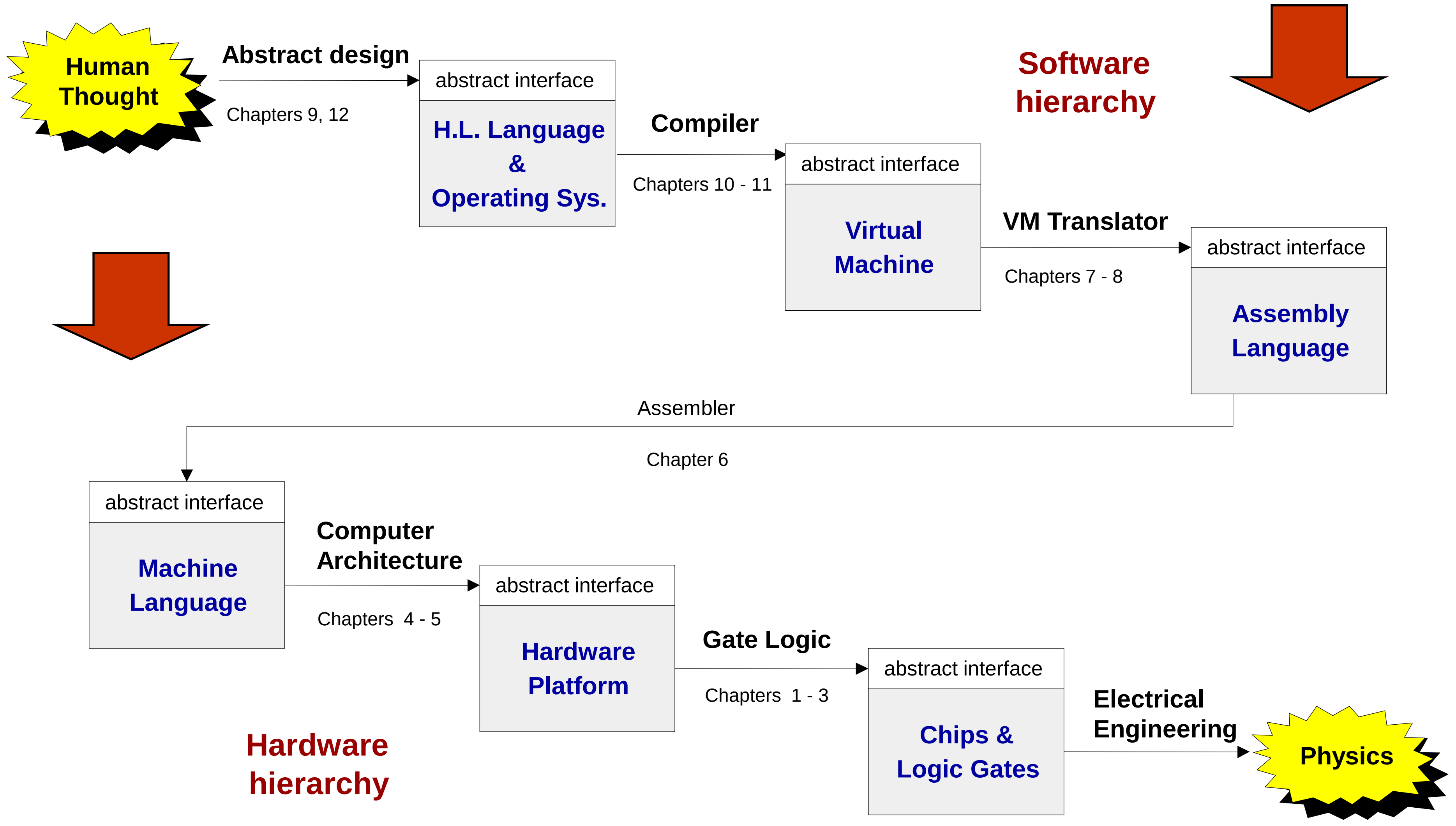
Word-Processors, Spreadsheets,
Database Software, IDEs,
etc…

**System Software**

Compilers, Interpreters,Preprocessors, etc.
Operating System, Device Drivers

**Machine with all its hardware**

# The Flow


Human Thought
Abstract design
Chapters 9, 12
abstract interface
H.L. Language & Operating Sys.
Compiler
Chapters 10 - 11
abstract interface
Virtual Machine
Software hierarchy
VM Translator
Chapters 7 - 8
abstract interface
Assembly Language
Assembler
Chapter 6
abstract interface
Machine Language
Computer Architecture
Chapters 4 - 5
abstract interface
Hardware Platform
Gate Logic
Chapters 1 - 3
abstract interface
Chips & Logic Gates
Electrical Engineering
Physics
Hardware hierarchy

## Programs

- Programs are written in programming languages
    - PL = programming language
    - Pieces of the same program can be written in different PLs
        - Languages closer to the machine can be more efficient
        - As long as they agree on how to communicate
- A  PL is
    - A special purpose and limited language
    - A set of rules and symbols used to construct a computer program
    - A language used to interact with the computer

# Computer Languages

- Machine Language

  Uses binary code

  Machine-dependent

  Not portable
- Assembly Language
  - Uses mnemonics
  - Machine-dependent
  - Not usually portable
- High-Level Language (HLL)
  - Uses English-like language
  - Machine independent
  - Portable (but must be compiled for different platforms)
  - Examples: Pascal, C, C++, Java, Fortran, . . .

# Machine Language

- The representation of a computer program which is actually read and understood by the computer.
  - A program in machine code consists of a sequence of machine instructions.
- Instructions:
  - Machine instructions are in binary code
  - Instructions specify operations and memory cells involved in the operation

**Example:**

| Operation | Address |
|---|---|
| 0010 | 0000 0000 0100 |
| 0100 | 0000 0000 0101 |
| 0011 | 0000 0000 0110 |

## Machine language

Abstraction – implementation duality:

- Machine language ( = instruction set) can be viewed as a programmer-oriented abstraction of the hardware platform
- The hardware platform can be viewed as a physical means for realizing the machine language abstraction

Another duality:

- Binary version: 0001 0001 0010 0011 (machine code)
- Symbolic version ADD R1, R2, R3 (assembly)

# Example of machine-language

Here's what a program-fragment looks like:

```
10100001 10111100 10010011 00000100
00001000 00000011 00000101 11000000
10010011 00000100 00001000 10100011
11000000 10010100 00000100 00001000
```

It means: z = x + y;

## Incomprehensible?

- Though possible, it is extremely difficult, tedious (and error-prone) for humans to read and write "raw" machine-language
- When unavoidable, a special notation can help (called hexadecimal representation):

```
A1 BC 93 04 08
03 05 C0 93 04 08
A3 C0 94 04 08
```

- But still this looks rather meaningless!

## Assembly Language

- A symbolic representation of the machine language of a specific processor.
- Is converted to machine code by an assembler.
- Usually, each line of assembly code produces one machine instruction (One-to-one correspondence).
- Programming in assembly language is slow and error-prone but is more efficient in terms of hardware performance.
- Mnemonic representation of the instructions and data
- **Example:**

```
Load    Price
Add     Tax
Store   Cost
```

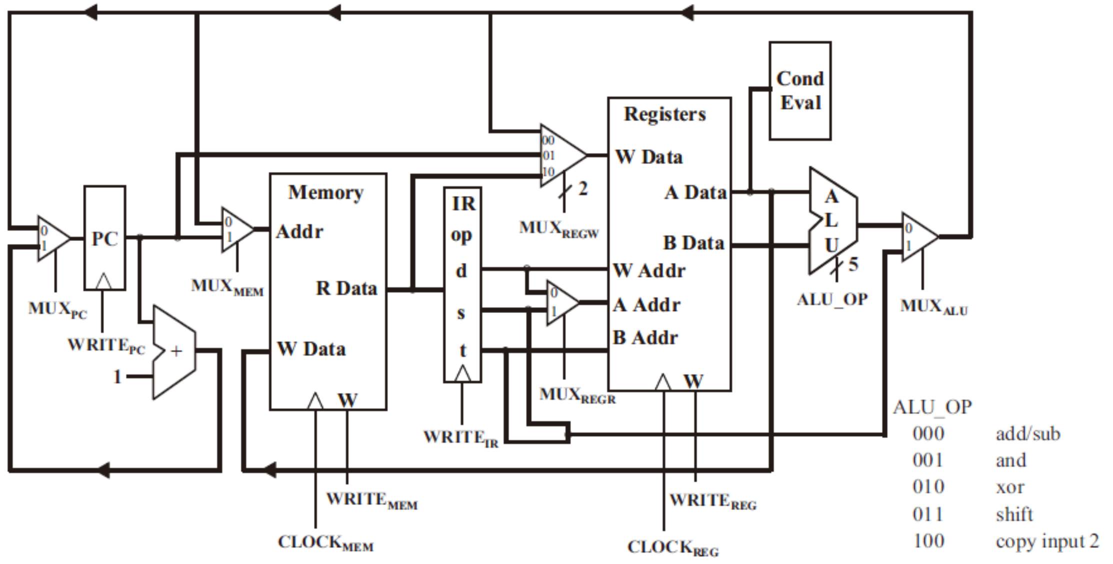
# Assembly language

Abstraction – implementation duality:

- Assembly language( = instruction set) can be viewed as a programmer-oriented abstraction of the hardware platform

- The hardware platform can be viewed as a physical means for realizing the machine language abstraction

| # | Operation | Fmt | Pseudocode |
|---|---|---|---|
| 0: | halt | 1 | `exit(0)` |
| 1: | add | 1 | R[d] ← R[s] + R[t] |
| 2: | subtract | 1 | R[d] ← R[s] - R[t] |
| 3: | and | 1 | R[d] ← R[s] & R[t] |
| 4: | xor | 1 | R[d] ← R[s] ^ R[t] |
| 5: | shift left | 1 | R[d] ← R[s] << R[t] |
| 6: | shift right | 1 | R[d] ← R[s] >> R[t] |
| 7: | load addr | 2 | R[d] ← addr |
| 8: | load | 2 | R[d] ← mem[addr] |
| 9: | store | 2 | mem[addr] ← R[d] |
| A: | load indirect | 1 | R[d] ← mem[R[t]] |
| B: | store indirect | 1 | mem[R[t]] ← R[d] |
| C: | branch zero | 2 | if (R[d] == 0) pc ← addr |
| D: | branch positive | 2 | if (R[d] > 0) pc ← addr |
| E: | jump register | 1 | pc ← R[t] |
| F: | jump and link | 2 | R[d] ← pc; pc ← addr |



# Typical machine language commands (a small sample)

```
// In what follows R1,R2,R3 are registers, PC is program counter,
// and addr is some value.

ADD R1,R2,R3      // R1 ← R2 + R3

ADDI R1,R2,addr   // R1 ← R2 + addr

AND R1,R1,R2      // R1 ← R1 and R2 (bit-wise)

JMP addr          // PC ← addr

JEQ R1,R2,addr    // IF R1 == R2 THEN PC ← addr ELSE PC++

LOAD R1, addr     // R1 ← RAM[addr]

STORE R1, addr    // RAM[addr] ← R1

NOP               // Do nothing

// Etc. – some 50-300 command variants
```

## High-level language

- A programming language which use statements consisting of English-like keywords such as "FOR", "PRINT" or “IF“, ... etc.
- Each statement corresponds to several machine language instructions (one-to-many correspondence).
- These languages deliberately “hide” from a programmer many details concerning HOW his problem actually will be solved by the underlying computing machinery
- Much easier to program than in assembly language.
- Data  are referenced using descriptive names
- Operations can be described using familiar symbols
- Example:

```
Cost := Price + Tax
```

## The example in BASIC

```
1    LET X = 4
2    LET Y = 5
3    LET Z = X + Y
4    PRINT  X, “+”, Y, “=“, Z
5    END
```

```
Output:    4 + 5 = 9
```

## Same example: rewritten in C

```
#include <stdio.h> // needed for printf()

int     x = 4, y = 5;  // initialized variables
int     z;                      // unitialized variable

int main()
{
        z = x + y;
        printf( “%d + %d = %d \n”, x, y, z );
}
```

# Syntax & Semantics

- Syntax:
  - The structure of strings in some language. A language's syntax is described by a grammar.
  - Examples:
    - Binary number

```
<binary_number>          = <bit> | <bit>  <binary_number>
<bit>                    = 0 | 1
```

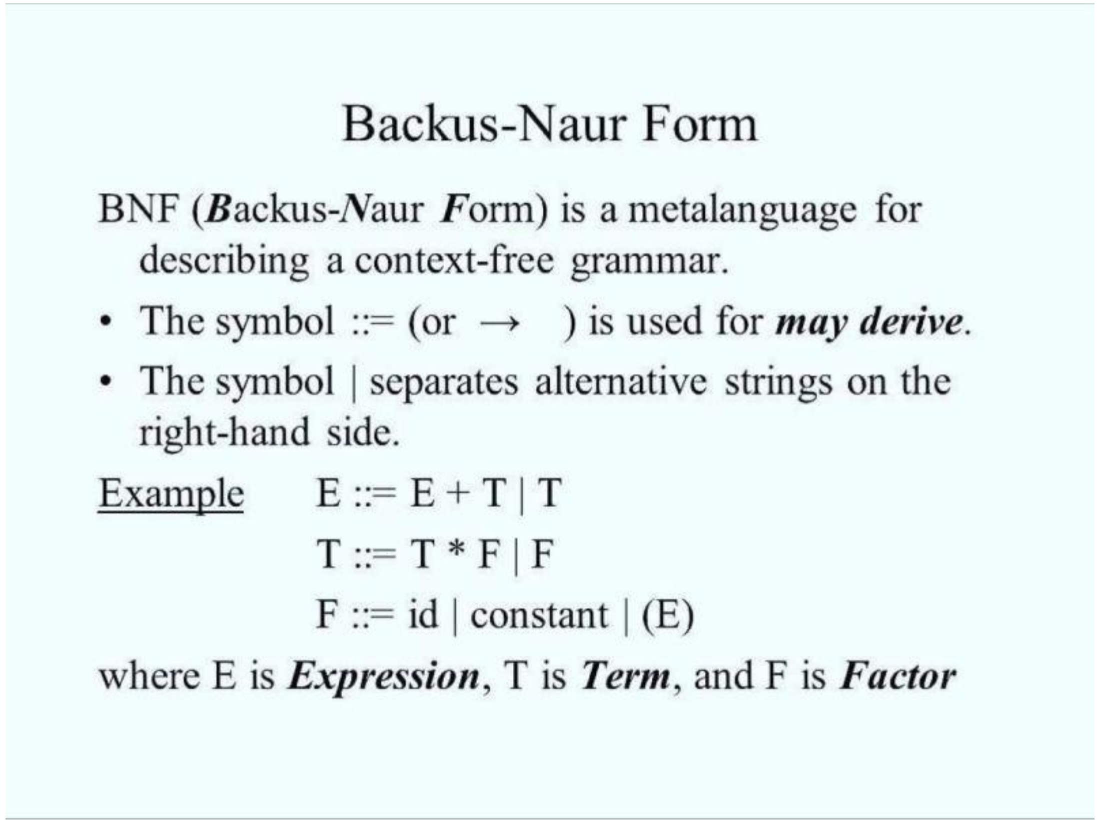
    - Identifier

```
<identifier> = <letter> {<letter> | <digit> }
<letter>                 = a | b | . . . | z
<digit                   = 0 | 1 | . . . | 9
```

- Semantics:
  - The meaning of the language

## Syntax & Grammars

- Syntax descriptions for a PL are themselves written in a formal language.
  - E.g. Backus-Naur Form (BNF)
- The formal language is not a PL but it can be implemented by a compiler to enforce grammar restrictions.
- Some PLs look more like grammar descriptions than like instructions.



Backus-Naur Form

BNF (***B****ackus-****N****aur* ***F****orm*) is a metalanguage for describing a context-free grammar.

- The symbol ::= (or $\rightarrow$ ) is used for ***may derive***.
- The symbol | separates alternative strings on the right-hand side.

Example E ::= E + T | T

T ::= T * F | F

F ::= id | constant | (E)

where E is ***Expression***, T is ***Term***, and F is ***Factor***

# Source Program

- **Source program**
  - The form in which a computer program is written in some formal programming language by the  programmer is called source.
  - It can be compiled automatically into <u>object code</u> or <u>machine code</u> or executed by an interpreter.
  - E.g. Pascal source programs have extension ‘.pas’ (source program)

## Object & Executable Programs

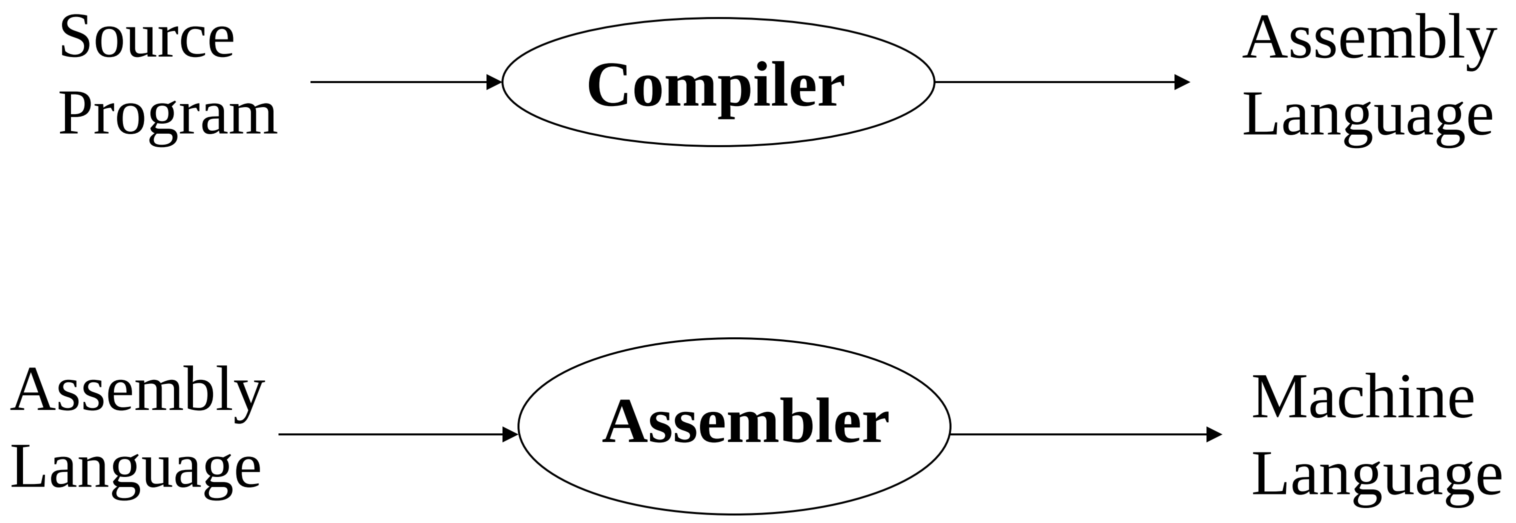
- **Object program**
  - Output from the compiler
  - Equivalent machine language translation of the source program
  - Files usually have extension '.obj'

- **Executable program**
  - Output from linker/loader
  - Machine language program linked with necessary libraries & other files
  - Files usually have extension '.exe'

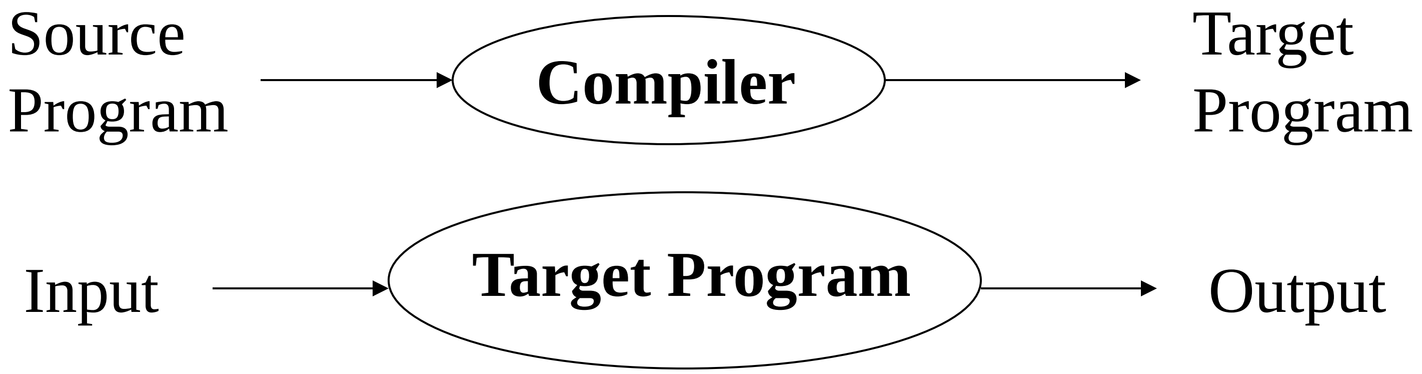
## Compilers

- **Compiler**
  - A program that converts another program from some source language (or high-level programming language / HLL) to machine language (object code).
  - Some compilers output assembly language which is then converted to machine language by a separate assembler.
  - Is distinguished from an assembler by the fact that each input statement, in general, correspond to more than one machine instruction.

# Compilation into Assembly Language

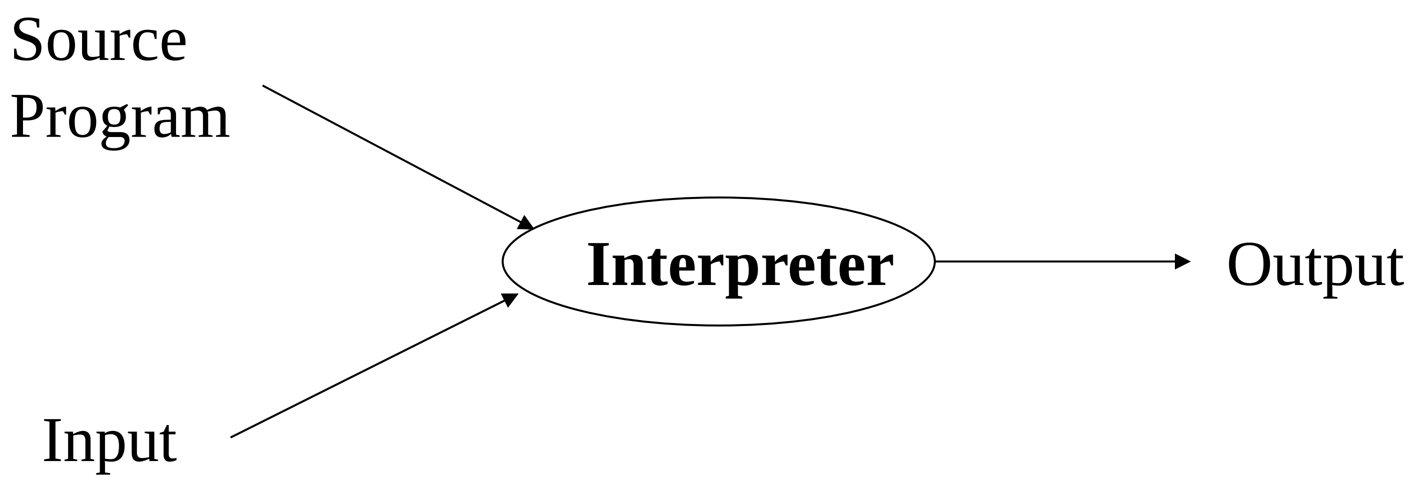


## Compilation



- Compiler translates source into target (directly a machine language program)
- Compiler goes away at execution time
- Compiler is itself a machine language program, presumably created by compiling some other high-level program
- Machine language, when written in a format understood by the OS is **object code**

# Interpretation



- The interpreter stays around during execution
- It reads and executes statements one at a time

# Compilation vs. Interpretation

- Compilation:
  - Syntax errors caught before running the program
  - Better performance
  - Decisions made once, at compile time
- Interpretation:
  - Better diagnostics (error messages)
  - More flexibility
  - Supports **late binding** (delaying decisions about program implementation until runtime)
    - Can better cope with PLs where type and size of variables depend on input
  - Supports creation/modification of program code on the fly (e.g. Lisp, Prolog)

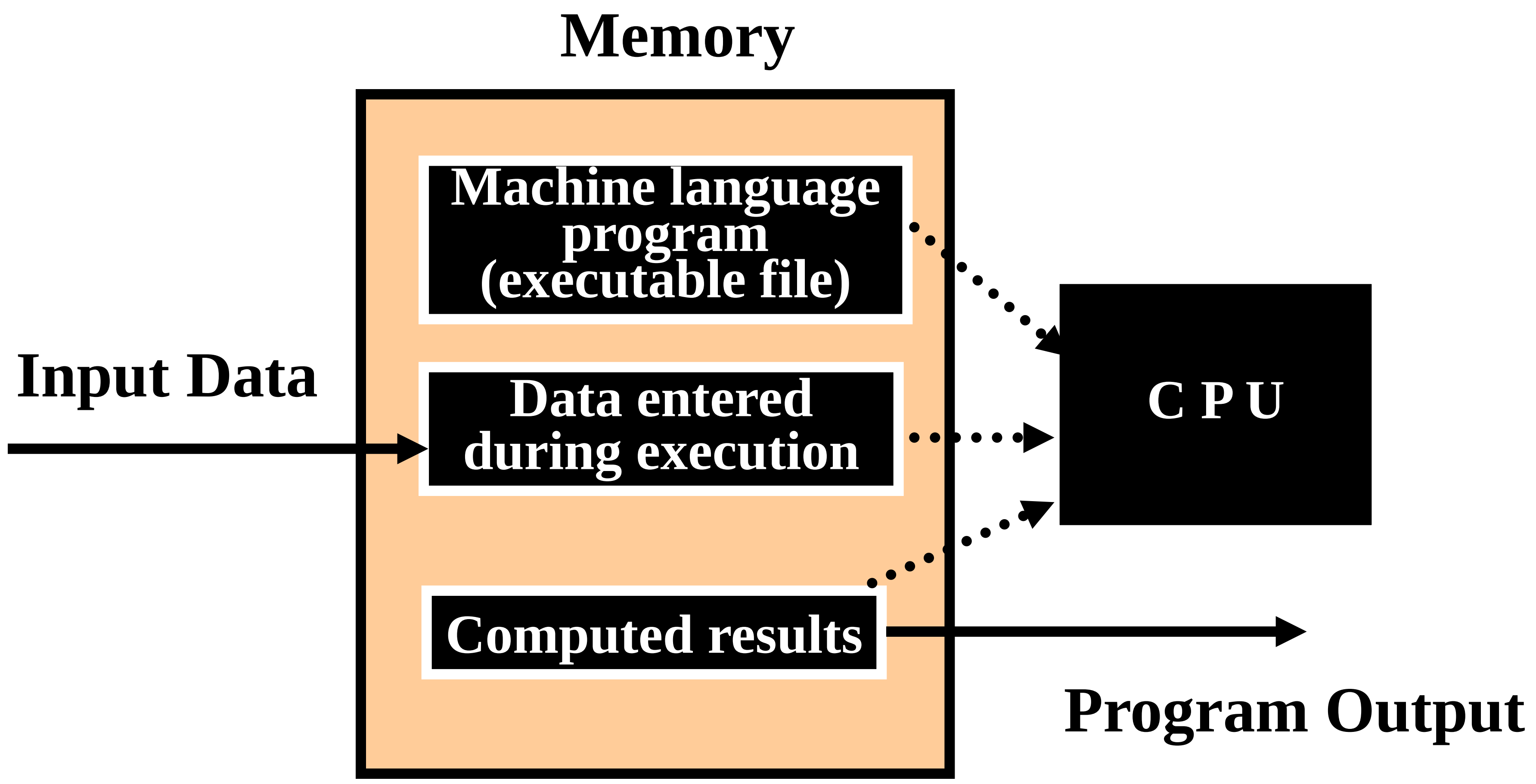
# What is a Linker?

- A program that pulls other programs together so that they can run.
- Most programs are very large and consist of several <u>modules</u>.
- Even small programs use existing code provided by the programming environment called <u>libraries</u>.
- The linker pulls everything together, makes sure that references to other parts of the program (code) are resolved.

## Program Execution

- Steps taken by the CPU to run a program (instructions are in machine language):
    1. Fetch an instruction
    2. Decode (interpret) the instruction
    3. Retrieve data, if needed
    4. Execute (perform) actual processing
    5. Store the results, if needed

# Running Programs


Memory
Machine language program (executable file)
Input Data
Data entered during execution
C P U
Computed results
Program Output

## Program Errors

- Syntax Errors:
    - Errors in grammar of the language
- Runtime error:
    - When there are no syntax errors, but the program can’t complete execution
        - Divide by zero
        - Invalid input data
- Logical errors:
    - The program completes execution, but delivers incorrect results
    - Incorrect usage of parentheses

## What is Problem Solving?

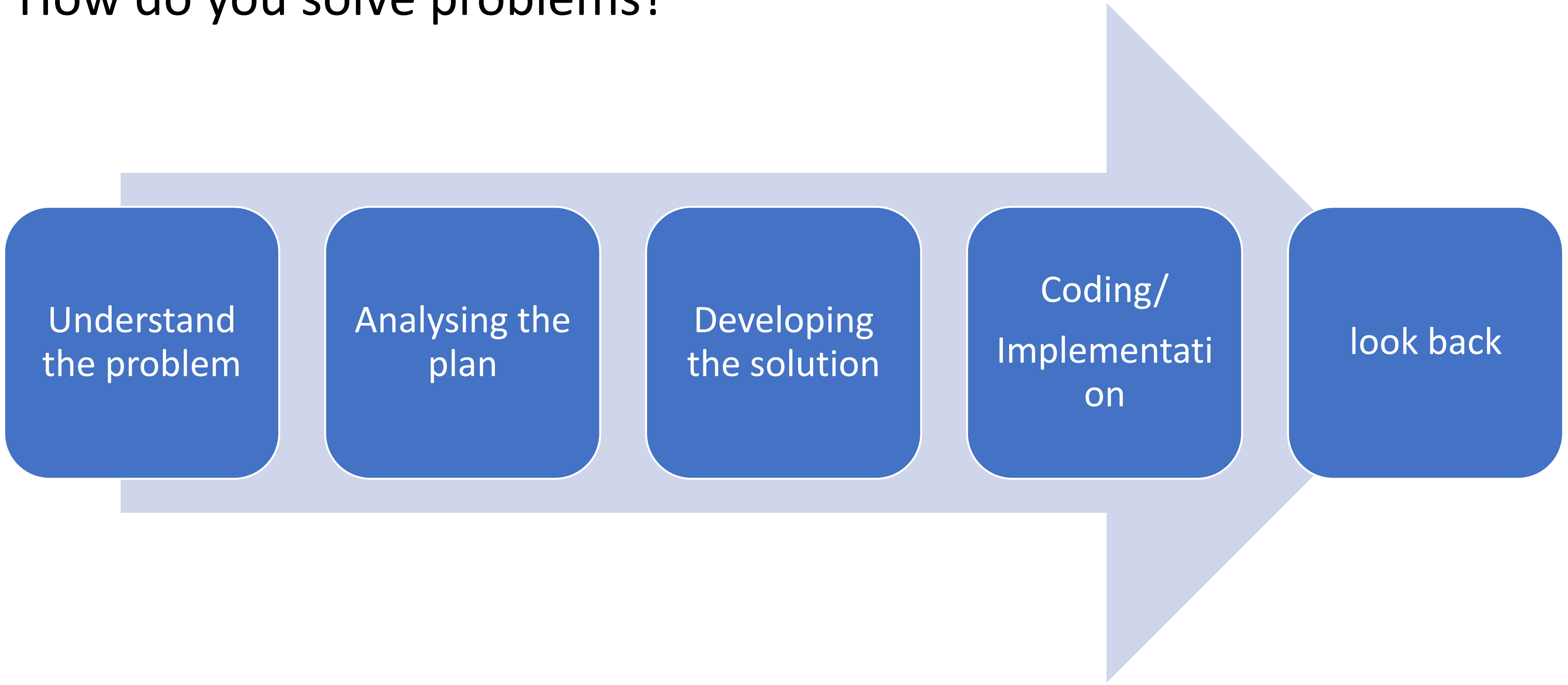
- Problem solving
  - The process of taking the statement of a problem and developing a computer program that solves that problem
- A solution consists of:
  - Algorithms
    - Algorithm: a step-by-step specification of a method to solve a problem within a finite amount of time
  - Ways to store data

## Problem Solving

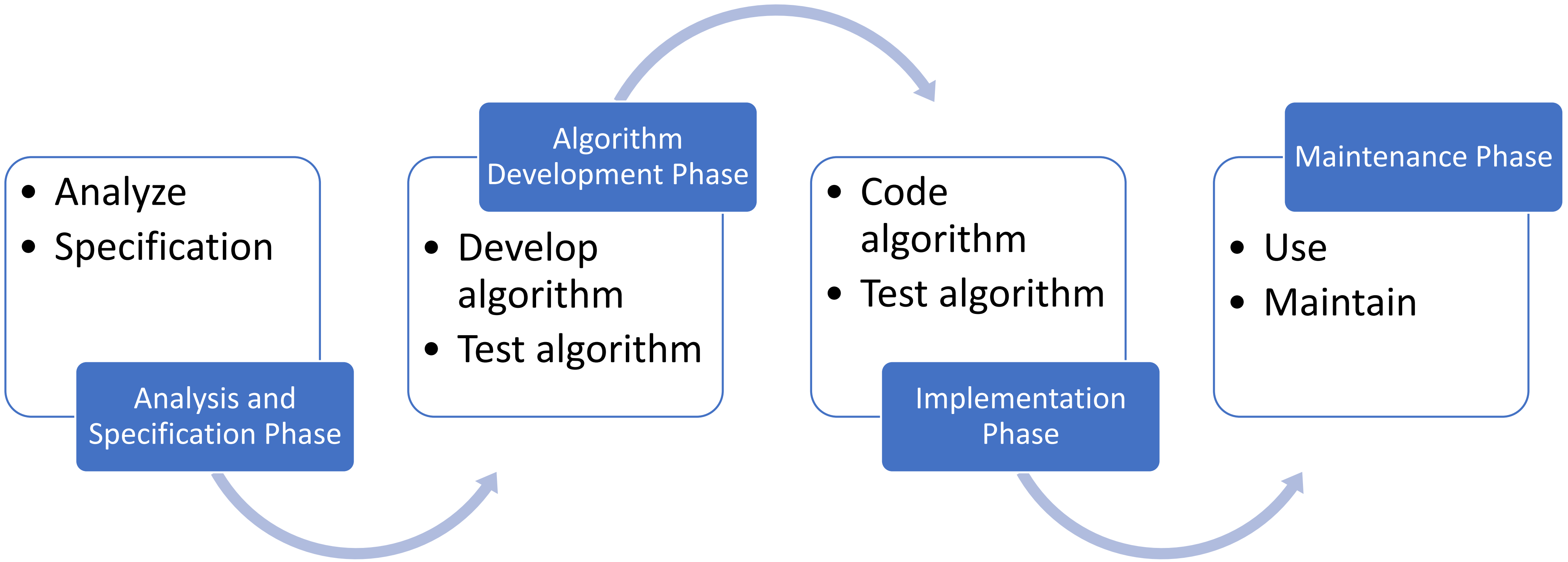
- A team of programmers is needed for a large software development project
- Teamwork requires:
  - An overall plan
  - Organization
  - Communication
- Software engineering
  - Provides techniques to facilitate the development of computer programs
- Coding without a solution design increases debugging time

## Problem Solving

How do you solve problems?



# Computer Problem-Solving



## Strategies

**Ask questions!**

- What input data/information is available ?
- What does it represent ?
- What format is it in ?
- Is anything missing ?
- Do I have everything that I need ?
- What output information am I trying to produce ?
- What do I want the result to look like … text, a picture, a graph … ?
- What am I going to have to compute ?

## Strategies

**Divide and Conquer!**

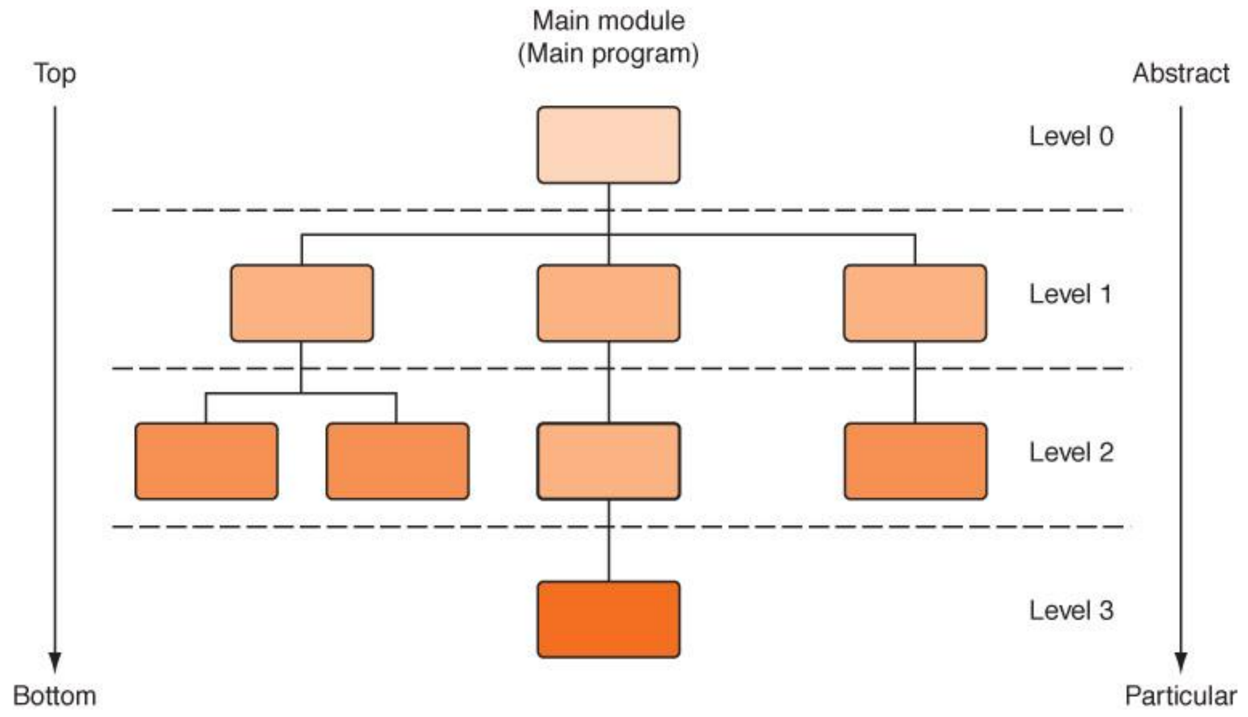
Break up a large problem into smaller units and solve each smaller problem

- Applies the concept of abstraction
- The divide-and-conquer approach can be applied over and over again until each subtask is manageable

## Table 1-1: Commonly Used Problem-Solving Strategies

| Method | Description | Example |
|---|---|---|
| **Trial and error** | Continue trying different solutions until problem is solved | Restarting phone, turning off WiFi, turning off Bluetooth in order to determine why your phone is malfunctioning |
| **Algorithm** | Step-by-step problem-solving formula | Instruction manual for installing new software on your computer |
| **Heuristic** | General problem-solving framework | Working backwards; breaking a task into steps |
| **Means-ends analysis** | Analysing a problem at series of smaller steps to move closer to the goal | Envisioning the ultimate goal and determining the best strategy for attaining it in the current situation |

# Top-Down Design

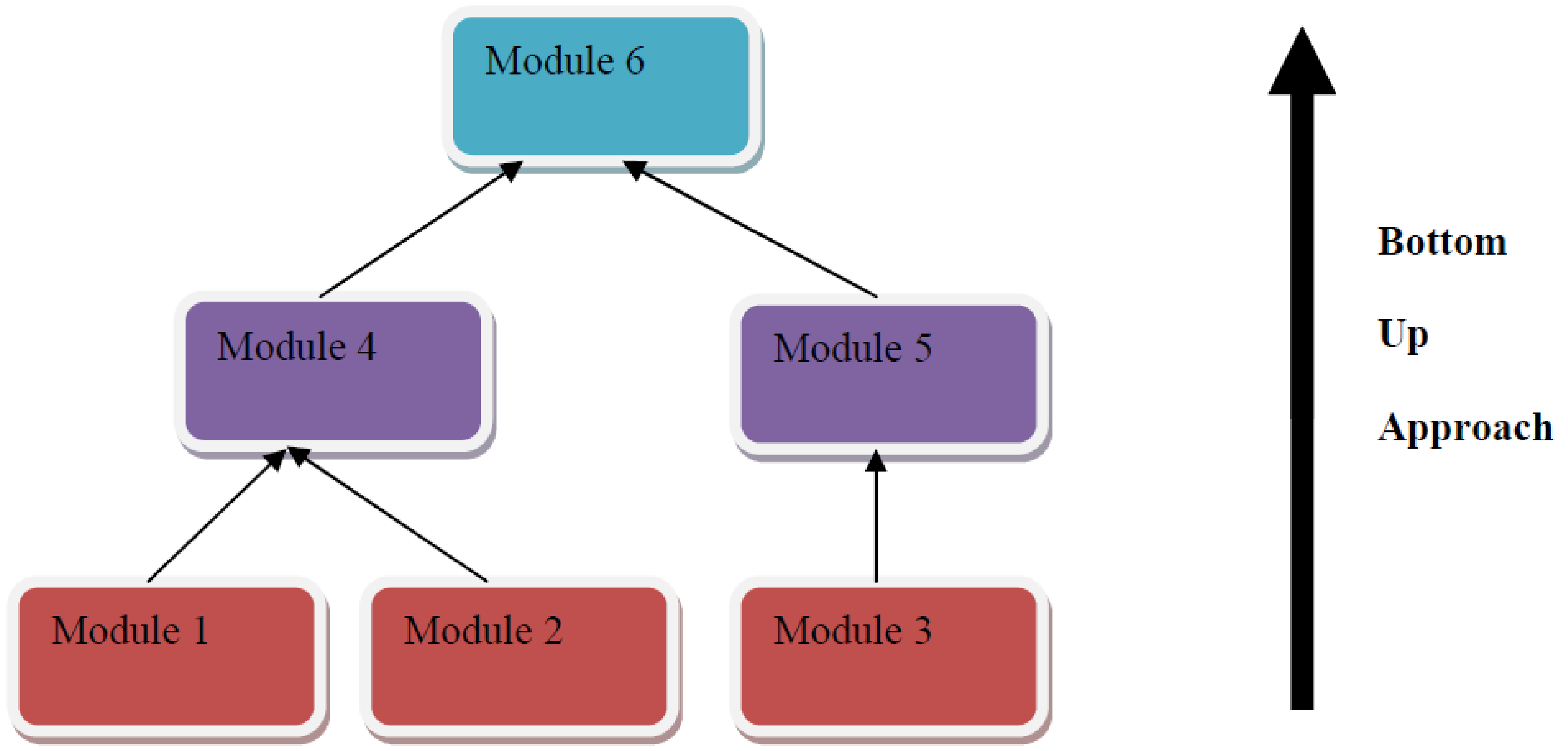


Process continues for as many levels as it takes to make every step concrete

Name of (sub)problem at one level becomes a module at next lower level

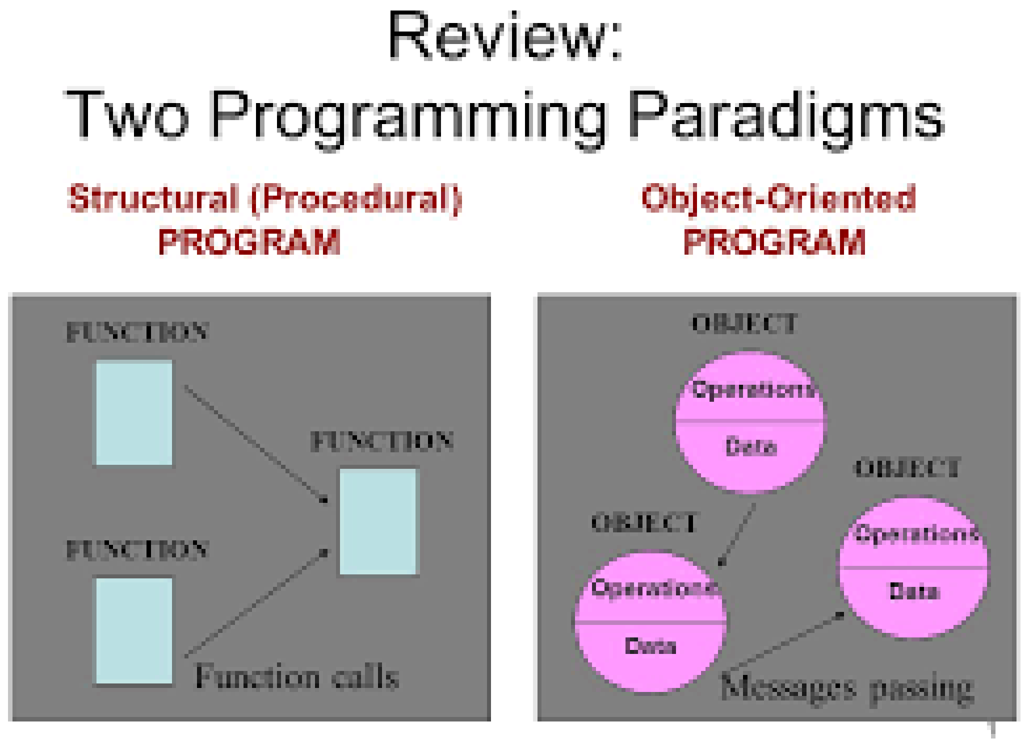
## Bottom Up approach

In bottom-up approach, system design starts with the lowest level of components, which are then interconnected to get higher level components. This process continues till a hierarchy of all system components is generated. However, in real-life scenario it is very difficult to know all lowest level components at the beginning.

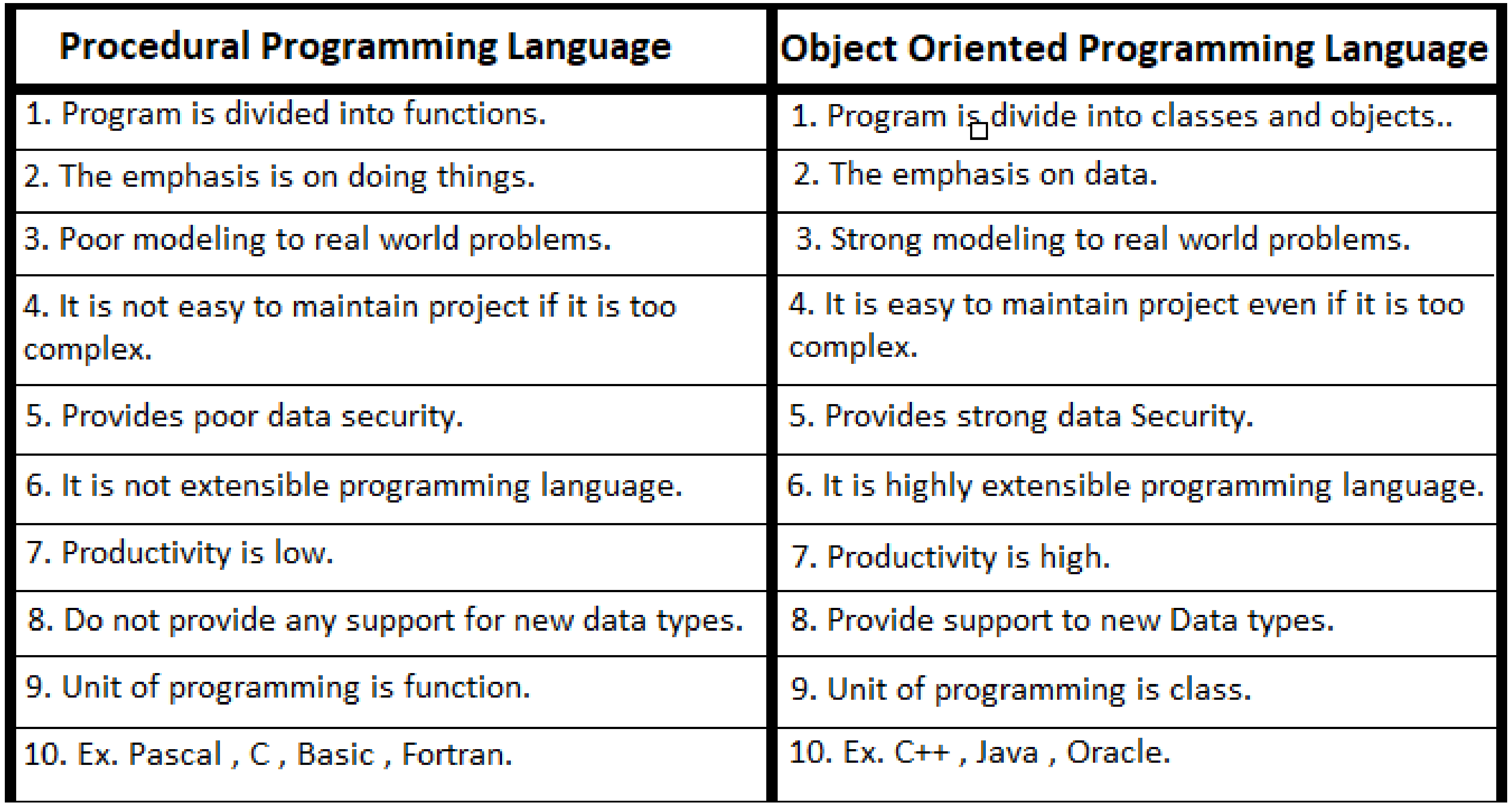


# Procedural versus Object-Oriented Programming

- Procedural programming focuses on the process/actions that occur in a program. The program starts at the beginning, does something, and ends.
- Object-Oriented programming is based on the data and the functions that operate on it. Objects are instances of abstract data types that represent the data and its functions



# Comparison



| Procedural Programming Language | Object Oriented Programming Language |
|---|---|
| 1. Program is divided into functions. | 1. Program is divide into classes and objects.. |
| 2. The emphasis is on doing things. | 2. The emphasis on data. |
| 3. Poor modeling to real world problems. | 3. Strong modeling to real world problems. |
| 4. It is not easy to maintain project if it is too complex. | 4. It is easy to maintain project even if it is too complex. |
| 5. Provides poor data security. | 5. Provides strong data Security. |
| 6. It is not extensible programming language. | 6. It is highly extensible programming language. |
| 7. Productivity is low. | 7. Productivity is high. |
| 8. Do not provide any support for new data types. | 8. Provide support to new Data types. |
| 9. Unit of programming is function. | 9. Unit of programming is class. |
| 10. Ex. Pascal , C , Basic , Fortran. | 10. Ex. C++ , Java , Oracle. |

## Algorithms

It is a step by step description of how to arrive at a solution to a given problem.

The characteristics of an Algorithm are as follow:

- Each instruction should be precise and unambiguous.
- No instruction should repeat infinitely.
- It should give the correct result at the end.

# Algorithms

Advantages of an algorithm

1. It is a step-by-step representation of a solution to a given problem ,which is very easy to understand
2. It has got a definite procedure.
3. It is independent of programming language.
4. It is easy to debug as every step got its own logical sequence.

Disadvantages of an algorithm

1. It is time consuming process.
2. It is difficult to show branch and looping in the algorithm.

# Algorithm with Selection

Determine Dress

```
IF (temperature > 90)
        Write “Texas weather: wear shorts”
ELSE IF (temperature > 70)
        Write “Ideal weather: short sleeves are fine”
ELSE IF (temperature > 50)
        Write “A little chilly: wear a light jacket”
ELSE IF (temperature > 32)
        Write “Philadelphia weather: wear a heavy coat”
ELSE
        Write “Stay inside”
```

Example  : Write an algorithm to check whether he is eligible to vote? (More than or equal to 18 years old).

```
Step 1: Start
Step 2: Take age of the user and store it in age
Step 3: Check age value, if age >= 18 then go to step 4 else step 5
Step 4: Print “Eligible to vote” and go to step 6
Step 5: Print “Not eligible to vote”
Step 6: Stop
```

Example : Write an algorithm to print all natural numbers up to n‟.

```
Step 1: Start
Step 2: Take any number and store it in n.
Step 3: Store 1 in I
Step 4: Check I value, if I<=n then go to step 5 else go to step 8
Step 5: Print I
Step 6: Increment I value by 1
Step 5: Go to step 4
Step 8: Stop
```

## The Flowchart

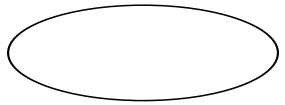
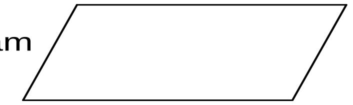
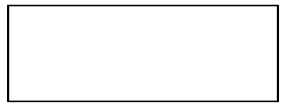
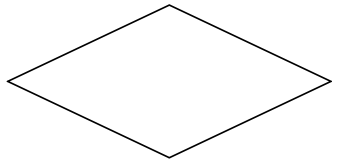
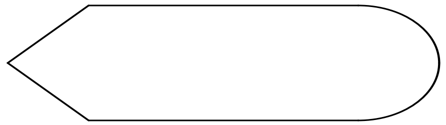
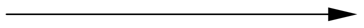
- A graphical representation of the sequence of operations in an information system or program.

### A Flowchart

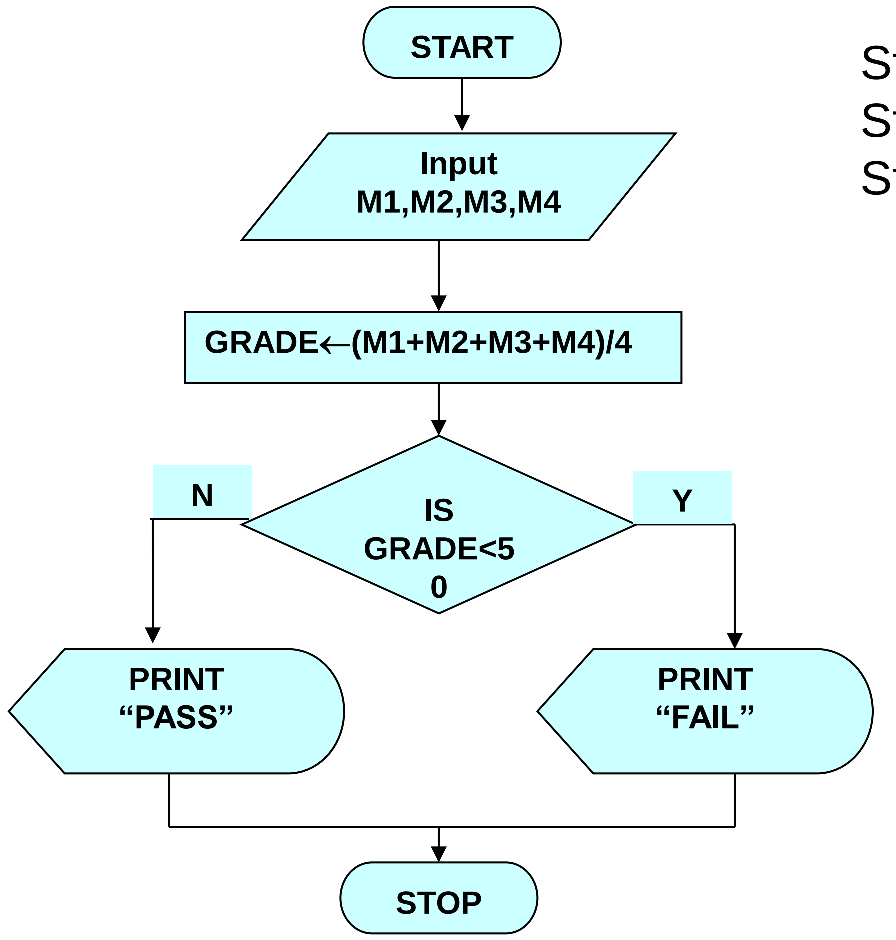
- shows logic of an algorithm
- emphasizes individual steps and their interconnections
- e.g. control flow from one action to the next

# Flowchart Symbols

## Basic

| Name | Symbol | Use in Flowchart |
|---|---|---|
| Oval |  | Denotes the beginning or end of the program |
| Parallelogram |  | Denotes an input operation |
| Rectangle |  | Denotes a process to be carried out<br>e.g. addition, subtraction, division etc. |
| Diamond |  | Denotes a decision (or branch) to be made.<br>The program should continue along one of<br>two routes. (e.g. IF/THEN/ELSE) |
| Hybrid |  | Denotes an output operation |
| Flow line |  | Denotes the direction of logic flow in the program |

# Example



```
Step 1:   Input M1,M2,M3,M4
Step 2:   GRADE ← (M1+M2+M3+M4)/4
Step 3:   if (GRADE <50) then
                    Print “FAIL”
          else
                    Print “PASS”
          endif
```

## Example 3

**Write an algorithm and draw a flowchart that will read the two sides of a rectangle and calculate its area.**

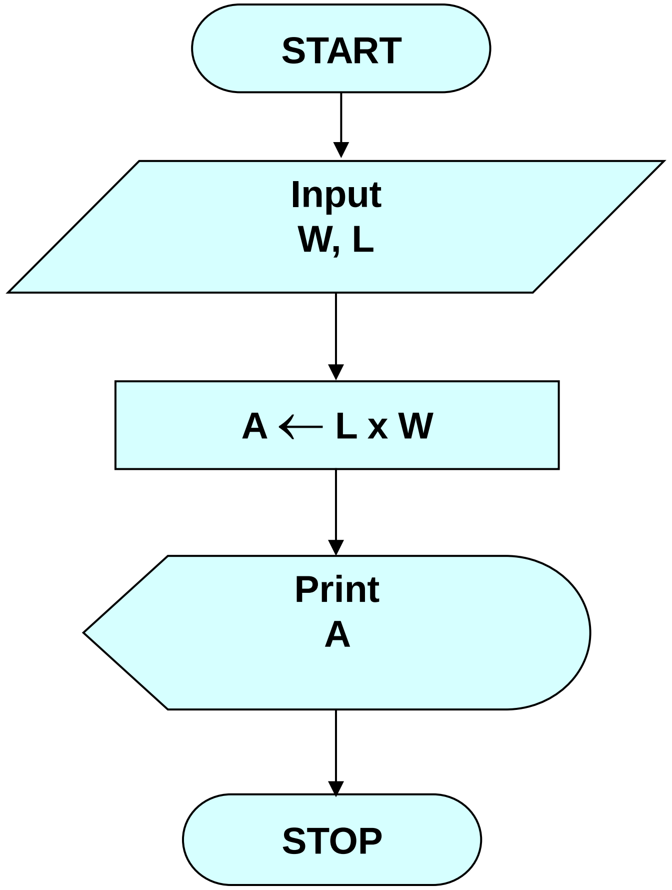
**Pseudocode**

- *Input the width (W) and Length (L) of a rectangle*
- *Calculate the area (A) by multiplying L with W*
- *Print A*

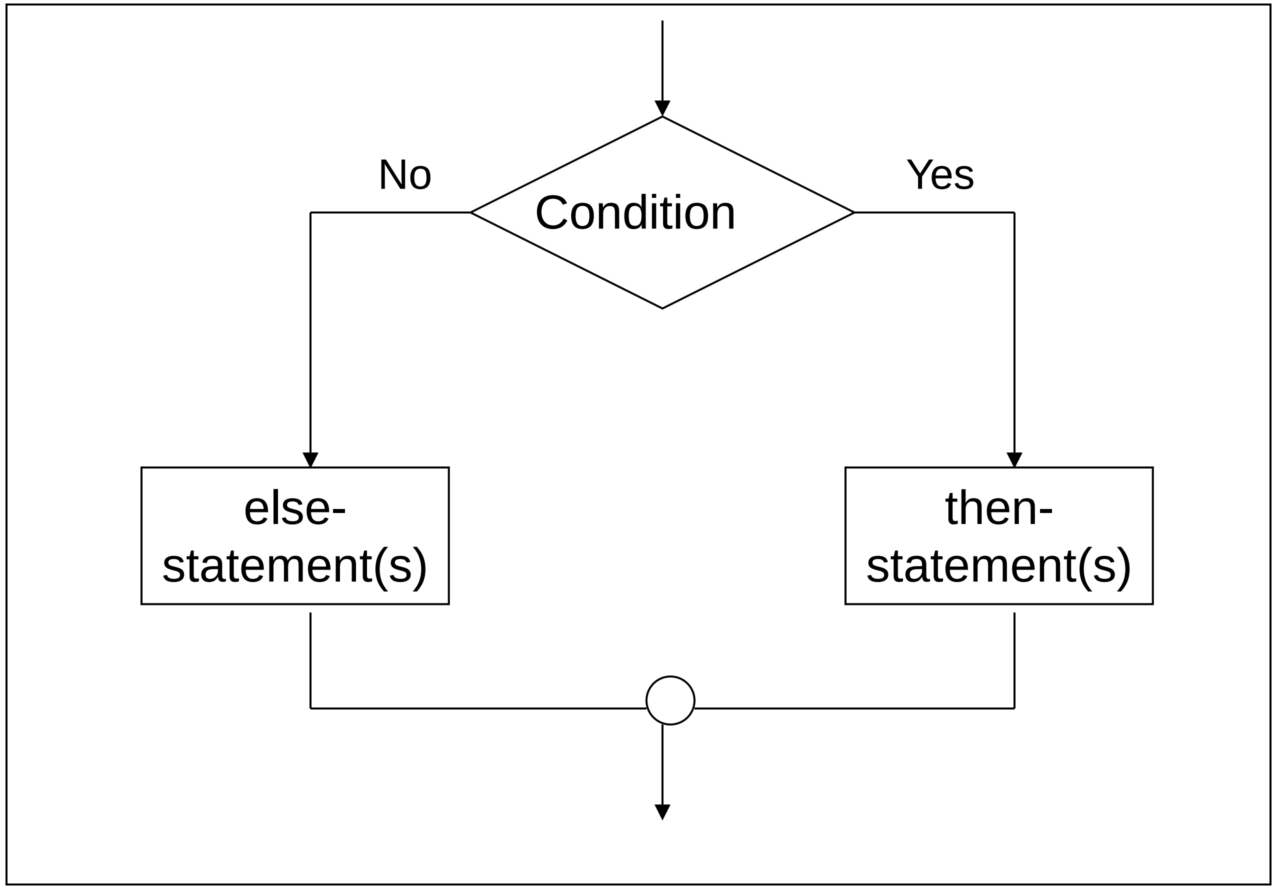
## Example 3

**Algorithm**

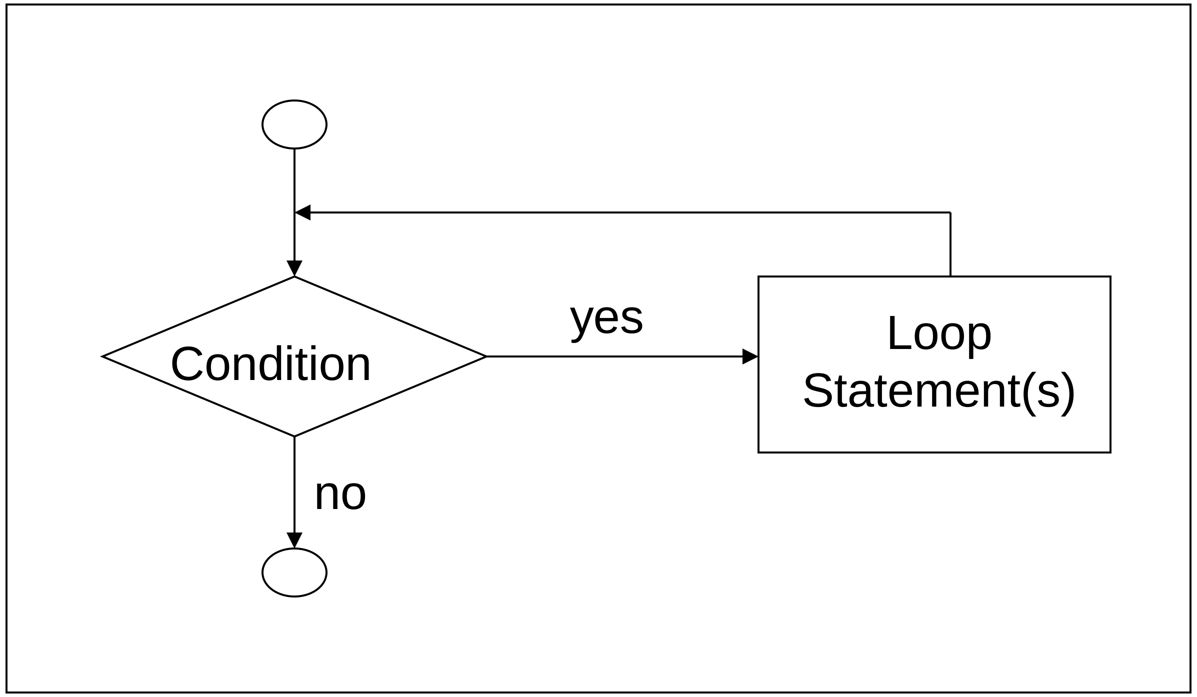
- Step 1: Input W,L
- Step 2: A ← L x W
- Step 3: Print A



## Flowchart – selection control structure



## Flowchart – repetition control structure



Condition
yes
Loop Statement(s)
no