LANGUAGE EVALUATION CRITERIA

The language evaluation criteria are:

- Readability: the ease with which programs can be read and understood.
- Writability: the ease with which a language can be used to create programs.
- Reliability: conformance to specifications (i.e., performs to its specifications).
- Cost: the ultimate total cost.

We will look at them in details:

a) **Readability** - It is the ability to understand the program, and then modify it. Readability has strong relation with the language of choice and the application domains. Readability involves:

   i. Simplicity
      - If it is simple, it is easy to learn.
      - Programmers usually use the features they are familiar with, which are not necessarily the ones that the readers of the program are familiar with.
      - Most often a large number of features will do the same thing, causing unnecessary confusion. That is, multiplicity of features does not help. Example: i = i + 1; i+ = 1; i + +; + + i;
      - Operator overloading gives an operator multiple meanings, which may be confusing when done incorrectly. For example, in some languages, + is used for integer addition, floating point addition, decimal addition and string concatenation
      - Assembly languages are simple, but are too low-level (or not “sophisticated” enough) to express our ideas.

   ii. Orthogonality
      - Features can be combined in a systematic way to form new features.
      - Orthogonality means independent, i.e., if you can do one thing with a feature, there is no reason that you cannot do the same to the other. In a sense the features are symmetric.
      - If a language is more orthogonal, it has fewer exceptions, and may become simpler, but may have unnecessary complexity since the number of legal constructs would be very large.
      - VAX and IBM addition example.
        - Orthogonality requires every possible combination of primitives to be legal. Example: 32bit addition from IBM/VAX assembly languages:

          <u>IBM</u>

          A reg, mem;   adds mem into reg

          AR reg1, reg2;   adds reg2 into reg1

          <u>VAX</u>

          addl op1, op2 ;   op1 and op2 are any reg/mem

          The VAX instruction set exhibits orthogonality
      - Simplicity is the result of a combination of a small number of features through Orthogonality.

   iii. Control Statements
      - Structured programming proclaims blocked control structures with single entry and exit, and discourages the use of goto statements.
      - Goto statements should be used with caution. They should precede their destination, and never jump to distant labels.
      - Limited use of goto statements is still unavoidable in certain languages,

like FORTRAN or BASIC.

iv. Data Types and Structures
- Sufficient data type and data structure support not only improve readability, but also expressibility.
- Adequate facilities for defining data types and structures aids readability.
- Primitive/intrinsic data types should be adequate, too.

v. Syntax Considerations
- Identifier form - should not be too restrictive on length. The length and valid combination of characters.
- Special words - Words such as while, if, end, class, etc., have special meaning within a program. How to express compound statement? Can special words be used as identifiers?
- Form and meaning - How a statement appears should clear indicate what it actually means.

b) **Writability** - Does the language make it easy to write what you have in mind? Readability first, then writability. How easily can the language be used to create programs for a particular domain? Consider Visual Basic (VB) and C for – A graphical game program, – An embedded controller for automotive brakes.

i. Simplicity and Orthogonality
- The combination of a small number of features (simplicity) and the consistent way to combine them to form new features (orthogonality) is crucial.

ii. Support for Abstraction
- Abstraction hides the details (in implementation) of a construct and only provides a clear interface of how this construct should be used. It specifies what a construct works, not how it works.
- Procedure or Process abstraction isolate what a process does from how the process does it. Procedural or process abstraction is a specification that:
  - Describes effects on outputs for given inputs
  - what it does, not how it does it
  - ignores implementation
  - Treats procedure or function as a “black box”
  - Can be applied in any language. Classic example: sorting, that is:

    We just want to call a sort routine when we need to sort something.

    We don’t want to clutter up code by implementing a sort algorithm every time.
- Data abstraction separate what an object should behave given certain events from how it implements the actions.

iii. Expressivity
- The ability to handle data in meaningful, natural ways.
- The language should have convenient way to express computation, e.g., goto may be sufficient, but while loop is better.
- Abstract Data Types:
  - ➢ Extend procedural abstraction to data. Example: type float.
  - ➢ Extends imperative notion of type by:

Providing encapsulation of data/functions.
Separation of interface from implementation.

c) Reliability

Reliability is the property of performing to specifications under all conditions. Many design considerations contribute to reliability. The question here is "Will the program crash easily"?

i. Type Checking
   - Is the process of verifying and enforcing the constraints of types, and it can occur either at compile time (i.e. statically) or at runtime (i.e. dynamically). That is, checks errors either at runtime or compile time.
   - A language is statically-typed if the type of a variable is known at compile time instead of at runtime. Common examples of statically-typed languages include Ada, C, C++, C#, Java, Fortran, Haskell, ML, Pascal, and Scala.
   - Dynamic type checking is the process of verifying the type safety of a program at runtime. Common dynamically-typed languages include Groovy, JavaScript, Lisp, Lua, Objective-C, PHP, Prolog, Python, Ruby, Smalltalk and Tcl.
   - Runtime checks are significantly more expensive.
   - Early detection is less expensive than correcting released code -
     But forcing early detection of type errors reduces writability and expressiveness.
       - Many scripting languages freely cast types at runtime which can result in bizarre and difficult to detect errors later.
   - Compile-time type checking is cheaper, but for many dynamic execution languages, run-time checking is necessary.
   - Type checking includes arguments in function calls, assignment, index range in certain languages (e.g., PASCAL).

ii. Exception Handling
   - Refers to built-in mechanisms to intercept runtime errors, handle them and continue normal execution
   - Possible in any language, but some languages have better facilities than others.

iii. Aliasing
   - To have more than one method to access the same memory.
   - Two or more distinct names that refer to the same memory location or object.
   - Widely regarded as a fruitful source of error.
   - Difficult to detect and almost impossible to prevent.

iv. Readability and Writability
   - A reliable program is written with the best method, the best method is implemented only when we understand what exactly we want to do.

- A language that does not support natural ways of expressing an algorithm will require the use of "unnatural" approaches, and hence reduced reliability.
- It is possible to write unreadable code in any language.

d) Cost

- Cost to train the programmers → Simplicity and orthogonality.
- Cost to write programs → Writability
- Cost to compile the program → Quick-and-dirty or highly optimized.
- Cost to execute the program → Most likely determined by what language you use.
- Cost to build the compiler and runtime support system → The complexity to build compiler should be considered.
- Cost to recover from problems
- Cost to maintain the program → Readability, Most of the software costs are in maintenance, not developments.

e) Other Criteria

- Portability - The ease with which programs can be moved from one implementation/platform to another.
  - If a standard is established, portability is usually not a problem.
- Generality - The applicability to a wide range of applications.
- Well-definedness - The completeness and precision of the language's official definition.

LANGUAGE DESIGN TRADE-OFF

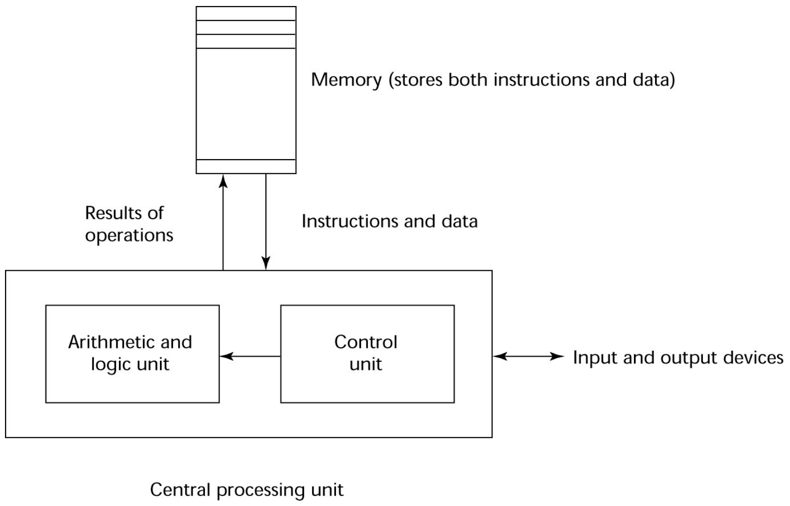
- Reliability vs. Cost of execution. Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs.
- Readability vs. Writability. Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability.
- Writability (flexibility) vs. Reliability. Example: C++ pointers are powerful and very flexible but are unreliable.

INFLUENCES ON LANGUAGES

i. Computer Architecture

- By 1950, the basic architecture of digital computers had been established (and described nicely in John Von Neumann's EDVAC report).
- A computer's machine language is a reflection of its architecture, with its assembly language adding a thin layer of abstraction for the purpose of making easier the task of programming.
- The modern computers are based upon Von Neumann architecture, in which both data and program are stored in memory.
  - Program is fetched and executed, and modifying data in the process.
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
- Most of the programming languages based on Von Neumann architecture are called imperative languages, in which machine state is kept in variables, and the operation modify the variables to perform desired computation.
- On the other hand, functional (or applicative) languages perform computation by applying a series of function on the given parameters.

The Von Neumann Architecture



ii. Programming Methodology

- Advances in methods of programming also have influenced language design.
- Refinements in thinking about **flow of control** led to better language constructs for selection (i.e., if statements) and loops that force the programmer to be disciplined in the use of jumps/branching (by hiding them). This is called **structured programming**.
- Software development cost has become much higher than hardware costs. Software productivity becomes an important issue.

- Before the 1960's, hardware was much more expensive than software.
- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
  - structured programming
  - top-down design and step-wise refinement
- Late 1970s: Shift from Procedure-oriented to data-oriented

  - data abstraction **(abstraction** is the process by which data and programs are defined with a representation similar to its meaning (semantics), while hiding away the implementation details)
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

  (inheritance suggests an object is able to inherit characteristics from another object)
  (Polymorphism is the capability of an action or method to do different things based on the object that it is acting upon)

Programming Methodology teaches the widely-used programming language along with good software engineering principles. Emphasis is on good programming style and the built-in facilities of the language. It is the effective programming practices that result in the development of correct programs.

Good programming methodology?

- A methodology that enables the lowest-cost and on-schedule development of programs that are correct, easy to maintain & enhance.

Correct program?

- A program with correct syntax & semantics

Readable program?

- A program that is easy to read & understand, and therefore, easy to maintain & enhance

Steps of Programming

In order to develop a good program, the following steps are to be followed:

- Step 1: Analysis and understanding of problems
- Step 2: designing of algorithm for the problem
- Step 3: Testing the correctness of the algorithm to make sure it solves the problem
- efficiently and accurately.
- Step 4: Translating the algorithm into a suitable programming language.
- Step 5: Testing and executing the program. That is, the program developed is compiled and executed.
- Step 6: documenting the program, which is very essential in programming. Two types of documentation: Internal and external
  - ➢ Internal involves putting comments and white spaces in our program source codes while
  - ➢ External are in form of manual for operating and maintaining the program by any user of the program.

Good Programming styles

- Choose appropriate names for your variables.
- Special care must be given to the choice of names for procedures, functions, constants and all variables and types used in different parts of your program.

- Avoid using attractive names whose meaning has little or nothing to do with the problem.
- Avoid choosing names that are too close to each other in spelling or otherwise easy to confuse, for example, Num1, Num2 etc
- Be careful in the use of letter i(small letter i), l(small l), O(capital O) and 0(zero)
- Choose appropriate data types for your variables
- The program must be well structured for readability
- Modularize the program for easy maintenance
- Appropriate documentation formats such as comments and whitespaces must be used in the program.
- Test the program with different sets of data(both valid and invalid)

Errors in programming

Errors are defect or bugs in a program.

- Testing: The tasks performed to determine the existence of defects in a program.
- Debugging: The tasks performed to detect the exact location of defects in a program.

Types:

- **Syntax Errors** - They are caused by the code that somehow violates the rules of the language.
- **Semantic Errors -**Occur when a statement executes and has an effect not intended by the programmer. Hard to detect during normal testing. Often times occur only in unusual and infrequent circumstances.
- **Run-Time Errors** – Occur when the program is running and tries to do something that is against the rules.
- **Logic error –** usually made by programmers. The program will definitely compile and run very well but the result/output gotten will have errors.
- **Compile time errors** – They are errors brought out during the execution stage of the program development. For example, when we try to divide a number by zero, it causes memory overflow error.

PROGRAMMING PARADIGMS/LANGUAGE CATEGORIES

- Paradigm can also be termed as method to solve some problem or do some task.
- Programming paradigm is an approach to solve problem using some programming language.
- We can say it is a method to solve a problem using tools and techniques that are available to us following some approach.

❖ Fundamentally, languages can be broken down into two types:

- Imperative languages - in which you instruct the computer how to do a task
- Declarative languages - in which you tell the computer what to do.

1. IMPERATIVE PROGRAMMING PARADIGMS

- The oldest programming model
- It features close relation to machine architecture.
- You tell the compiler what you want to happen, step by step.
- Resolve a certain set of problems by describing the steps required to find solution.
- Describes computation in terms of statements that change a program state.
- 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.
- Language has assignments, loops, conditionals, procedure and functional calls, and control flow.
- It works by changing the program state through assignment statements.
- Example imperative languages:
  - C : developed by Dennis Ritchie and Ken Thompson
  - Fortran : developed by John Backus for IBM
  - Basic : developed by John G Kemeny and Thomas E Kurt

❖ Imperative programming is divided into: Procedural and Object Oriented Programming.

a) Procedural programming paradigm

- Derived from Structured programming, based upon the concept of procedure call.
- It relies on procedures or subroutines to perform computations.
- A program consists of data and modules/procedures that operate on the data.
- There is no difference between procedural and imperative approach.
- It has the ability to reuse the code and it was boon at that time when it was in use because of its reusability.
- Examples of Procedural programming paradigm:
  - C : developed by Dennis Ritchie and Ken Thompson
  - C++ : developed by Bjarne Stroustrup
  - Java : developed by James Gosling at Sun Microsystems
  - ColdFusion : developed by J J Allaire
  - Pascal : developed by Niklaus Wirth

b) Object Oriented Programming paradigm

- An OO Program is a collection of objects that interact by passing messages that transform the state.
- The smallest and basic entity is object and all kind of computation is performed on the objects only.
- An object is an instance of a class, which is an encapsulation of data (called fields) and the procedures (called methods) that manipulate them.
- An object therefore is like a miniature program or a self-contained component, which makes the OOP approach more modularized and thus easier to maintain and extend.
- Concepts of OOP:
  Abstraction, Inheritance, Polymorphism, Encapsulation, Sending Messages
- Examples of Object Oriented programming paradigm:
  - Simula: first OOP language
  - Java: developed by James Gosling at Sun Microsystems
  - C++: developed by Bjarne Stroustrup
  - Objective-C: designed by Brad Cox
  - Visual Basic .NET: developed by Microsoft (not considered as full OOP)
  - Python: developed by Guido van Rossum
  - Ruby: developed by Yukihiro Matsumoto
  - Smalltalk: developed by Alan Kay, Dan Ingalls, Adele Goldberg

2. DECLARATIVE PROGRAMMING PARADIGM

- Defines what needs to be accomplished by the program without defining how it needs to be implemented.
- It focuses on what needs to be achieved instead of instructing how to achieve it.
- In computer science, the declarative programming is a style of building programs that expresses logic of computation without talking about its control flow.
- It often considers programs as theories of some logic.
- It just declares the result we want rather how it has been produced.
- This is the only difference between imperative (how to do) and declarative (what to do) programming paradigms.

❖ Declarative programming is divided into: Logic, Functional, and Database.

a) Logic programming paradigms

- Logic programming declares what outcome the program should accomplish, rather than how it should be accomplished.
- In logic programming, we have a knowledge base which we know before and along with the question and knowledge base which is given to machine, it produces result.
- In logical programming, the main emphasis is on knowledge base and the problem.
- The execution of the program is very much like proof of mathematical statement.
- 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

b) Functional programming paradigms

- Functional programming models a computation as a collection of mathematical functions.
- The central model for the abstraction is the function which are meant for some specific computation and not the data structure.
- Data are loosely coupled to functions.
- Function can be replaced with their values without changing the meaning of the program.
- Functional languages are characterized by:
    - Functional composition
    - Recursion
    - Conditional evaluation
- Examples of Functional programming paradigm:
    - JavaScript : developed by Brendan Eich
    - Haskell : developed by Lennart Augustsson, Dave Barton
    - Scala : developed by Martin Odersky
    - Erlang : developed by Joe Armstrong, Robert Virding
    - Lisp : developed by John Mccarthy
    - ML : (Meta Language) developed by Robin Milner
    - Clojure : developed by Rich Hickey

c) Database/Data driven programming approach

- This is based on data and its movement.
- Program statements are defined by data rather than hard-coding a series of steps.
- A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions.
- There are several programming languages that are developed mostly for database application.
- For example, SQL.
- It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs.