Chapter 1

Introduction

Topics

- Reasons for Studying Concepts of Programming Languages
- Programming Domains
- Language Evaluation Criteria
- Influences on Language Design
- Language Categories
- Language Design Trade-Offs
- Implementation Methods
- Programming Environments

WHY DO WE NEED TO STUDY THE CONCEPTS OF PLS?

- Increased ability to express ideas
  - ∴hence reducing the limitations posed by a particular programming language.
- Improved background for choosing appropriate languages
  - Avoid from sticking to traditional programming languages
- Increased ability to learn new languages
  - top-down approach to learning programming

Why do we need to study the concepts of PLs?

- Better understanding of significance of implementation
  - $\therefore$ More intelligent use of a language that suits its design purpose
- Better use of languages that are already known
  - Optimum use of a known language
- Overall advancement of computing
  - The need to choose which language to use

Programming Domains

- Scientific applications
  - Large number of floating point computations
  - Efficiency is the primary concern
  - Fortran – still in use to date
- Business applications
  - Producing reports, describing and storing decimal numbers and character data, specifying decimal arithmetic operations
  - COBOL – still the most commonly used language
- Artificial intelligence
  - Symbols rather than numbers manipulated
  - LISP, Prolog, C
- Systems programming
  - Need efficiency because of continuous use
  - Machine dependent
  - PL/I, PL/S (IBM), BLISS (Digital), Extended ALGOL (UNISYS), C (UNIX)

Programming Domains

- Web Software
  - Initially for presentation but gradually evolves to include dynamic content.
  - Eclectic collection of languages: markup (e.g., XHTML), scripting (e.g., PHP), general-purpose (e.g., Java)

Language Evaluation Criteria

- **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

Evaluation Criteria: Readability

- Overall simplicity
  - A manageable set of features and constructs
    - A language with a large number of basic constructs is more difficult to learn than one with a smaller number
  - Few feature multiplicity (means of doing the same operation)
    - count = count + 1
    - count += 1
    - count ++
  - Minimal operator overloading
  - in which a single operator
    - symbol has more than one meaning.
- Orthogonality
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways to build data structures
  - Every possible combination is legal
- Data types
  - The presence of adequate facilities for defining data types
- Syntax design
  - Identifier forms: should limit restrictions to allow flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords

Evaluation Criteria: Writability

- Simplicity and orthogonality
  - Few constructs, a small number of primitives, a small set of rules for combining them.
- Support for abstraction
  - The ability to define and use complex structures or operations in ways that allow details to be ignored.
    - G = STD(X,Y,Z)
  - Two distinct categories: process and data abstractions.
- Expressivity
  - A set of relatively convenient ways of specifying operations.
    - count ++
  - Example: the inclusion of `for` statement in many modern languages.

Evaluation Criteria: Reliability

- Type checking
  - Testing for type errors
  - Run-time type checking is expensive,
  - Compile-time type checking is more desirable.
- Exception handling
  - Intercept run-time errors and take corrective measures
- Aliasing
  - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
  - A language that does not support "natural" ways of expressing an algorithm will necessarily use "unnatural" approaches, and hence reduced reliability

Evaluation Criteria Characteristics

	CRITERIA		
Characteristic	Readability	Writability	Reliability
Simplicity	✓	✓	✓
Orthogonality	✓	✓	✓
Data types	✓	✓	✓
Syntax design	✓	✓	✓
Support for abstraction		✓	✓
Expressivity		✓	✓
Type checking			✓
Exception handling			✓
Restricted aliasing			✓

Evaluation Criteria: Cost

- Training programmers to use language
- Writing programs (closeness to particular applications)
- Compiling programs
- Executing programs
- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs

Other Evaluation Criteria

- Portability
  - The ease with which programs can be moved from one implementation to another
- Generality
  - The applicability to a wide range of applications
- Well-definedness
  - The completeness and precision of the language’s official definition

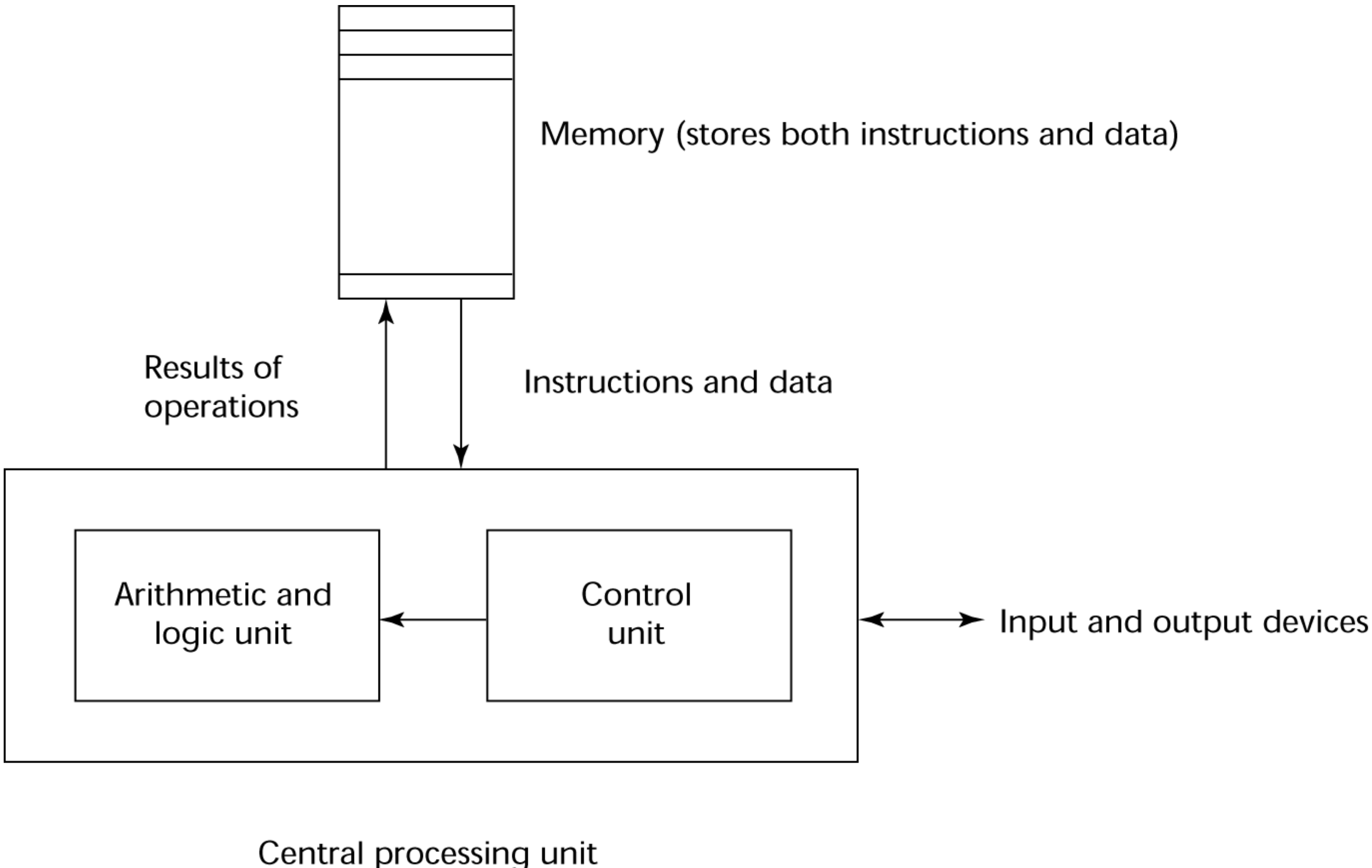
Influences on Language Design

- Computer Architecture
  - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture
- Programming Methodologies
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

Influences on Language Design: Computer Architecture

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient

Influences on Language Design: Computer Architecture



THE VON NEUMANN ARCHITECTURE

Influences on Language Design: Programming Methodologies

- 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: Process-oriented to data-oriented
  - data abstraction
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

Language Categories

- Imperative
  - Central features are variables, assignment statements, and iteration
  - Examples: C, Pascal
- Functional
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme
- Logic
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog
- Object-oriented
  - Data abstraction, inheritance, late binding
  - Examples: Java, C++
- Markup
  - New; not a programming per se, but used to specify the layout of information in Web documents
  - Examples: XHTML, XML

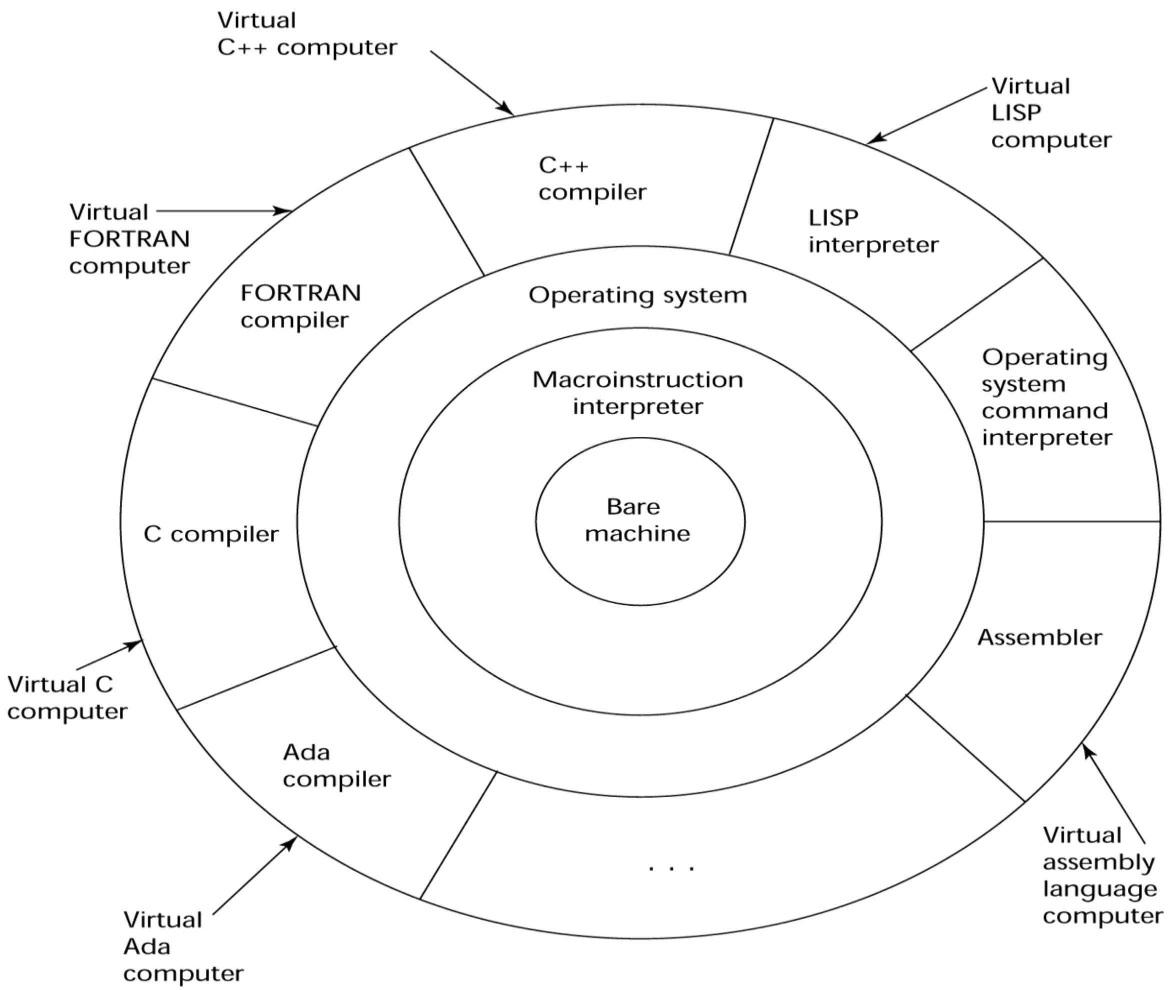
Language Design Trade-Offs

- Reliability vs. cost of execution
  - Conflicting criteria
  - Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs
- Readability vs. writability
  - Another conflicting criteria
  - 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
  - Another conflicting criteria
  - Example: C++ pointers are powerful and very flexible but not reliably used

Implementation Methods

- Compilation
  - Programs are translated into machine language
- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems
  - A compromise between compilers and pure interpreters
- Preprocessors
  - Programs are processed immediately prior to compilation

Implementation Methods



Virtual layered view of a typical computer system

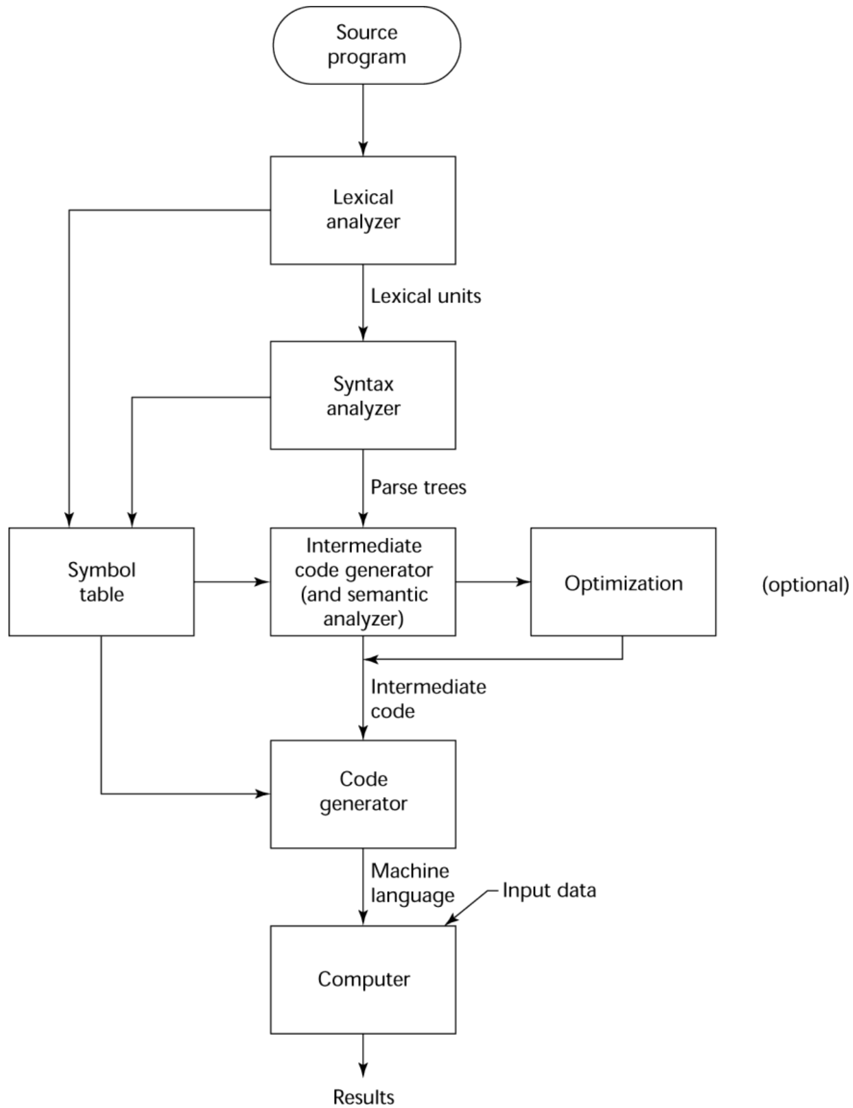
Implementation Methods: Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Phases of a compilation process:
  - lexical analysis: converts characters in the source program into lexical units
  - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  - Intermediate code generator: generate intermediate code
  - Semantics analysis: checks for errors unidentifiable during the previous phases
  - Optimisation: improves program (often optional)

Implementation Methods: Compilation

- Phases of a compilation process (continued):
  - Code generation: translate (optimised) intermediate code into equivalent machine code
- Once compiled, the machine code produced often requires programs from the operating system in order to run. A **linker** links them together to produce a **load module**. A linker also links user program with previously compiler (user) programs.

Implementation Methods: Compilation



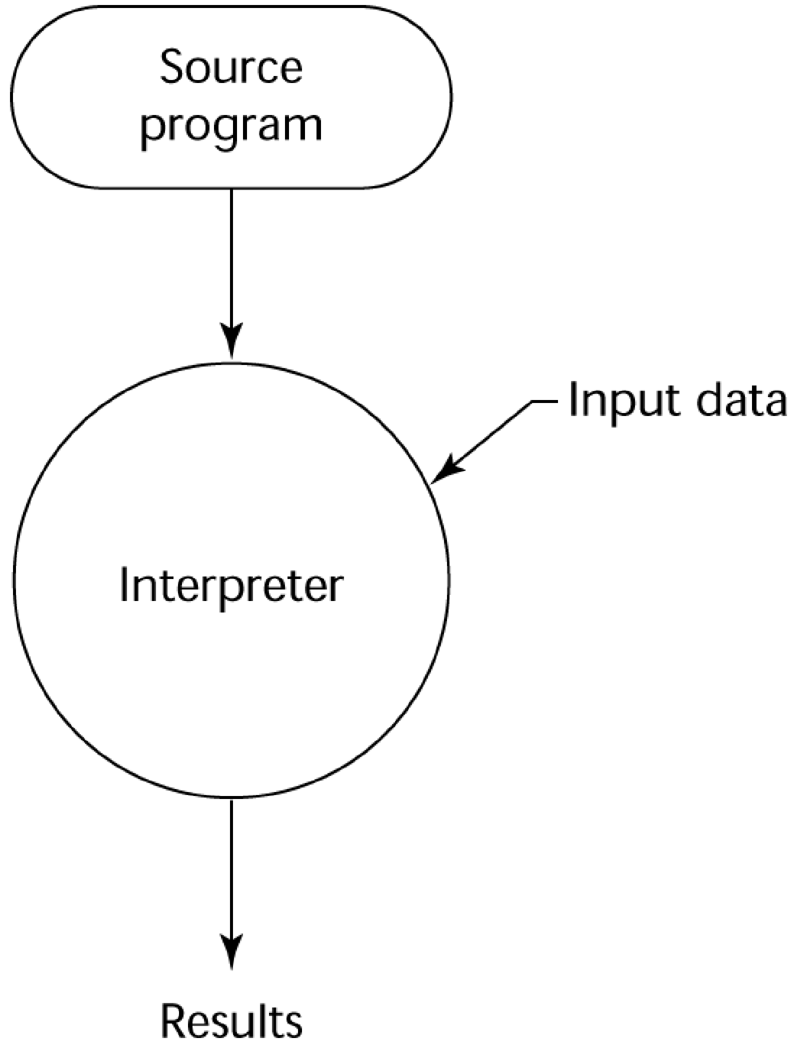
Implementation Methods: Compilation

- An issue to be considered in compilation is the von Neumann bottleneck, which is referring to the connection between a computer's memory and its processor
- The fact that program instructions can often be executed a lot faster than they (the instructions) can be moved to the processor for execution causes *bottleneck*.
- Hence, this connection is the primary limiting factor in the speed of von Neumann architecture computers.

Implementation Methods: Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Becoming rare on high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript)

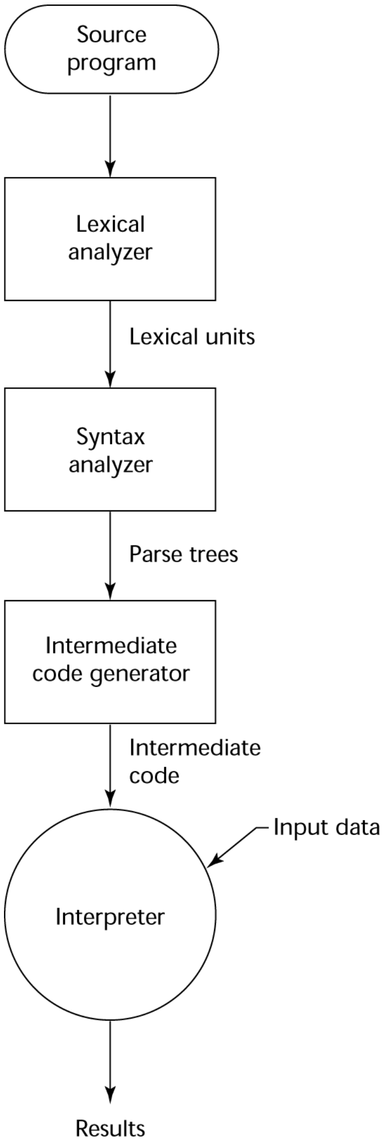
Implementation Methods: Pure Interpretation



Implementation Methods: Hybrid Implementation

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)

Implementation Methods: Hybrid Implementation



IMPLEMENTATION METHODS: HYBRID IMPLEMENTATION

https://www.youtube.com/watch?v=qaj7nO1HUqA

Implementation Methods: Preprocessors

- A program that processes a program immediately before it is compiled
- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well-known example: C preprocessor
  - expands `#include`, `#define`, and similar macros

Programming Environments

- The collection of tools used in software development
- UNIX
  - An older operating system and tool collection
  - Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that run on top of UNIX
- Borland JBuilder
  - An integrated development environment for Java
- Microsoft Visual Studio.NET
  - A large, complex visual environment
  - Used to program in C#, Visual BASIC.NET, Jscript, J#, or C++

SUMMARY

- The study of programming languages is valuable for a number of reasons including:
  - Increase our capacity to use different constructs
  - Enable us to choose languages more intelligently
  - Makes learning new languages easier
- Most important criteria for evaluating programming languages are:
  - Readability, writability, reliability, cost
- Major influences on language design have been machine architecture and software development methodologies
- The major methods of implementing programming languages are: compilation, pure interpretation, hybrid implementation and preprocessor