Preliminaries

Programming Language

1. What is it?

A programming language is a formal system of rules and symbols used by humans to communicate instructions to a computer. “It is an artificial language designed to express computations or algorithms that can be performed by a computer”. – Wikipedia

2. What are the instructions?

They are known as code, written by programmers.
A program is the coding of an algorithm that follows this :
Input → Calculations → Output
It is readable by both man and machine.

3. Why are there so many Programming Language?

Programming languages vary in:

- **Syntax**: the structure or form of the language (it's not directly said in the slides)
- **Semantics**: the meaning of the instructions (it's not directly said in the slides)
- **Purpose**:
    - General-purpose: used for a wide range of applications
    - Specialized: designed for specific tasks

page - 5 - Reasons for Studying Concepts of Programming Languages

---

1. Increased Ability to Express Ideas

The language a programmer uses limits how they think and solve problems.
Each language provides specific control structures, data structures, and abstractions, which affect the kinds of algorithms that can be written.

Example:
A programmer who only knows C may not easily solve problems that require associative arrays. After learning Perl, the programmer can use associative arrays to express ideas more clearly and efficiently.

---

2. Improved Background for Choosing Appropriate Languages

Many programmers learned programming years ago using languages that are no longer common. As a result, they often choose the language they know best, even if it is not suitable for the task.

---

3. Increased Ability to Learn New Languages

Learning a new programming language can be difficult if a programmer only knows one or two languages and has never studied language concepts.

Understanding fundamental programming language concepts makes learning new languages much easier.

Example:
A programmer who understands object-oriented programming concepts will find it easier to learn Ruby.

---

4. Better Understanding of Significance of Implementation

Knowing how a language is implemented explains why it is designed in a certain way.
This understanding helps programmers use the language correctly and intelligently.

Some bugs can only be understood and fixed by knowing implementation details.

Example:
Understanding pass-by-value and pass-by-reference

Knowing how the stack(last in first out) works helps visualize how function calls are executed.

Better Use of Languages Already Known

Modern programming languages are large and complex.
Most programmers use only a small portion of the language features they know.

This course leads programmers to learn about previously unknown and unused parts of the languages they already use and begin to use those features.

---

5. Overall Advancement of Computing

Popular programming languages are not always the best languages available.
Sometimes languages become widely used because decision-makers are not familiar with alternatives.

page 12-Programming Domains

- **Scientific Applications:**
  - Appeared in late 1940s–early 1950s
  - large numbers of floating-point computations
  - arrays and matrices the most common data structures
  - counting loops and selections most common control structures
  - Fortran, ALGOL 60

- **Business Applications:**
  - Appeared in early 1950s
  - report generation
  - decimal numbers and characters
  - precise decimal arithmetic
  - COBOL

- **Artificial Intelligence:**
  - Symbolic processing rather than numeric
  - Use linked lists - LISP, Prolog
  - Now modern AI are written using MATLAB, Python, R
  - LISP

- **Systems Programming:**
  - Need efficiency because of continuous use
  - C

- **Web Software:**
  Markup (XHTML), scripting (PHP) , and general-purpose languages (Java)

- **Mobile Applications**

Language Evaluation Criteria

- **Readability:** Ease of reading and understanding programs.

- **Writability:** Ease of creating programs.

- **Reliability:** Conformance to specifications.

- **Cost:** Total cost of using the language.

Readability

- **Overall Simplicity:**
  - A manageable number of features and constructs improves readability.
  - Languages with many basic constructs are harder to learn.
  - Programmers often learn only a subset of a large language.
  - Readability problems occur when programmers use different subsets.

  Minimal Feature Multiplicity:

  - Fewer ways to perform the same operation improve readability.
  - Example java:

    ```
    count = count + 1
    count += 1 ..etc
    ```

  Minimal Operator Overloading:

  - Each operator should have only one meaning.
  - Example in C++:

    count = 4 if (count = 5)

- **Orthogonality(تناسق وبساطة تركيب عناصر اللغة):**

  A relatively small set of primitive constructs combined in a small number of ways

  - Every possible combination is legal
  - Primitive data types: integer, float, double, character
  - Two type operators: array and pointer
  - Lack of orthogonality leads to exceptions to language rules

- **Control Statements:**
  Presence of well-known control structures (e.g., while statement).

- **Data Types and Structures:**

  Adequate predefined types and facilities for defining structures.

  - **Example:**
    - `timeout = 1` → unclear meaning
    - `timeout = true` → clear meaning

- **Syntax Considerations:**
  Identifier forms: flexible composition.
  - Special words and compound statements: rules for forming them.
  - Form and meaning: self-descriptive constructs, meaningful keywords.
    - Example: In Fortran 95, `Do` and `End` can be variable names; their appearance may or may not be special.
  - Using words that indicate their purpose aids readability.
    - Example: Same variable defined as local inside a function and global outside all functions.

Writability

Definition: Ease of using a language to create programs for a chosen problem domain

Fair Comparison: Comparing writability only makes sense if both languages were designed for the application.

- **Example:** Visual BASIC vs C for GUI programs – VB is ideal

- **Simplicity and Orthogonality:**
  - Few constructs, small number of primitives, small set of rules for combining them.
  - Large number of constructs may reduce familiarity for programmers.
- **Support for Abstraction:**
  - Ability to define and use complex structures or operations while ignoring details.
- **Expressivity :**
  - Convenient ways to specify operations (e.g., `for` statements).
  - Strength and number of operators/predefined functions (e.g., `count++` vs `count = count + 1`).

Reliability(الموثوقية)

A program is said to be reliable if it performs to its specifications under all conditions

- **Type Checking:**
    - **What it is:** Testing for type errors either during compilation (before running) or at run-time (while running).
    - **Efficiency:** Compile-time checking is preferred because it is less "expensive" (doesn't slow down the program).
    - **Cost of Repair:** The earlier an error is found, the cheaper and easier it is to fix.
    - Presence of two or more distinct referencing methods for the same memory location

- **Exception Handling:**
    - Intercept run-time errors and take corrective measures.
    - **Languages with extensive support:** Ada, C++, Java, C# for exception handling, and nonexistent for example C

- **Aliasing :**
    - Multiple references to the same memory location (e.g., pointers in C++, array name in Java).
    - Restricting aliasing increases reliability.

- **Readability and Writability:**
    - Poor support for “natural” expression reduces reliability.
    - Easier-to-write programs are more likely correct.
    - Difficult-to-read programs are hard to write and modify.

Cost

Total cost depends on multiple language characteristics:

- **Training Programmers:** depends on simplicity and orthogonality.
- **Writing Programs:** depends on writability (closeness to application).
- **Compiling Programs.**
- **Executing Programs:**
    - Languages with many run-time type checks execute slower.
    - Trade-off possible between compilation cost and execution speed.
- **Language Implementation System:**
    - Availability and cost of compilers affect adoption.

- **Reliability:** Poor reliability increases costs, especially for critical systems.
- **Maintenance:** The cost of software maintenance depends on a number of language characteristics, primarily readability

---

Others

- **Portability (قابلية النقل):** Ease of moving programs between implementations.
- **Generality (العمومية):** Applicability to a wide range of applications.
- **Well-definedness (الدقة في التعريف):** Completeness and precision of the language's official definition.

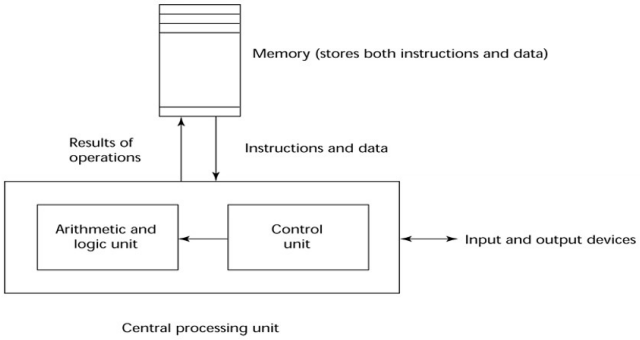
31-Influences on Language Design

Language design is primarily shaped by **two forces**:

- **Computer Architecture**
- **Programming Methodologies**

A. Computer Architecture: Von Neumann Model

- Most modern programming languages are **imperative** because they were designed to fit the **von Neumann architecture**.
- Characteristics of the von Neumann model:
    - Data and programs are stored in memory.
    - Memory is separate from the CPU.
    - Data and instructions are moved from memory to the CPU.
    - Basis for imperative languages:
        - **Variables** model memory cells.
        - **Assignment statements** model data movement (piping).
        - **Iteration** is efficient on this architecture.



Fetch-execute-cycle (on a von Neumann architecture computer)

```
initialize the program counter
repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```

B. Programming Methodologies (Evolution)

- **1950s–1960s**: Focus on **machine efficiency**.
- **Late 1960s**: Focus on **people efficiency**.
    - Emphasis on readability and better control structures.
    - Structured programming.
    - Top-down design and step-wise refinement.
- **Late 1970s**:
    - Shift from process-oriented to **data-oriented design**.
    - Data abstraction.
    - **SIMULA 67** was the first language to provide limited support for data abstraction.
- **Mid-1980s**:
    - **Object-oriented programming**.
    - Data abstraction + inheritance + polymorphism.

Programming Language Categories

Programming languages are categorized into **four categories**:

- Imperative
- Functional
- Logic
- Markup / programming hybrid

Languages that support **object-oriented programming are not considered a separate category**, because:

- Most object-oriented languages grew out of **imperative languages**.
- Example: expressions, assignment statements, and control statements of **C and Java** are nearly identical.

- **Imperative**
  - Central features: **variables, assignment statements, iteration**.
  - Includes:
    - Languages that support object-oriented programming.
    - Scripting languages.
    - Visual languages.
  - Examples: **C, Pascal, Java, Perl, JavaScript, Visual BASIC .NET, C++**.

- **Functional**
  - Computation by **applying functions to given parameters**.
  - Examples: **LISP, Scheme, ML, F#**.

- **Logic**
  - **Rule-based** languages.
  - Rules are specified in **no particular order**.
  - Example: **Prolog**.

- **Markup / Programming Hybrid**
  - New category.
  - Not programming languages per se.
  - Used to specify the **layout of information in Web documents**.
  - Extended to support some programming.
  - Examples: **HTML, XHTML, XML, JSTL, XSLT**.

- **Object-Oriented**
  - Data abstraction, inheritance, late binding.
  - **Examples:** Java, C++.
  - ***Note*:** Object-oriented programming is a style supported by some imperative languages, not a separate category.

40-1st View: Imperative & Procedural

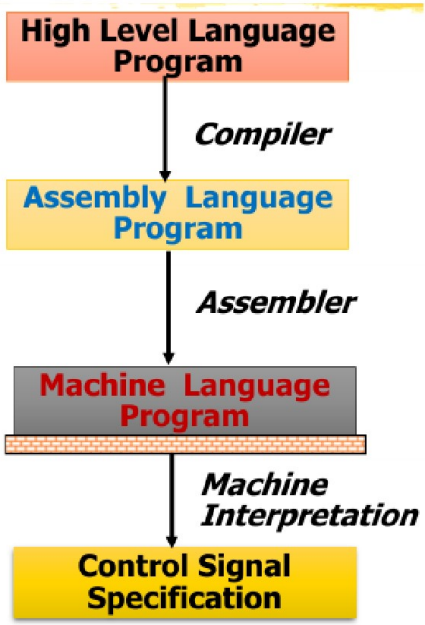
- Computers take commands and perform operations.
- Programming is like issuing procedural commands to the computer.
- Since most computers use the von Neumann architecture, programming languages mimic the architecture.

Imperative Languages and Architecture

- Imperative languages (e.g., C, C++, Java) dominate programming.
- Mapping between language and hardware:
    - Variables ↔ memory cells.
    - Assignment statements ↔ data movement between memory and CPU.
    - Operations and expressions ↔ CPU execution.
    - Explicit control flow ↔ program counter.
- **Advantages:**
    - Efficient mapping between language and hardware for good execution performance
- **Limitation:**
    - Restricted by the von Neumann bottleneck.

Layers of Abstraction / Translation

- All layers follow the imperative model.



2nd View: Functional

- Programming is like solving mathematical functions.
- Programs and subprograms are implementations of functions.
- Computation by applying functions to parameters.

Functional Language Characteristics

- **No concept of storage-based variables or assignment.**
  - Example: a factorial function in Lisp (defun fact (x) (if (<= x 0) 1 (* x (fact (- x 1)))))
  - **Single-valued variables.**
- **Control through:**
  - Conditional expressions (branches).
  - Recursion (iteration).
- **Dynamic linked lists.**
- **Examples**:
  - ML
  - LISP ♦ 2nd-oldest general-purpose PL still in use (1958)

3rd View: Logic

- Programming is like logical induction.
- Programs expressed as facts and rules in formal logic.
- Execution through rule resolution.
- **Example**: relationship among people

fact:
```
mother(joanne,jake).
father(vern,joanne).
```

rule:
```
grandparent(X,Z) :- parent(X,Y),
                    parent(Y,Z).
```

goal:
```
grandparent(vern,jake).
```

- **Non-procedural**:
  - The programmer supplies facts and inference rules.
  - The system then infers answers to queries/goals .

- **Characteristics:**
  - Highly inefficient.
  - Used in limited application areas (AI, databases).
- **Example language:**
  - Prolog.

```
fact(X,1) :-  X =:= 1.
fact(X,Fact) :-
  X > 1, NewX is X - 1,
  fact(NewX,NF),
  Fact is X * NF.
```

Summary: Language Categories

- **Imperative:**
    - Variables, assignment statements, iteration.
    - Includes object-oriented, scripting, and visual languages.
    - Examples: C, Java, Perl, JavaScript, Visual BASIC .NET.
- **Functional:**
    - Computing by applying functions to given parameters.
    - Examples: LISP, Scheme, ML.
- **Logic:**
    - Rule-based computation.
    - Example: Prolog.

Language Design Trade-Offs

- **Reliability vs execution cost.**
    - Example: Array bounds checking in Java increases cost.
- **Readability vs writability.**
    - Example: APL is compact but hard to read.
- **Writability (flexibility) vs reliability.**
    - Example: C++ pointers are powerful but not reliable (error-prone).

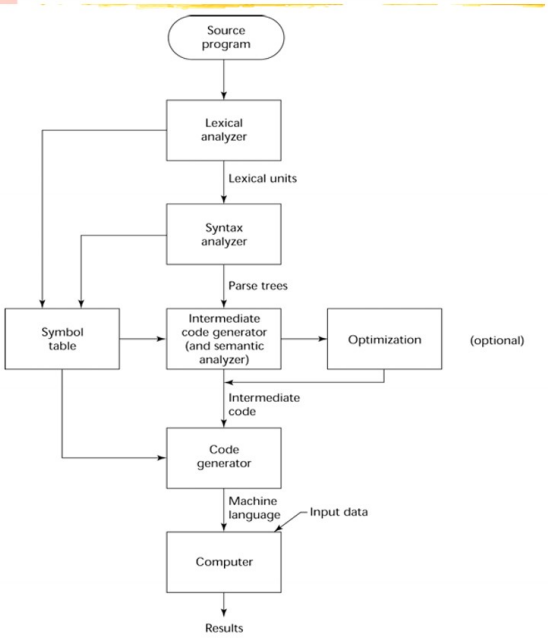
Implementation Methods

- **Compilation:**
    - Translate high-level language to machine language.
    - Slow translation, fast execution.
    - Phases:
        - Lexical analysis:
            - Converts **characters** in the source program into **lexical units**
            - will output **Tokens**
        - Syntax analysis.
            - Transforms **lexical units** into **parse trees** which represent the **syntactic structure** of the program.
        - Semantic analysis.
            - Generates **intermediate code**.
        - Code generation.
            - Converts intermediate code into **machine code**.

example i don't think it's important :

	Lexical Analysis	Syntax Analysis	Semantic Analysis	Code Generation
input	The source code	Sequence of tokens	Parse tree	Intermediate code

output	Tokens generated
	int,
	sum
	(
	int
	a
	,
	int
	B
	)
	{
	Return
	a
	+
	b
	;
	}
		Parse Tree
		ReturnType: int
		FunctionName: sum
		Parameters
		Parameter: int a
		Parameter: int b
		Body
		ReturnStatement
		BinaryExpression
		Identifier: a
		Operator: +
		Identifier: b
			Intermediate Code
			t1 = a + b
			return t1
				Machine Code
				LOAD R1, a
				LOAD R2, b
				ADD R3, R1, R2
				STORE R3, result
				RETURN



Additional Compilation Terminologies

- **Load module (Executable Image):**
  The combination of user code and system code together.

- **Linking and Loading:**
  The process of collecting system programs and linking them to the user program

---

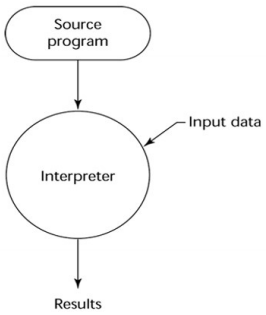
Execution of Machine Code

Execution on a von Neumann architecture follows the Fetch–Execute Cycle:

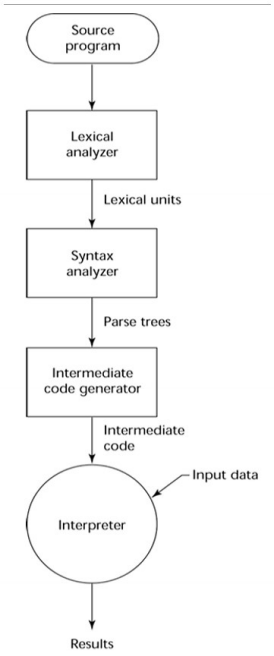
- Initialize the program counter
- Repeat forever:
    - Fetch the instruction pointed to by the counter
    - Increment the counter
    - Decode the instruction
    - Execute the instruction
- End repeat

Pure Interpretation

- **Pure Interpretation:**
  - No translation.
  - Easier error detection.
  - Slower execution.
  - Often requires more space.
  - Rare in high-level languages, but used in web scripting (e.g., JavaScript).



- **Hybrid Implementation Systems:**
  - Combine compilation and interpretation.
  - Use intermediate code.
  - Faster than pure interpretation.
  - Examples:
    - Perl (partial compilation to detect error before interpretation).
    - Java (bytecode + JVM).



Programming Environments

- Tools used in software development.
- **Examples:**
    - **UNIX**
        - An older operating system and tool collection
        - (often used with GUIs like CDE, KDE, GNOME). runs top of unix
    - **Borland JBuilder**
        - (Java IDE).
    - **Microsoft Visual Studio .NET**
        - A large, complex visual environment
        - (C#, VB.NET, Jscript, J# C++).

Summary

- **Studying programming languages:**
    - Increases ability to use different constructs.
    - Helps choose languages intelligently.
    - Makes learning new languages easier.
- **Main evaluation criteria:**
    - Readability, writability, reliability, cost.
- **Main influences on language design:**
    - Machine architecture.
    - Software development methodologies.
- **Main implementation methods:**
    - Compilation.
    - Pure interpretation.
    - Hybrid systems.