Programming Languages mod2

1 Introduction

There are many similar languages, with different syntax. We will study characteristics of various program language paradigms:

- Imperative
- Object-oriented
- Functional

2 Short history of programming languages

- **Hilbert and Ackerman** (1928) presented the **Entscheidungsproblem** (or the decision problem) which was to find an algorithm (mechanical procedure) that if you give a mathematical statement (input) it would say "Yes" or "No" according to whether the statement is valid or not.
- **Church** (1932): **Lambda calculus** which is the foundation for functional programming
  - $\lambda x.x^2$: the function that takes x to x^2
  - computation is defined as applying functions to functions
- **Turing** (1936): computation defined via Turing Machines
- types of programming languages:
  - **Logic** (e.g. prolog): write programs in logic
  - **Functional** (e.g. ML)
  - **Object-oriented** (e.g. Java): designed to be full portable and compiled into "bytecode" that can be run on any Java Virtual Machine
  - **Scripting** (e.g. MS-DOS or python): compiled and interpreted, no declarations, simple scoping rules

3 Abstract Machines

- A computer is composed of at least:
    - Processor (**CPU**): which executes the machine instructions and to do this it can move data from and into memory;
    - Main memory (**RAM**): which stores data and programs (sequence of machine instructions). It's fast, but volatile;
    - **Mass storage** device: slower then RAM, but persistent;
    - Peripherals for I/O;
- **Architecture**:
    - The different components of a computer (one or more CPUs, RAM, I/O devices, etc.) are connected by a **bus** (e.g. system bus) which are composed of a series of electrical connection;
    - Used to **transmit machine instructions** and **data** between the CPU and RAM or for input and output of data from mass storage devices;
    - In the Von Neumann architecture (Figure 1) memory contains both data and instructions and the bus connects the CPU to the memory and I/O devices;

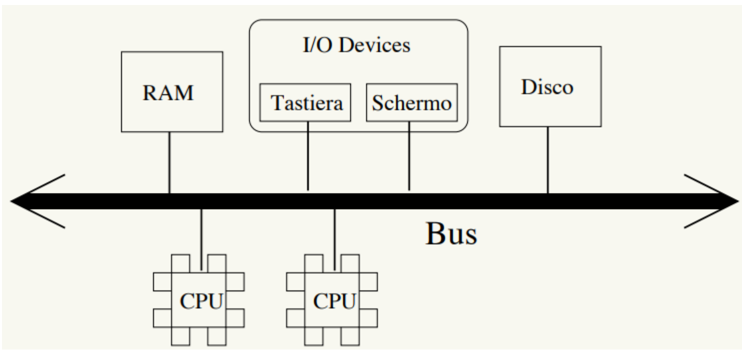

Figure 1: Von Neumann architecture

- **Processor**:
  - Obtain the machine instructions from the memory and executes them;
  - Principal components of a CPU are:
    - **Control unit** that obtains and executes instructions;
    - **Arithmetic Logic Unit** (**ALU**) that executes arithmetic and logic instructions;
    - **Registers** that may be invisible or visible to the programmer:
      * **Invisible** (not accessed directly by machine instructions): **Address Register** (AR) that holds address to access the bus and the **Data Register** (DR) that holds data to read or write;
      * **Visible** (mentioned by machine instructions): **Program Counter** (PC) that holds the address of the next machine instruction to execute (also called the Instruction Pointer (IP)) and the **Status Register** (SR) that holds flags describing the result of an operation of the ALU and the state of the machine (also called F, for Flag register);

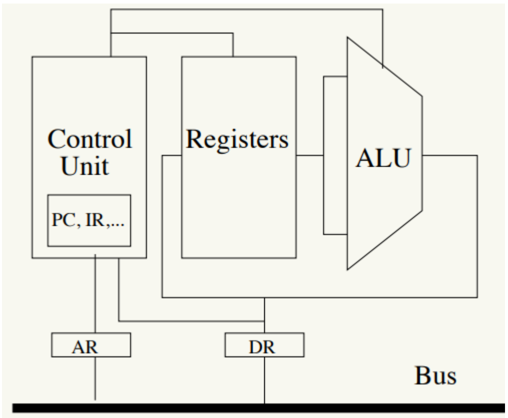

Figure 2: Example of a CPU

- **Execution of instructions** (3 main phases):
  - **Fetch** (read the next instruction to be executed):
    1. Copy the program counter (PC) into the address register (AR);
    2. Transfer the data (addressed in the AR) from RAM to the Data Register (DR);
    3. Save the DR in an invisible register;
    4. Increment the PC;
  - **Decode** the instruction;
  - **Execute** the decoded instruction: it may load data from memory, save data in memory or modify the PC (jump instructions);

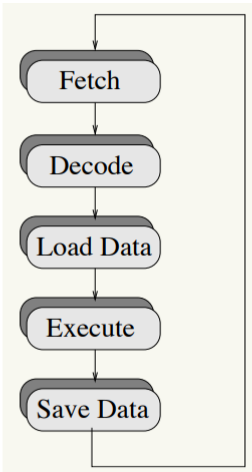

Figure 3: Execution of a program

- **Main memory**:
  - Von Neumann model: the same memory may contain both data and instructions, and can be accessed via the bus;
  - Set of cells, or locations, each **8 bits long** (modern computers have more);
  - **Access to memory**:
    - Load the address to be accessed in the AR register;
    - In case of write operation, load the data to be written in the DR register;
    - Signal, via the bus, which operation (read/write) to execute;
    - In case of read, the data that is read will now be in the DR;

4 Execution of a program

4.1 Physical machine

A **physical machine** is designed for execution of programs and each machine executes programs written in its **own language**.
The execution of a program is an (infinite) cycle of fetch/decode/load/execute/save, where the CPU implement this cycle in hardware and an (interpreted) algorithm can understand and execute its machine language.

4.2 Abstract machine

In **abstract machine** the algorithm that executes a program does not necessarily have to be implemented in hardware, but in abstract machine is implemented in software a collection of algorithms and data structures that enable us to store and execute programs.
Similar to a physical machine (CPU), each abstract machine is associated to a language (e.g. $\mathcal{M}_{\mathcal{L}}$ is an abstract machine that can understand and execute the language $\mathcal{L}$)

4.2.1 Operation of an abstract machine

- To execute a program written in $\mathcal{L}$, $\mathcal{M}_{\mathcal{L}}$ must:
  1. **Execute** elementary operations (the ALU in hardware);
  2. **Control** the sequence of execution: **non-sequential** (e.g. jumps and cycles) and in hardware control the PC;
  3. **Transfer** data from/to memory;
  4. Memory organization: dynamic allocation, stack management, memory of data and programs, etc.;

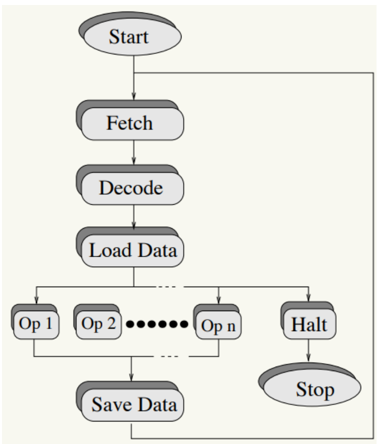

Figure 4: Example of an abstract machine

5 Implementation of a language

$\mathcal{L}$ can be understood and executed by many different abstract machines, these can differ in their implementation, data structures, etc. Implementation of a language $\mathcal{L}$ means to realize an abstract machine $\mathcal{M}_{\mathcal{L}}$ that can execute programs written in this language (implementation can be via hardware (CPU), software or firmware).

5.1 Implementation in software

$\mathcal{M}_{\mathcal{L}}$ is implemented in software and it runs on a Host Machine $\mathcal{MO}_{\mathcal{LO}}$ with machine language $\mathcal{LO}$. There are two types of implementation:

- **Interpretive** (Figure 5): a program written in $\mathcal{LO}$ that understands and executes the language $\mathcal{L}$ (implements the cycle fetch/decode/load/execute/save);
- **Compiler** (Figure 6): a program that can translate other programs from $\mathcal{L}$ to $\mathcal{LO}$;

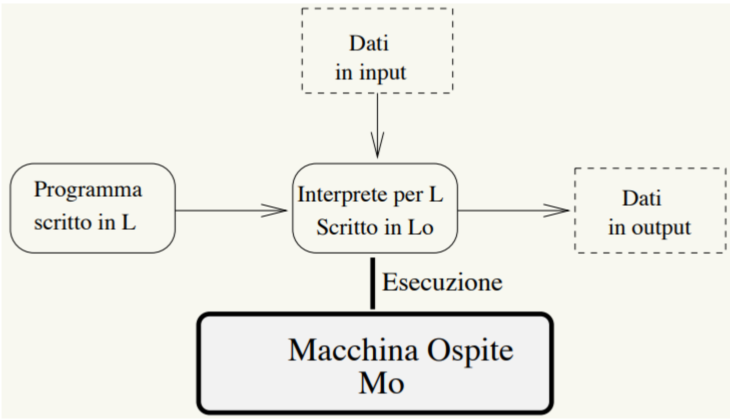

Figure 5: Interpretive implementation

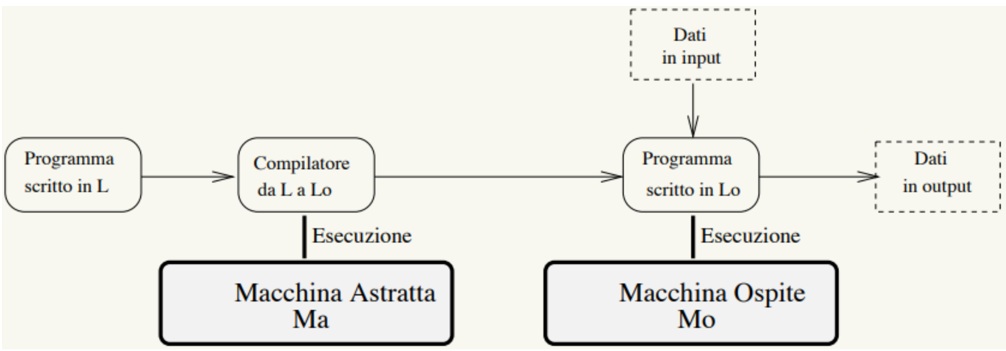

Figure 6: Compiler implementation

5.1.1 Hybrid implementation

Hybrid implementation (Figure 7) is neither purely compiled nor purely interpreted. The compiler translates the program into an **intermediate language** $\mathcal{LI}$ and then this is interpreted by a $\mathcal{MO}_{\mathcal{LO}}$ program written in $\mathcal{LI}$.
Depending on the differences and similarities between $\mathcal{LI}$ and $\mathcal{LO}$ we talk about an implementation mainly compiled or interpreted. For example **Java** $\mathcal{LI}$ (bytecode) is very different from the machine language so the implementation is mainly interpretive (some JVM are mainly compiled), on the other hand C $\mathcal{LI}$ is more or less the machine language, so the implementation is mainly compiled.

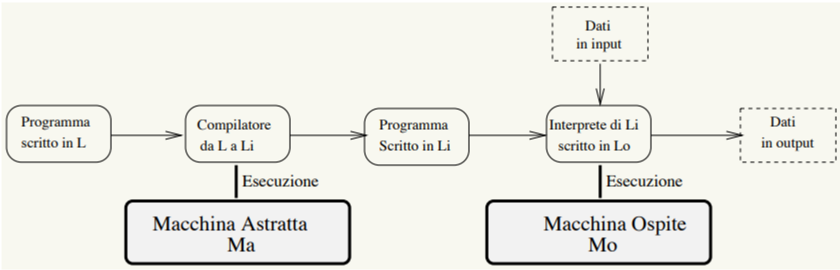

Figure 7: Hybrid implementation

5.2 Implementation in firmware

In **firmware** implementation the abstract machine that executes the machine language of the CPU is not implemented in hardware, but is implemented as a **micro-interpreter**. The program execution cycle is implemented using **micro-instructions** invisible to normal user. The data structures and algorithms of the abstract machine are realized ad micro-programs. For these this implementation is **more flexible** than pure hardware implementation.

6 Names and Elements

6.1 Names

Names are sequence of characters used to **denote something** else (e.g. const p = 3.14, object denote the constant 3.14). In programming languages names are often identifiers (alpha-numerical) and serves to indicate the object denoted.

Object can be associated with a name defined **by the user** (e.g. variables, procedures, labels, etc.) or defined **by the language** (e.g. primitive types, primitive operations, etc.).

The term *Binding* means the association between name and object. Bindings can be made in different times:

- **Language design**: + or int types;
- **Program writing**: for example the binding of an identifier to a variable is defined in the program but is only created when the space is allocated in memory;
- **Compile time**: the compiler allocates memory space for statically data structures (such as global variables). The connection between an identifier and the corresponding memory location is formed at this time;
- **Run-time**: all the association that have not previously been created must be formed at run-time (e.g. local variables);

6.2 Environments

The **environment** is a collections of associations between names and objects that exist at run-time at a specific point in a program and at a specific moment in the execution. An example of declaration:

```
int x;
int f(){
    return 0;
}
type T = int;
```

The same name can denote different object at different points in the program. In fact in modern programming languages, the environment is **structured** (blocks).

A block is a section of the program, identified by opening and closing signs (e.g. {}, ()). A local declaration in a block is visible in this block, and in all nested blocks, as long as these do not contain another declaration of the same name.