

# Paradigmi di programmazione: il paradigma imperativo

## 1 Paradigmi di programmazione

Nell'ambito della filosofia della scienza, il termine *paradigma* indica “un insieme di teorie, standard e metodi che rappresentano un modo di organizzare la conoscenza, cioè un modo di guardare il mondo” (T. Kuhn, *La struttura delle rivoluzioni scientifiche*, 1970).

Analogamente, un **paradigma di programmazione** fornisce un metodo per concettualizzare il processo di computazione e per organizzare e strutturare i compiti che il calcolatore deve svolgere. I principali paradigmi di programmazione sono:

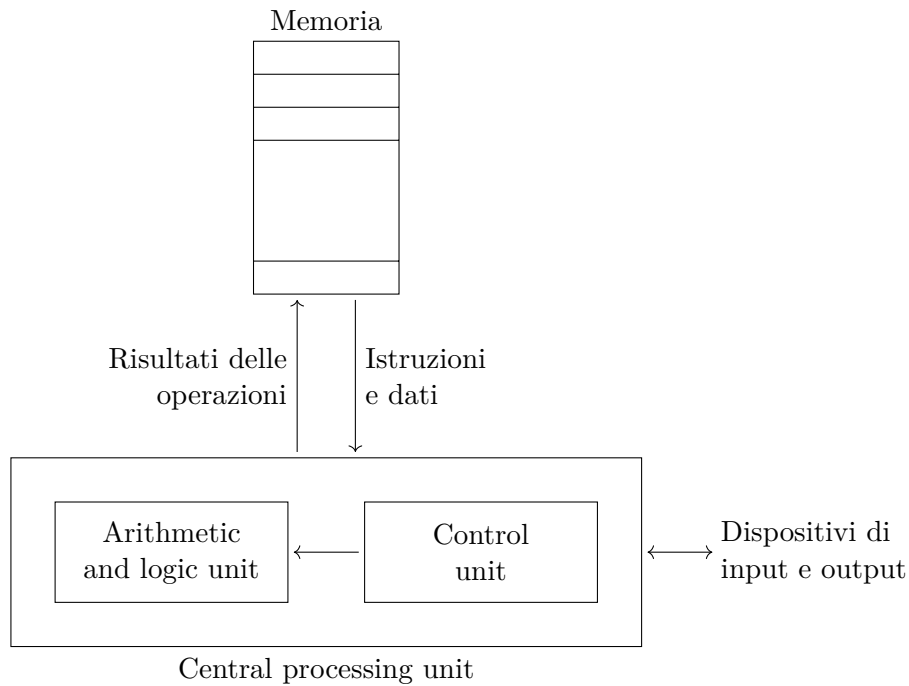
- **imperativo** (linguaggi Pascal, C, ecc.);
- **funzionale** (LISP, FP, ...);
- **logico** (Prolog).

L'**object-oriented programming** (OOP) è spesso considerata un paradigma, ma è più corretto definirla come un concetto ortogonale ai principali paradigmi, in quanto può essere combinata con ciascuno di essi: nel caso dei linguaggi ad oggetti più usati (Java, C++, ecc.) si tratta di un'estensione della programmazione imperativa, mentre ad esempio il linguaggio Scala combina l'OOP con la programmazione funzionale.

Il paradigma imperativo è quello finora più usato in ambito industriale, quindi è utile presentarne le caratteristiche, al fine di confrontarlo con il paradigma funzionale (che sarà l'oggetto di questo corso).

## 2 Macchina di von Neumann

Il paradigma imperativo si basa su elementi derivati dalla struttura dei calcolatori. La struttura concettuale dei calcolatori moderni è la **macchina di von Neumann** (che viene arricchita in vari modi, ma rimane sostanzialmente equivalente). Essa fu introdotta da von Neumann nel 1946, come proposta di un dispositivo realizzabile fisicamente che fosse equivalente alla *macchina di Turing universale*, un modello matematico in grado di calcolare tutto ciò che è calcolabile.



La macchina di von Neumann ha quattro principali componenti.

- La **memoria** contiene sia i dati che le istruzioni.
- La **Central Processing Unit (CPU)** è suddivisa in:
  - **Arithmetic and Logic Unit (ALU)**, che implementa le operazioni di base sui numeri interi;<sup>1</sup>
  - **Control Unit**, che gestisce l'esecuzione delle istruzioni, che consiste in un **ciclo di esecuzione** a tre fasi:
    1. **fetch**: preleva dalla memoria la prossima istruzione da eseguire;
    2. **decode**: interpreta l'istruzione, riconoscendone il significato;
    3. **execute**: esegue le operazioni elementari corrispondenti all'istruzione.
- Dei **dispositivi di input e output** permettono l'interazione con l'utente.

---

<sup>1</sup>Tramite opportune codifiche, i numeri interi possono essere utilizzati per rappresentare dati di qualsiasi tipo, quindi un modello di calcolo teorico può limitarsi a trattare i numeri interi.

## 3 Linguaggio macchina e assembler

Ogni processore ha un proprio **linguaggio macchina**, che comprende un'istruzione per ciascun'operazione implementata nell'hardware. Poiché la tecnologia usata per implementare i calcolatori si presta naturalmente alla rappresentazione di valori binari, le istruzioni sono sequenze di bit che codificano l'operazione da eseguire e gli operandi su cui eseguirla (che possono essere costanti, registri, locazioni di memoria, ecc.). Il formato esatto delle istruzioni dipende dal processore.

Il linguaggio **assembler** di un processore è la versione simbolica del linguaggio macchina, nella quale le istruzioni sono rappresentate da nomi (simboli) significativi anziché da sequenze di bit.

### 3.1 Esempio di assembler

A scopo illustrativo, si consideri un assembler “giocattolo” con le seguenti caratteristiche:

- Le *celle di memoria* sono individuate da numeri interi e contengono numeri interi.
- I *registri del processore* sono indicati con  $R1, R2, \dots$  e contengono numeri interi.
- Le *istruzioni di trasferimento* sono LOAD e STORE:
  - LOAD  $R, X$  trasferisce il contenuto della cella  $X$  nel registro  $R$ ;
  - STORE  $R, X$  trasferisce il contenuto del registro  $R$  nella cella  $X$ .
- Le *istruzioni aritmetico/logiche* sono ADD, SUB, MUL, DIV, LT, GT ed EQ. Ad esempio:
  - ADD  $R1, R2$  calcola la somma tra i valori contenuti in  $R1$  e  $R2$ , e scrive il risultato in  $R1$ ;
  - LT  $R1, R2, R3$  scrive in  $R1$  il valore 1 se  $R2 < R3$ , altrimenti scrive in  $R1$  il valore 0.
- Le *istruzioni di salto*, che gestiscono il flusso di esecuzione del programma, sono JUMP e JZERO:
  - JUMP  $\text{alfa}$  è un salto incondizionato all'istruzione con etichetta  $\text{alfa}$ ;  
    JUMP  $\text{alfa}$   
    ...  
     $\text{alfa}: \dots$
  - JZERO  $R1, \text{alfa}$  è un salto condizionato all'istruzione con etichetta  $\text{alfa}$ , che viene eseguito se  $R1$  contiene il valore 0, altrimenti si prosegue a eseguire l'istruzione successiva a JZERO.

```

        JZERO R1, alfa
        ...
    alfa: ...

```

Questo linguaggio può essere usato, ad esempio, per implementare l'algoritmo di Euclide, che calcola il massimo comune divisore (MCD) di due numeri  $x$  e  $y$ . A parole, quest'algoritmo può essere descritto come la seguente sequenza di operazioni elementari:

1. Si calcola il *resto* della divisione di  $x$  per  $y$ .
2. Se *resto*  $\neq 0$ , si ricomincia dal passo 1 con  $x \leftarrow y$  e  $y \leftarrow \text{resto}$ , altrimenti si prosegue con il passo successivo.
3. L'MCD è il valore di  $y$ .

Una possibile implementazione nel linguaggio assembler appena presentato è questa (qui si indicano con // i commenti, usati per chiarire il significato del codice):

```

// input: la cella 101 contiene x e la cella 102 contiene y
    LOAD  R1, 101    // R1 <- x
    LOAD  R2, 102    // R2 <- y

    // calcola x % y
alfa: DIV  R1, R2    // R1 <- x / y
    MUL   R1, R2    // R1 <- (x / y) * y
    LOAD  R2, 101    // R2 <- x
    SUB   R2, R1    // R2 <- x - ((x / y) * y) (resto)

    JZERO R2, fine  // se resto = 0 vai a fine
    // altrimenti (resto != 0) continua
    LOAD  R1, 102    // R1 <- y
    STORE R1, 101    // x <- y
    STORE R2, 102    // y <- resto
    JUMP  alfa

fine: LOAD  R1, 102    // R1 <- y (MCD)
    STORE R1, 103    // 103 <- MCD
// output: la cella 103 contiene l'MCD

```

### 3.2 Svantaggi

Come si può in parte intuire dall'esempio precedente, la programmazione in linguaggio macchina o assembler ha diversi svantaggi:

- è necessario conoscere il linguaggio e i dettagli dell'architettura dello specifico processore utilizzato;

- è impossibile trasportare i programmi da una macchina a una diversa; e correggerlo in presenza di errori;
- la struttura logica del programma è nascosta, quindi risulta difficile comprendere il programma
- il programmatore si specializza nell'uso di “trucchi” legati alle caratteristiche specifiche della macchina, rendendo ancora più difficili da comprendere e modificare i programmi.

## 4 Linguaggi ad alto livello

L'obiettivo principale dei **linguaggi ad alto livello** è proprio evitare gli svantaggi del linguaggio macchina — in particolare, rendere la programmazione indipendente dalle caratteristiche peculiari della macchina utilizzata. Essi aggiungono un *livello di astrazione* rispetto all'architettura della macchina, introducendo in particolare due tipi di astrazioni: *astrazioni sui dati* e *astrazioni sulle istruzioni*.

### 4.1 Semantica e macchine astratte

La **semantica**, cioè il significato, delle istruzioni di un linguaggio viene spesso descritta sotto forma di **semantica operativa**, cioè in termini delle operazioni elementari corrispondenti a ciascuna istruzione.

La semantica operativa del linguaggio assembler specifica il significato di ciascun'istruzione in termini di operazioni del processore, indicando ad esempio le modifiche del contenuto della memoria (compresi i registri) che l'istruzione provoca.

La semantica operativa di un linguaggio ad alto livello potrebbe essere descritta associando a ciascun'istruzione del linguaggio una sequenza di istruzioni assembler della macchina sottostante, ma ciò sarebbe contrario all'obiettivo di indipendenza dall'architettura, dunque si fa invece riferimento a una **macchina astratta**, in grado di effettuare operazioni più ad alto livello rispetto alle operazioni elementari dei processori reali. Il programmatore può allora scrivere i programmi in linguaggio ad alto livello facendo riferimento a tale macchina astratta, senza pensare alle operazioni del processore.

### 4.2 Variabili e assegnamento

Uno dei principali concetti presenti nei linguaggi (imperativi) ad alto livello è quello di **variabile**, che è un contenitore preposto a contenere dei valori, ovvero un'astrazione del concetto di locazione di memoria. Allora, l'**assegnamento** di un valore a una variabile è un'astrazione dell'operazione STORE, e il **deferenziamento** di una variabile è un'astrazione dell'operazione LOAD.

L'istruzione di assegnamento può essere indicata come<sup>2</sup>

$$\text{variabile} \leftarrow \text{espressione}$$

e la sua semantica operativa (rispetto alla macchina astratta) è la seguente:

1. viene calcolato il valore dell'*espressione*;
2. il risultato ottenuto viene assegnato alla *variabile*.

Ad esempio, per eseguire l'istruzione  $x \leftarrow y + z$ :

1. si valuta l'espressione  $y + z$ , recuperando i valori contenuti in  $y$  e  $z$  (cioè dereferenziando  $y$  e  $z$ ) e facendone la somma;
2. si pone il risultato nella variabile  $x$ .

Dalla descrizione della semantica dell'assegnamento si può individuare uno degli svantaggi dei linguaggi ad alto livello: essi hanno operazioni che sono inevitabilmente più complesse rispetto ai linguaggi a basso livello, quindi la descrizione della semantica risulta più difficile, e richiede l'introduzione di un nuovo lessico: ad esempio, bisogna definire cosa significhi "calcolare il valore di un'espressione". Allora, bisogna prestare attenzione a comunicare in modo chiaro la semantica di un linguaggio ad alto livello, poiché conoscerla bene è indispensabile per scrivere programmi corretti.

### 4.3 Tipi

Nella memoria, tutte le variabili sono rappresentate come sequenze di bit, che possono essere interpretate in modi diversi: ad esempio, la sequenza 01000001 potrebbe essere interpretata come il numero intero 10 o come il carattere 'A'.

La nozione di **tipo** fornisce un'astrazione rispetto alla rappresentazione dei dati, permettendo al programmatore di usare variabili di vari tipi senza bisogno di conoscerne l'effettiva rappresentazione.

---

<sup>2</sup>L'uso di una freccia (invece che dell'uguale) per rappresentare l'assegnamento rende più esplicito il flusso delle informazioni.