# Linguaggi di Programmazione

UniShare

Davide Cozzi
@dlcgold

Gabriele De Rosa
@derogab

Federica Di Lauro
@f_dila

# Indice

# Capitolo 1

# Introduzione

**Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link:** `https://github.com/dlcgold/Appunti`.
**Grazie mille e buono studio!**

# Capitolo 2

# I linguaggi

Si possono classificare in 3 gruppi i linguaggi di programmazione:

1. **Linguaggi imperativi**, come *C, Assembler, Python etc....* Le caratteristiche dei linguaggi imperativi sono legate all'architettura di Von Neumann, composta da una componente passiva (la memoria) e una attiva (il processore). Il processore esegue calcoli e assegna valori a varie celle di memoria. Si ha quindi il concetto di *astrazione*. Una variabile non è altro che un'astrazione di una cella di memoria fisica. Ogni linguaggio ha diversi livelli di astrazione dell'archiettura di Von Neumann (che ricordiamo usare il "ciclo" formato da *Fetch instruction, Execute e Store result*), con i cosiddetti linguaggi di *alto* e *basso* livello. Possono essere sia linguaggi compilati (come *C*) che interpretati (come *Python*).
   I linguaggi imperativi usano quindi il *Paradigma Imperativo*, detto anche *Procedurale*. In questo paradigma si adotta uno *stile prescrittivo*, si prescrivono infatti operazioni che il processore deve eseguire e le istruzioni vengono eseguite in ordine, al più di strutture di controllo, e per questo è il miglior paradigma per rappresentare gli algoritmi. Questi linguaggi sono tra i più vecchi e tutt'ora tra i più usati soprattutto per la manipolazione numerica. Si ha la seguente formula che ben descrive il paradigma imperativo:

   $$Programma = Algoritmi + Strutture\ Dati$$

   In un linguaggio imperativo si ha sia una parte dedicata alla dichiarazione di variabili che una parte dedicata agli algoritmi risolutivi del problema. Inoltre le istruzioni possono essere così divise:

   - istruzioni di I/O

- istruzioni di assegnamento
- istruzioni di controllo

La ricerca di gestire applicazioni ancora a più alto livello con codice più conciso e semplice, che affrontano i problemi in maniera più logica (o comunque in maniera differente) ha portato alla nascita di altri paradigmi. Linguaggi logici e funzionali sono accomunati dall'essere di altissimo livello, dall'essere generati per manipolazione simbolica e non numerica, dal non distinguere perfettamente programma e strutture dati, dall'essere basati su concetti matematici e sull'adottare uno *stile dichiarativo*

2. **Linguaggi a oggetti**, come *C++* utilizzano il paradigma ad oggetti con l'uso di classi etc. Non vengono affrontati nel corso.

3. **Linguaggi Logici**, come il *Prolog*. Si basano sul concetto della deduzione logica e hanno come base la logica formale e come obbiettivo la formalizzazione del ragionamento. Programmare con un linguaggio logico significa descrivere un problema con frasi del linguaggio (ovvero con formule logiche) e interrogare il sistema che effettua deduzioni sulla base della conoscenza rappresentata. Il paradigma logico si può rappresentare con la seguente formula:

$$Programma = Conoscenza + Controllo$$

Si ha uno *stile dichiarativo* in quanto la conoscenza del problema è espressa indipendentemente dal suo utilizzo (si usa il **cosa** e non il **come**). Si ha quindi un'alta modularità e flessibilità ma si ha la problematica della rappresentazione della conoscenza, infatti definire un linguaggio logico significa definire come il programmatore può esprimere la conoscenza e quale tipo di controllo si può utilizzare nel processo di deduzione.
Analizziamo le basi del Prolog:

- dopo ogni asserzione si mette un . mentre le **,** sono degli *and* logici
- le costanti si indicano in minuscolo e le variabili in maiuscolo
- **Asserzioni Incondizionate (fatti)** così indicate:

  ```
  A.
  ```

- **Asserzioni Condizionate (regole)**, che ricordiamo non essere regole di inferenza, così indicate:

```
A :-B, C, D, ..., Z.
```

dove A è il *conseguente*, ovvero la conclusione, mentre le altre sono gli antecedenti, ovvero le premesse. Il simbolo ":-" è un implica che per ragioni di interprete si legge al contrario rispetto al solito: seguendo l'esempio si ha che *B, C, D, ..., Z implicano A* ovvero $B, C, D, ..., Z \rightarrow A$

4. **Interrogazione**, che rappresenta l'input utente, è così espressa:

```
:-K, L, M, ..., P.
```

e indica che si chiede cosa implicano quei dati antecedenti.

Vediamo un esempio più completo (anche se non del tutto):
*due individui sono colleghi se lavorano per la stessa ditta:*

```
collega(X, Y) :-
     lavora(X, Z),
     lavora(Y, Z),
     diverso(X, Y).

lavora(ciro, ibm).
lavora(ugo, ibm).
lavora(olivia, samsung).
lavora(ernesto, olivetti).
lavora(enrica, samsung).

:-collega(X, Y).
```

dove la prima asserzione rappresenta la regola, le successive 5 i fatti e l'ultima riga è l'interrogazione. Il programma non è completo in quanto non si definisce concretamente *diverso*. La logica di risoluzione è la seguente: L'interprete cerca un X e un Y (che sono variabili) in grado di rappresentare quella regola, infatti l'interrogazione è la conseguenza, e li cerca tra i fatti partendo dal primo ("ciro, ibm") che viene messo come X = ciro e Z = ibm. Parte il confronto con se stesso (si ha tanto la funzione *diverso*) e con gli altri (che mano a mano diventeranno gli Y e Z del secondo lavora) dando alla fine come risultato solo i colleghi cercati.

5. **Linguaggi funzionali:**, come *Lisp* i suoi "dialetti" come *Common Lisp*, hanno come concetto primitivo la *funzione* che è una regola di associazione tra due insiemi (dominio e codominio). La regola di

una funzione ne specifica dominio, codominio e regola di associazione. Una funzione può essere applicata ad un elemento del dominio (detto *argomento*) per restituire l'elemento del codominio associato (mediante il processo di *valutazione* o *esecuzione*). Nel paradigma funzionale puro l'unica applicazione è l'applicazione di funzioni e il ruolo dell'esecutore si esaurisce nel valutare l'applicazione di una funzione e produrre un valore. In questo paradigma "puro" il valore di una funzione è determinato soltanto dal valore degli argomenti che riceve al momento della sua applicazione, e non dallo stato del sistema rappresentato dall'insieme complessivo dei valori associati a variabili (e/o locazioni di memoria) in quel momento, comportando l'assenza di effetti collaterali. Il concetto di variabile è qui quello di *costante matematica* con valori immutabili (non si ha l'operazione di assegnamento). La programmazione funzionale consiste nel combinare funzioni mediante composizioni e utilizzare la **ricorsione**. Il paradigma è ben rappresentato da questa formula:

$$Programma = Composizione\ di\ Funzioni + Ricorsione$$

Si ha quindi un insieme di funzioni mutualmente ricorsive e l'esecuzione del programma consiste nella valutazione dell'applicazione di una funzione principale a degli argomenti.
Il linguaggio *Lisp*, inizialmente proposto da John McCarthy nel '58 era un linguaggio funzionale puro. Si sono poi sviluppati molti ambienti di programmazione Lisp come: *Common Lisp, Scheme e Emacs Lisp.*
Si analizza un esempio di codice in *Lisp*: *Controllare se un elemento (item) appartiene ad un insieme (rappresentato con una lista);*

```
(defun member (item list)
        (cond((null list) nil)
                ((equal item (first list)) T)
                (T (member item (rest list)))))
(member 42 (list 12 34 42))
```

Si ha che tutto è rappresentato da una lista, si hanno delle funzioni standard (*defun, equal, first, rest e list*) e *member* che viene definita dal programmatore. L'ultima linea definisce il numero da cercare e la lista, sempre con la logica di funzioni dentro ad altre. L'esecuzione è la seguente: Si definisce, nella prima riga, la funzione che cerca un valore (*item*) in una lista (*list*). Nella seconda riga cominciano le condizioni:

(a) prima si controlla se la lista è nulla con *nil* (che sarebbe falso) si ha il risultato di questo controllo. Questo rappresenta anche il caso base della nostra funzione ricorsiva.

(b) si controlla se l'elemento cercato è uguale al primo della lista e con $T$ si indica *true*

(c) con l'ultima parte si ha la vera e propria ricorsione, forzata da $T$ iniziale, che ripete l'operazione togliendo ogni volta il primo elemento, facendo ricominciare i controlli con l'elemento successivo finché non si trova o non si ha il caso base della lista vuota

Si nota l'assenza di assegnamenti.