Linguaggi di Programmazione 2022-2023
Lisp e programmazione funzionale I

Marco Antoniotti
Gabriella Pasi
Fabio Sartori

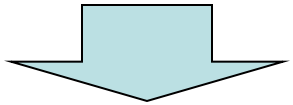
Diversi Paradigmi di Programmazione

- *Imperativo*
  I programmi computano modificando lo "stato" della "memoria" del sistema
  - Esempi di linguaggi imperativi puri: C, Fortran, Ada, Assembly
  - Linguaggi con componente imperativa: C++, Java, Python
- *Object Oriented*
  I programmi computano mantenendo lo "stato" della "memoria" in "oggetti" che rispondono in modo particolare alle richieste che vengono fatte loro (tramite chiamate a metodi)
  - Esempi tipici: Java, Smalltalk, C++ (ma anche Ada, Common Lisp, Ocaml, Python)
- *Funzionale*
  I programmi computano combinando "valori" (rappresentati in memoria) trasformati da chiamate a funzioni
  - Esempi tipici LISP, ML, Haskell, FP
- *Dichiarativo*
  I programmi "computano" verificando che varie asserzioni su un sistema sono vere o false; il processo di verifica sostituisce determinati valori a variabili presenti nelle asserzioni
  - Prolog

I linguaggi imperativi

Basati sull'architettura di Von Neumann:

- Memoria costituita da celle identificate da un indirizzo,contenenti dati o istruzioni
- Unità di controllo e aritmetica
- Programma immagazzinato nella memoria centrale



– *variabili* = celle, *nome* = indirizzo
– azione unitaria: *istruzione*
– programma: combinazione di istruzioni

- Concetto di assegnamento di valori a variabili
  ogni valore calcolato deve essere esplicitamente memorizzato, cioè assegnato ad una cella
- Concetto di sequenza di istruzioni
  ogni programma consiste nell'esecuzione di una sequenza di istruzioni con possibili "salti"

Effetti collaterali

- Gli effetti collaterali sono una delle caratteristiche negative dei linguaggi imperativi
- Modifiche dello stato di memoria di un programma
  - Ad esempio tramite modifica di locazioni di memoria accessibili da variabili passate per riferimento o globali
  - Tipico uso: comunicazione tra diverse parti di un sistema

- Problemi
  - Distinguere tra effetti collaterali desiderati e non desiderati
  - Leggibilità del programma (l'istruzione di chiamata non rivela quali variabili globali sono coinvolte)
  - Risulta difficile verificare la correttezza di un programma

Effetti collaterali

- Consideriamo il seguente esempio

```
w := x + f(x, y) + z
```

- La funzione `f()` può modificare `x` e `y` se sono passati per indirizzo, e `z` se è globale
  - si riduce la leggibilità del programma
  - non ci si può fidare della commutatività dell'addizione
- Altro caso

```
u := x + z + f(x,y) + f(x,y) + x + z
```

- gli stessi problemi dell'esempio precedente...
- ... più l'impossibilità per il compilatore di generare codice ottimizzato valutando una sola volta `x + z` e `f(x, y)`

Trasparenza referenziale

- Un concetto matematico fondamentale è quello di trasparenza referenziale

  il significato del tutto si può determinare dal significato delle parti

- Questa proprietà è valida per espressioni aritmetiche e matematiche, in particolare questo concetto rende possibile la sostituzioni di espressioni con altre a patto che esse denotino gli stessi valori

- Esempio:
  nell'espressione matematica `f(x) + g(x)` si può sostituire la funzione `f` con la funzione `h` se producono valori identici

- Nei linguaggi imperativi tradizionali non si può essere certi della sostituzione, né che `f(x) + g(x) = g(x) + f(x)`, o che
  `f(x) + f(x) = 2 * f(x)`

- ***I linguaggi funzionali (puri) hanno il concetto di trasparenza referenziale come fondamento***

Linguaggi e programmazione *funzionali*

- L'idea fondamentale dei linguaggi (e della programmazione) funzionale è il seguente

  programmi = funzioni matematiche

- Ovvero, un programma è costituito dalla combinazione di varie funzioni
  - Funzioni **primitive**
  - Funzioni più complesse ottenute via composizione

- La trasparenza referenziale propria della matematica viene mantenuta

Linguaggi e programmazione *funzionali*

- Come ci si può aspettare, programmare in un linguaggio funzionale richiede la manipolazione di funzioni
- La notazione normale per denotare funzioni è la seguente

$$\boldsymbol{f}(x_1, x_2, \ldots, x_N)$$

ad esempio

$$\boldsymbol{cos}(3.14)$$

oppure

$$\boldsymbol{substring}(\text{“la vita l’e` bela!”}, 3, 7)$$

- Ogni funzione denota un valore ottenuto tramite una mappa a partire dagli “argomenti”

Linguaggi e programmazione *funzionali*

- Nel paradigma funzionale vi sono oggetti di vario tipo e strutture di controllo, ma vengono raggruppati logicamente in modo diverso da come invece accade nel paradigma imperativo. In particolare è utile pensare in termini di

- Espressioni (funzioni primitive e non)
- Modi di combinare tali espressioni per ottenerne di più complesse (composizione)
- Modi e metodi di costruzione di “astrazioni” per poter far riferimento a gruppi di espressioni per “nome” e per trattarle come unità separate
- Operatori speciali (condizionali ed altri ancora, che verranno introdotti in seguito)

Linguaggi e programmazione *funzionali*

- Definizione di funzione: regola per associare gli elementi di un insieme (*dominio*, *domain*) a quelli di un altro insieme (*codominio*, *range*)
- Una funzione può essere applicata a un elemento del dominio (si noti che dominio e codominio possono essere il prodotto cartesiano di insiemi più semplici)
- L'applicazione produce (restituisce) un elemento del codominio detto valore
- *Esempio*

```
quadrato(x) ≡ x * x
```

dove `x` è un numero, detto **argomento**, che indica un generico elemento del dominio (ad esempio, l'insieme dei numeri reali **R**)

  - `x` è una variabile matematica, senza una precisa locazione in memoria: non ha senso pensare di modificarla!
  - In un'implementazione di un linguaggio funzionale, `x` sarà mappata in particolari locazioni di memoria, ma non ci è dato di modificarne il contenuto

Linguaggi e programmazione funzionali: composizione

- Nei linguaggi funzionali espressioni più complesse vengono costruite mediante composizione

- Se F è definita come composizione di G e H:

$$F \equiv G \circ H$$

applicare F equivale ad applicare G al risultato dell'applicazione di H

- Esempio:

$$alla\_quarta \equiv quadrato \circ quadrato$$

quindi:

$$alla\_quarta(5) = (quadrato \circ quadrato)(5) = quadrato(quadrato(5))$$

Linguaggi e programmazione funzionali: ricorsione e operatori speciali

- Finora siete stati istruiti ad utilizzare dei linguaggi più o meno imperativi. In particolare avete visto come inserire semplici dati in un programma e di stampare dei risultati tramite quello che viene definito il sistema (o libreria) di input/output di un linguaggio. Tra questi due momenti avete visto una serie di costrutti e di oggetti raggruppabili come
- Oggetti (numeri, strutture dati, files, etc. etc.)
- Strutture di controllo
  - if-then-else
  - for
  - while
  - case
  - …
- Assegnamenti
- Procedure, funzioni e metodi
  - Nei linguaggi imperativi le regole di trasformazione dal dominio al codominio corrispondono ai passi da eseguire in un ordine determinato dalle strutture di controllo

Linguaggi e programmazione funzionali: ricorsione e operatori speciali

- Le espressioni matematiche sono invece composte da composizione di funzioni spesso organizzate ricorsivamente e controllate da operatori speciali

- Esempio (non necessariamente in un linguaggio corrente)

```
fattoriale(x) ≡ if (x = 0) then 1 else x * fattoriale(x - 1)
```

  questa definizione di fattoriale utilizza un operatore speciale (i**f-then-else**) per rappresentare valutazioni condizionali

- Nella programmazione funzionale (*pura*) non è possibile produrre effetti collaterali

Linguaggi e programmazione funzionali: riassunto caratteristiche principali

- La principale modalità di calcolo è l'applicazione di funzione
- Il calcolo procede valutando espressioni, senza effetti collaterali (trasparenza referenziale)
- Le funzioni sono oggetti di prima classe (non solo "puntatori a funzione")
  - Possono essere parte di una struttura dati
  - Possono essere costruite durante l'esecuzione di un programma e ritornate come valore di un'altra funzione
- I linguaggi funzionali consentono l'uso di funzioni di ordine superiore (*), cioè funzioni che prendono altre funzioni come argomenti e che possono restituirne come valore, in modo assolutamente generale
- Nei linguaggi funzionali puri non esistono strutture di controllo iterative come **while** e **for**; questi sostituiti da ricorsione combinata con gli operatori speciali condizionali

(*) Nella STL C++ ed in Java vi sono meccanismi per approssimare questi comportamenti

LISP ed il paradigma funzionale

- LISP non è propriamente un linguaggio, ma una famiglia di linguaggi, il cui primo esponente risale alla fine degli anni 50 (McCarthy)
  - L'acronimo sta per *LIS*t *P*rocessing

- A tutt'oggi vi sono due dialetti principali: Common Lisp e Scheme (il linguaggio clojure è un nuovo dialetto molto usato di recente)
- Lo studio del LISP in una delle sue incarnazioni è importante dato che il linguaggio è uno dei più vecchi rappresentanti del paradigma di programmazione *funzionale*
  - Altri linguaggi funzionali sono quelli della famiglia ML, Haskell, FP, F# (Microsoft) e molti altri

- Le versioni minimali di LISP ammettono solo funzioni primitive su **liste**, un operatore speciale per creare funzioni (**lambda**), un operatore condizionale (**cond**) ed un piccolo insieme di predicati ed operatori speciali che vedremo in seguito
- Gli standard e varie implementazioni introducono diversi costrutti "imperativi" per convenienza

Utili consigli

- Potete recuperare varie implementazioni di (Common) Lisp presso http://www.alu.org
  le versioni “personal” di Lispworks, http://www.lispworks.com, e di Franz http://www.franz.com sono consigliate

- I veri programmatori Linux/UNIX possono scaricare CMUCL, SBCL, ECLS etc etc

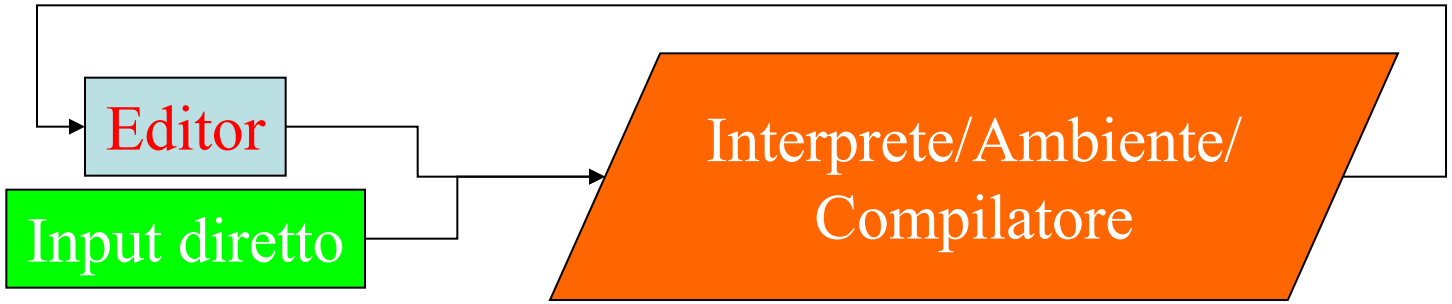
- Potete recuperare varie implementazioni di Scheme presso http://www.schemers.org
  la consigliata è Racket

- I libri principali sono
  - *Structure and Interpretation of Computer Programs* (SICP), Abelson, Sussman e Sussman, http://mitpress.mit.edu/sicp
  - *Practical Common Lisp* (PCL), Seibel, http://www.gigamonkeys.com/book

Programmazione funzionale: interpreti, ambienti e compilatori

In Java, C/C++ e vari linguaggi di programmazione la costruzione di programmi avviene via il processo di “compilazione”

In LISP e vari altri linguaggi di programmazione spesso si interagisce invece con un “*ambiente*” (spesso contenente il *compilatore*) la cui interfaccia principale è un *interprete*



Espressioni in LISP

- Una volta fatto partire l'interprete LISP si può procedere ad una esplorazione delle espressioni di base del linguaggio

- Utilizzeremo il Common Lisp come linguaggio

- Una prima cosa da notare è che in LISP ogni "espressione" denota un "valore". Le espressioni più semplici sono numeri e stringhe. Possiamo vedere che succede inserendo queste espressioni nell'interprete

```
prompt> 42
42

prompt> "Sapete che cos'e` '42'?"
"Sapete che cos'e` '42'?"
```

Programmazione funzionale

- Prima di procedere facciamo la seguente osservazione

  *Le operazioni aritmetiche elementari +, -, * e / non sono altro che funzioni*

  Infatti

  +(40, 2)

  denota il numero 42, così come

  -(82, 40) *(21, 2) /(126, 3)

- Questo tipo di notazione per le operazioni aritmetiche elementari si dice notazione prefissa

Programmazione Funzionale in LISP

- In (Common) Lisp, la sintassi per le “chiamate” o “valutazioni” o “**applicazione**” di funzioni è molto semplice ma relativamente diversa dalla notazione tradizionale. Ogni espressione in (Common) Lisp ha la forma seguente

$$(f\ x_1\ x_2\ \ldots\ x_N)$$

Le parentesi iniziali e finale sono obbligatorie (vedremo più in là che ciò ha conseguenze importanti) e gli *spazi* (almeno uno) sono necessari per separare tra di loro la funzione e gli argomenti

Programmazione funzionale in LISP

- Una volta stabilita questa convenzione, la costruzione di espressioni più complicate a partire da più semplici è una cosa da ragazzi
  Proviamo con l'interprete

```
prompt> (+ 40 2)
42

prompt> (- 84 42)
42

prompt> (* 2 3 7)
42
```

Programmazione funzionale in LISP

- Come si può notare, le funzioni aritmetiche in LISP accettano un numero variabile di argomenti

```
prompt> (+ 2 10 10 20)
42
```

- Le funzioni in LISP si combinano secondo le ovvie norme

```
prompt> (+ 2 (* 2 10) 20)
42
```

Ovvero al posto di un valore, possiamo inserire un'espressione (e.g., una chiamata ad una funzione) che lo denota

Programmazione funzionale in LISP

- Non vi sono ambiguità possibili nell'interpretazione poiché la funzione (anche detta operatore) è sempre il primo elemento dell'espressione
- Le espressioni possono essere complicate quanto si vuole

```
prompt> (+ (* 3 (+ (* 2 4) (+ 3 2))) (+ (- 10 8) 1))
42
```

- Al fine di rendere più leggibili le espressioni più complicate, di solito le si allinea in modo da avere gli argomenti (detti anche operandi) di una chiamata allineati verticalmente

```
prompt> (+ (* 3
              (+ (* 2 4)
                 (+ 3 2)))
           (+ (- 10 8)
              1))
42
```

Programmazione funzionale in LISP

- Notiamo come le funzioni aritmetiche elementari + e * in (Common) LISP rispettano i vincoli di “campo” algebrico

```
prompt> (+)
0

prompt> (*)
1
```

- Le funzioni aritmetiche elementari - e / in (Common) LISP richiedono almeno un argomento e rappresentano in questo caso il “reciproco”, sempre in senso algebrico

```
prompt> (- 42)
-42

prompt> (/ 42)
1/42
```

Funzioni e “costanti” in Common Lisp

- Numeri
  - Interi: `42 -3`
  - Virgola mobile: `0.5 3.1415 6.02E+21`
  - Razionali: `3/2 -3/42`
  - Complessi: `#C(0 1)`

- Booleani (*)
  `T NIL`

- Stringhe
  `”sono una stringa”`

- Operazioni su booleani
  `null and or not`

- Funzioni su numeri
  `+ - / * mod sin cos sqrt tan atan plusp > <= zerop`

Ordine di valutazione

- Data un'espressione LISP

$$(\boldsymbol{f}\ x_1\ x_2\ \ldots\ x_N)$$

la valutazione procede da sinistra verso destra a partire da x_1 fino a x_N producendo I valori v_1, …, v_N.
La funzione ***f*** viene "valutata" successivamente e viene applicata ai valori v_1, …, v_N.

- Questa regola è inerentemente ricorsiva.
- Alcuni operatori speciali valutano gli argomenti in modo diverso (**`if`**, **`cond`**, **`defun`**, **`defparameter`**, **`quote`**, etc)

Definizione di variabili e di funzioni

- In Common Lisp è possibile definire delle variabili usando l'operatore speciale `defparameter`

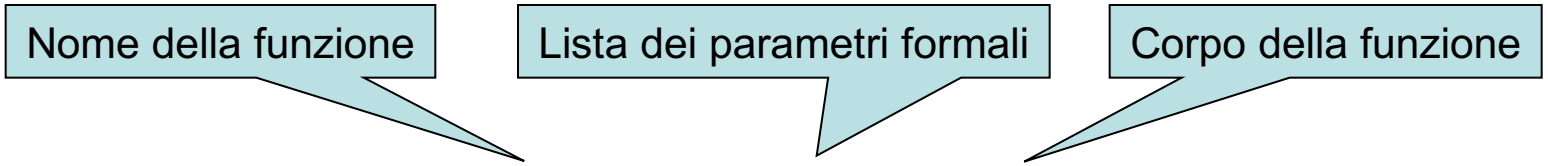
```
prompt> (defparameter quarantadue 42)
quarantadue
```

- Il **simbolo** `quarantadue` ha ora *associato* il valore `42`

- La sintassi è molto semplice: l'operatore **`defparameter`** è seguito da un **simbolo** (un "identificatore" o "nome") e da un'espressione che viene valutata al fine di produrre un valore che viene associato al simbolo

Definizione di variabili e di funzioni

- Le funzioni si possono definire usando l'operatore speciale **`defun`**




```
prompt> (defun quadrato (x) (* x x))
quadrato
```

- La sintassi è molto semplice: l'operatore **`defun`** è seguito dal nome della funzione e da una lista di simboli - ovvero delimitati da parentesi - che rappresentano i nomi dei parametri formali della funzione; infine troviamo un'espressione (o una sequenza di espressioni) che costituisce il "corpo" della funzione
- L'operatore **`defun`** associa il corpo della funzione al nome nell'*ambiente globale* del sistema Common Lisp e restituisce come valore il nome della funzione

Definizione di variabili e di funzioni

- Una volta definita, una funzione viene eseguita (o chiamata) usando la regola descritta precedentemente

```
prompt> (quadrato quarantadue)
1764

prompt> (- (quadrato (+ 20 22)) 10)
1754

prompt> (defun somma-di-quadrati (x y)
           (+ (quadrato x) (quadrato y)))
somma-di-quadrati

prompt> (- (somma-di-quadrati 6 3) 3)
42
```

Sommario

- Una definizione informale di cos'è il **paradigma funzionale**
  - Operazione fondamentale: **applicazione di funzioni mutuamente ricorsive**
- Il concetto di **trasparenza referenziale**
- Introduzione al Lisp
  - Notazione funzionale
  - Dichiarazione di funzioni e variabili
  - Ordine di valutazione