

## Computazioni, Programmazione funzionale e Preludio al $\lambda$ -calcolo

Cosa significa **computare**? In generale possiamo affermare che "computare" significa

*trasformare una informazione da una forma implicita ad una forma esplicita.*

Per esempio, quando trasformiamo una espressione come  $(3+4)*2$  in  $14$ , stiamo eseguendo un calcolo, una computazione. Entrambe le espressioni rappresentano (denotano) il numero  $14$ , ma tale informazione e' ovviamente piu' esplicita in  $14$  che in  $(3+4)*2$ .

Questa definizione di computazione e' pero' troppo vaga. Dobbiamo renderla precisa, formalizzarla, attraverso la definizione di uno specifico modello computazionale. Un *modello computazionale* e' una specifica formalizzazione matematica del concetto di "computazione" e delle nozioni ad esso collegate: informazione, trasformazione (passo di computazione), ecc.

Ci sono vari modelli computazionali (e tanti altri se ne potranno sviluppare ancora), anche molto differenti tra di loro: Turing Machines, URM, Sistemi di Post,  $\lambda$ -calcolo, Programmazione logica, il formalismo delle Funzioni parziali ricorsive, ecc. ecc.

I linguaggi di programmazione che utilizziamo sono, direttamente o indirettamente, ispirati ognuno ad uno o piu' modelli computazionali. In generale, "programmare" significa descrivere un particolare processo computazionale (un algoritmo) in un linguaggio di programmazione, che si basa su uno o piu' modelli computazionali<sup>1</sup>.

I linguaggi di programmazione si possono classificare in base al modello computazionale su cui si basano<sup>2</sup>

**I Linguaggi di programmazione imperativi** (come C, Pascal, Fortran, ...) sono quelli basati su modelli come le Turing Machines o le URM. Infatti le nozioni fondamentali di tali linguaggi (formalizzate nei modelli computazionali su cui si basano) sono:

- memoria
- variabile - qualcosa il cui contenuto si puo' "leggere" e "modificare" (cella di memoria)

---

<sup>1</sup>In realta' possiamo descrivere un algoritmo direttamente in un modello computazionale, che puo' venire inteso infatti come un modo di formalizzare il concetto stesso di algoritmo.

<sup>2</sup>Molti linguaggi appartengono a piu' di una classe. Per esempio C++ e' sicuramente imperativo, ma e' anche Object-Oriented.

- assegnamento
- istruzione
- iterazione

**I Linguaggi di programmazione funzionali**<sup>3</sup> (come Haskell, Scheme, ML, OCaml, Erlang...) Tali linguaggi fanno riferimento al modello computazionale del  $\lambda$ -calcolo ed i concetti presenti nei linguaggi imperativi, così naturali<sup>4</sup> per molti di noi, sono **completamente** assenti (o assumono un significato differente). Troviamo invece nozioni come

- espressione
- variabile - da intendere nel senso matematico:
  - entità sconosciuta
  - astrazione di un valore concreto
- valutazione - differente dalla nozione di *esecuzione*:
  - quando si valuta una espressione matematica non modifichiamo alcunché’.

**I Linguaggi di programmazione logici** (come Prolog ed i suoi numerosi dialetti) Si basano su alcune proprietà di sottosistemi della logica del primo ordine (che studieremo). Il concetto di computazione coincide con il concetto di ricerca di una deduzione per una formula logica. Programmare consiste nel definire le ipotesi che ci possono permettere di dimostrare l’esistenza del valore che intendiamo calcolare.

Non elenchiamo altre classi di linguaggi di programmazione poiché nel nostro corso ci interessiamo solo ai modelli computazionali relativi alle classi sopra elencate.

---

<sup>3</sup>I linguaggi funzionali sono sempre più utilizzati anche a livello industriale ([https://wiki.haskell.org/Haskell\\_in\\_industry](https://wiki.haskell.org/Haskell_in_industry)). Alcuni esempi: Facebook utilizza Haskell per facilitare l’esecuzione delle *news feeds*; WhatsApp si affida ad Erlang per i messaging servers, arrivando a poter gestire fino a 2 milioni di utenti connessi per server; Twitter, LinkedIn, Foursquare, Tumblr, and Klout utilizzano Scala per sviluppare le core infrastructure for i siti. Inoltre, anche se non utilizza direttamente linguaggi funzionali, il popolare modello MapReduce di Google per la computazione sul cloud è stato ispirato dalle funzioni `map` and `reduce` che comunemente si trovano nella programmazione funzionale. [citazione da Hu, Z., Hughes, J., Wang, M.: *How functional programming mattered*. *Ntl. Sci. Rev.* 2(3), 349-370 (2015)]

<sup>4</sup>solo perché abbiamo solo lavorato finora principalmente con linguaggi imperativi.

## La programmazione funzionale

Prendiamo la definizione **1.41** del testo di Ausiello:

la funzione che restituisce l'inversa di una stringa  $x \in \Sigma$  e' definita come

$$\tilde{x} = \begin{cases} \varepsilon & \text{se } x = \varepsilon \\ \tilde{y}a & \text{se } x = ay \quad \text{con } a \in \Sigma \text{ e } y \in \Sigma^* \end{cases}$$

Questa definizione, oltre ad identificare univocamente una funzione sulle stringhe di un alfabeto  $\Sigma$ , descrive implicitamente anche un metodo per calcolarla. Infatti, usando questa definizione, possiamo facilmente calcolare, per esempio,  $\widetilde{cbb}$ :

$$\begin{aligned} & \widetilde{c\tilde{b}d} \\ &= \widetilde{b\tilde{d}c} \\ &= \widetilde{d\tilde{b}c} \\ &= \widetilde{\tilde{\varepsilon}dbc} \\ &= dbc \end{aligned}$$

Nei linguaggi di programmazione funzionali, l'attivita' di programmazione consiste nel definire funzioni che, come nell'esempio appena visto, descrivono implicitamente anche un metodo per calcolarle<sup>5</sup>.

Nel linguaggio **Haskell**, per esempio, la funzione "inversa di una stringa"<sup>6</sup> si descrive, mutando leggermente la sintassi, esattamente come fatto sopra:

```
inv [] = []
inv (a:y) = (inv y)++[a]
```

Quando chiediamo ad **Haskell** di calcolare l'inversa della stringa  $cbd$ , scriviamo

```
> inv "cbd"
```

---

<sup>5</sup>Non tutte le definizioni di funzioni descrivono anche un procedimento per il loro calcolo. Per esempio, si consideri il seguente sistema di equazioni differenziali

$$\begin{aligned} f : \mathbb{R} \rightarrow \mathbb{R}, g : \mathbb{R} \rightarrow \mathbb{R} \\ f'' - 6g' = 6\text{sen}(x) \\ 6g'' + a2f' = 6\text{cos}(x) \\ f(0) = 0, f'(0) = 0, g(0) = 1, g'(0) = 1 \end{aligned}$$

Esistono solo una  $f$  ed una  $g$  che soddisfano queste equazioni, e quindi questo sistema e' una definizione di (cioe' un modo di identificare univocamente)  $f$  e  $g$ . Questa definizione pero' non ci dice anche come *calcolare* il valore di  $f$  e  $g$  quando gli applichiamo degli argomenti.

<sup>6</sup>Per **Haskell** le stringhe sono liste di caratteri. La stringa vuota corrisponde quindi alla lista vuota di caratteri: `[]`.

ricevendo come risposta

```
> "dbc"
```

La computazione eseguita da **Haskell** e' simile a quella che abbiamo eseguito prima per calcolare *cbd*.

Per comprendere in dettaglio qual e' il tipo di "computazioni" eseguite dai linguaggi di programmazione funzionali e formalizzate nel modello computazionale del  $\lambda$ -calcolo, forniamo, sempre in **Haskell**, una versione equivalente e piu' "elementare" di *inv*:

```
inv = \y -> if (null y) then y
          else (inv (tail y))++[head y]
```

Per capire meglio il senso di quanto scritto sopra, *null*, *tail*, *head* e *++* sono funzioni predefinite di **Haskell**:

*null* restituisce **True** se l'argomento e' una lista vuota, **False** altrimenti;

*tail* prende una lista come argomento e restituisce una lista uguale, ma senza il primo elemento;

*head* prende una lista come argomento e restituisce il suo primo elemento;

*++* prende due liste come argomenti e restituisce la loro concatenazione.

*if...then...else* invece permette di costruire espressioni condizionali. Il valore di una espressione *if expr1 then expr2 else expr3* corrisponde al valore dell'espressione *expr2* se il valore di *expr1* e' **True**, e al valore di *expr3* se il valore di *expr1* e' **False**.

Inoltre, possiamo interpretare il senso di *inv =* come *inv e' il nome di ...*, mentre possiamo interpretare il senso di *\y ->* come *questa e' una funzione che, se prende in input un argomento y, restituisce ...*

Vediamo qualche altro esempio di funzioni definite in **Haskell**:

```
square x = x*x
```

La funzione *square* calcola il quadrato di un numero. Questa definizione e', come nel caso della funzione *inv* una rappresentazione compatta di

```
square = \x -> x*x
```

Possiamo definire in **Haskell** la funzione *fattoriale*:

```
fact 0 = 1
fact n = n*(fact (n-1))
```

che e' una rappresentazione piu' comprensibile (zucchero sintattico, si usa dire) di

```
fact = \n -> if (n==0) the 1 else n*(fact (n-1))
```

Da quanto visto, possiamo dire che quando si definisce una funzione in un linguaggio di programmazione funzionale si fanno le seguenti cose:

1. costruire una *espressione matematica* partendo da **variabili, costanti, funzioni predefinite** e *nomi*<sup>7</sup> di funzioni precedentemente definite dall'utente (quando si definiscono funzioni ricorsive si puo' utilizzare il nome stesso della funzione che stiamo definendo). A partire dagli elementi indicati, l'espressione si costruisce utilizzando l'**applicazione funzionale**.<sup>8</sup>
2. costruire una *funzione anonima* (cioe' senza nome) a partire dall'espressione costruita nel punto 1. Tale espressione viene chiamata *corpo* della funzione. Per costruire la funzione anonima identifichiamo una variabile (che chiamiamo *parametro formale della funzione*) nel corpo della funzione e utilizziamo l'operatore di **astrazione funzionale**, che in Haskell e'

\ ->

Nell'esempio del fattoriale, dopo aver costruito l'espressione (il corpo della funzione) `if (n==0) then 1 else n*(fact (n-1))` identifichiamo `n` come parametro formale e applichiamo l'operatore di astrazione funzionale, ottenendo:

```
\n -> if (n==0) then 1 else n*(fact (n-1))
```

E' importante notare che nella programmazione funzionale una funzione anonima cosi' definita e' anch'essa una espressione. Questo permette di poter definire funzioni che prendono altre funzioni come argomento, o che restituiscono funzioni come risultato.

3. **associare** l'espressione che rappresenta la funzione anonima **ad un nome**, utilizzando l'operatore `=`. Nel caso del fattoriale:

<sup>7</sup>I nomi non sono altro che variabili, a cui possiamo associare una espressione attraverso l'operatore `=`.

<sup>8</sup>Per esempio, l'espressione `x*x` e' "costruita" applicando la funzione predefinita `'*'` a due argomenti, entrambi costituiti dalla variabile `x`.

```
fact = \n -> if (n==0) then 1 else n*(fact (n-1))
```

Da notare che non dobbiamo necessariamente associare ad un nome una espressione che rappresenta una funzione anonima. Possiamo associare ad un nome una qualsiasi espressione, per esempio

```
pippo = (2+3)*7
```

Poiche' un nome (una variabile) e' un'espressione, se chiediamo ad Haskell di valutare l'espressione `pippo`, verra' valutata l'espressione associata a `pippo`, restituendo 35

Dalla breve discussione fatta possiamo capire perche' i seguenti elementi sono utilizzati in tutti i linguaggi funzionali per costruire espressioni e definire funzioni:

- variabili
- costanti
- funzioni base
- applicazione funzionale
- astrazione funzionale
- associazione di espressioni a nomi<sup>9</sup>

E' importante notare come ogni volta che si utilizza un nome in un'espressione, per esempio in `(square 7)+5`, si potrebbe evitare di usarlo, sostituendo ad esso l'espressione ad esso associata:

```
((\x -> x*x) 7) +5)
```

Sembrerebbe quindi che, anche se e' qualcosa che aiuta a migliorare la leggibilita' del nostro codice, associare espressioni a nomi non sia strettamente indispensabile. Questo pero' (almeno in apparenza) non e' vero se consideriamo definizioni di funzioni ricorsive. Infatti, se e' possibile utilizzare

```
(\x -> x*x)
```

ogni qual volta si debba usare `square`, non possiamo certo utilizzare

---

<sup>9</sup>In realta' alcuni linguaggi associano ad un nome un'espressione, per esempio Haskell, mentre altri, come Scheme, associano ad un nome un *valore*. Noi non stiamo facendo un corso di Programmazione Funzionale e quindi ci interessa solo avere una idea, seppur abbastanza precisa, di cosa significa programmare in un linguaggio funzionale.

```
\n -> if (n==0) the 1 else n*(fact (n-1))
```

ogni qual volta dobbiamo utilizzare la funzione fattoriale. Questa funzione anonima contiene `fact`, e quindi dovremmo andare a sostituire la funzione anonima al posto del nome `fact`, ottenendo però un'espressione che ancora contiene `fact` e così via ... ma non possiamo certo andare avanti all'infinito. Associare espressioni a nomi sembra quindi indispensabile per definire funzioni ricorsive.

## Computazioni nei linguaggi funzionali

Ora, per capire meglio come avviene una computazione nei linguaggi funzionali, e poter quindi capire come questo tipo di computazione è formalizzata nel  $\lambda$ -calcolo, entriamo nel dettaglio della computazione<sup>10</sup> dell'espressione

```
> inv "cbd"
```

Indichiamo con  $\rightsquigarrow_A$ ,  $\rightsquigarrow_B$  e  $\rightsquigarrow_C$  differenti tipi di *passi di computazione*. La computazione corrispondente alla valutazione dell'espressione `inv "cbd"` è descritta in Figura 1.

- Il passo computazionale  $\rightsquigarrow_A$  corrisponde a sostituire un nome con l'espressione associata ad esso.
- Il passo computazionale  $\rightsquigarrow_B$  corrisponde a sostituire il parametro attuale<sup>11</sup> di una funzione anonima al posto del parametro formale della funzione all'interno del corpo della funzione stessa.
- Il passo computazionale  $\rightsquigarrow_C$  corrisponde al calcolo di una funzione predefinita a cui abbiamo applicato un valore esplicito.

Notiamo come ogni passo computazionale nella Figura 1 *trasformi* una sottoespressione. In Figura 1 abbiamo infatti sottolineato le sottoespressioni trasformate dai vari passi di computazione.

In alcuni punti si potrebbe in realtà trasformare più di una sottoespressione. Per esempio quando si arriva a

```
if (null "cbd") then "cbd" else (inv (tail "cbd"))++[head "cbd"]
```

potremmo trasformare non solo `(null "cbd")`, ma una qualsiasi delle seguenti

---

<sup>10</sup>ovviamente quella presentata è una semplificazione a scopo didattico.

<sup>11</sup>In una applicazione  $(f e)$ , l'argomento  $e$  viene chiamato *parametro attuale* della funzione  $f$ .

---



---

```

    inv "cbd"
    - al posto di inv mettiamo la funzione associata al nome inv -
  ~>A (\y -> if (null y) then y else (inv (tail y))++[head y]) "cbd"
    - mettiamo il parametro attuale "cbd" al posto del parametro formale y
      nel corpo della funzione -
  ~>B if (null "cbd") then "cbd" else (inv (tail "cbd"))++[head "cbd"])
    - calcolo della funzione predefinita null -
  ~>C if False then "cbd" else (inv (tail "cbd"))++[head "cbd"])
    - valutazione dell'espressione if...then...else -
  ~>C (inv (tail "cbd"))++[head "cbd"])
    - calcolo della funzione predefinita tail -
  ~>C (inv "bd")++[head "cbd"])
    - al posto di inv mettiamo la funzione associata al nome inv -
  ~>A ((y -> if (null y) then y else (inv (tail y))++[head y]) "bd")++[head "cbd"]
    - mettiamo il parametro attuale "bd" al posto del parametro formale y-
  ~>B (if (null "bd") then "bd" else (inv (tail "bd"))++[head "bd"])+[head "cbd"]
    :
    :
  ~>C (if (null []) then [] else (inv [])) ++[head "d"]++[head "bd"]++[head "cbd"]
  ~>C (if True then [] else (inv []))++[head "d"]++[head "bd"]++[head "cbd"]
  ~>C []++[head "d"]++[head "bd"]++[head "cbd"]
    - valutazione della funzione predefinita head-
  ~>C []++[d]++[head "bd"]++[head "cbd"]
    - valutazione della funzione predefinita ++-
  ~>C [d]++[head "bd"]++[head "cbd"]
  ~>C [d]++[b]++[head "cbd"]
  ~>C [db]++[head "cbd"]
  ~>C [db]++[c]
  ~>C "cbd"

```

---



---

Figure 1: Valutazione di `inv "cbd"`

sottoespressioni sottolineate

```
if (null "cbd") then "cbd" else (inv (tail "cbd"))++[head "cbd"]
```

Una *strategia di valutazione* e' un algoritmo, una politica, che, quando in una espressione sono presenti piu' sottoespressioni "trasformabili" (riducibili), sceglie quella da trasformare (ridurre). Quello che distingue un linguaggio funzionale da un'altro sono soprattutto le loro strategie di valutazione. Haskell utilizza una strategia chiamata *lazy*<sup>12</sup>, che riduce il sottotermini la cui valutazione e' indispensabile per arrivare al risultato.

Nella valutazione di una espressione, potremmo in realta' anche trasformare tutte le sottoespressioni riducibili, in parallelo. Questo e' possibile nei linguaggi funzionali grazie alla mancanza di *side effects*, che sono invece presenti nei linguaggi di programmazione imperativa, dove il risultato di una computazione puo' dipendere da effetti dovuti ad altre computazioni. Per comprendere quanto affermato, prendiamo ad esempio il seguente schema di programma imperativo

```
BOOLEAN even := TRUE;    // variabile globale
:
PROCEDURE calculate (INTEGER value) : INTEGER;
BEGIN
  even := NOT even;
  IF even
  THEN   RETURN (value + 1)
  ELSE   RETURN (value + 2)
  ENDIF
END;
:
print(calculate (6));
:
print(calculate (6));
:
```

Le due istruzioni di stampa sono identiche. Ma possono produrre risultati differenti. Puo' venire stampato il valore 7, oppure 8, e questo non dipende solo dall'argomento di `calculate`, che e' 6 in entrambi i casi, o dal codice della procedura `calculate`, che viene eseguito in entrambi i casi. Dipende dal valore della variabile globale `even`, la cui modifica e' un *side effect* della esecuzione di `calculate`. Il valore restituito da `calculate` dipende quindi anche dal numero di volte che viene eseguito il suo codice. Nella programmazione funzionale, invece, poiche' le funzioni che definiamo sono vere

---

<sup>12</sup>questa strategia appartiene alla classe di strategie *call-by-name*.

funzioni matematiche, il valore di una sottoespressione e' sempre lo stesso, indipendentemente da dove si trova e da quante volte viene valutata. Questa proprieta' dei linguaggi funzionali si chiama *referential transparency*. La presenza di side effects fa intuire perche' non e' semplice comprendere cosa fa un programma imperativo o dimostrarne qualche proprieta'.

Tornando alle strategie di valutazione dei linguaggi funzionali, ci sono strategie che producono computazioni che non terminano quando si valutano espressioni che sembrano avere un valore. Per esempio, consideriamo una strategia che in una applicazione di una funzione **f** ad un argomento **arg** valuta **arg** (se questo non e' un valore esplicito) prima di valutare (**f arg**)<sup>13</sup>. Seguendo tale strategia, valutiamo ora

(**alwayseven** (**inf 3**))

dove le funzioni **alwayseven** **inf** sono definite nel seguente modo:

**alwayseven** **n** = 7

**inf** **n** = **inf n**

Se nella valutazione di (**alwayseven** (**inf 3**)) valutiamo prima (**inf 3**), otteniamo ancora (**alwayseven** (**inf 3**)), e cosi' all'infinito. Se invece valutiamo prima l'applicazione di (**inf 3**) ad **alwayseven**, otteniamo subito il valore 7.

Quindi ci sono strategie che restituiscono il valore di espressioni che, se valutate con altre strategie, produrrebbero computazioni infinite. E' quindi naturale porsi la seguente domanda: Se una strategia di valutazione produce il valore *v* di un'espressione *e*, tutte le strategie che non comportano computazioni infinite restituiscono tutte *v* come valore di *e*? Questa proprieta' e' garantita dal Teorema di Confluenza del  $\lambda$ -calcolo, che studieremo.

## **Preludio al $\lambda$ -calcolo**

Se il modello computazionale del  $\lambda$ -calcolo e' il fondamento teorico dei linguaggi di programmazione funzionale esso dovra' formalizzare gli elementi che costituiscono il minimo indispensabile per definire funzioni.

Gli elementi costitutivi della programmazione funzionale che abbiamo visto sono: variabili, costanti, funzioni base, applicazione funzionale, astrazione funzionale, definizione di funzioni ricorsive (associando nomi ad espressioni) ed i passi computazionali  $\rightsquigarrow_A$ ,  $\rightsquigarrow_B$  e  $\rightsquigarrow_C$ . Tali elementi sembrano

---

<sup>13</sup>Strategie di questo tipo appartengono alla classe di strategie *call-by-value*.

tutti indispensabili. In realta' non lo sono! Alcuni sono rappresentabili utilizzando gli altri e quindi possiamo evitare di prenderli in considerazione in un modello computazionale, che deve formalizzare solo l'*essenza* del significato di *computazione*.

In particolare, *costanti*, *funzioni base*, *definizioni ricorsive* e i passi computazionali  $\rightsquigarrow_A$  e  $\rightsquigarrow_C$ , si possono tutti definire in termini esclusivamente di *variabili*, *applicazione*, *astrazione funzionale* e passo computazionale  $\rightsquigarrow_B$ <sup>14</sup>. Il modello computazionale del  $\lambda$ -calcolo e' un formalismo estremamente semplice. E questa semplicita' si riflette sul fatto che programmare nei linguaggi funzionali e' relativamente semplice, cosi' come dimostrare proprieta' di programmi scritti in tali linguaggi.

Nel  $\lambda$ -calcolo le variabili sono rappresentate da un insieme infinito di elementi  $x, y, z, \dots$ . Le variabili sono  $\lambda$ -termini. L'applicazione e' rappresentata con l'operatore  $'\cdot'$ . L'applicazione di due  $\lambda$ -termini  $M$  e  $N$  e' anch'essa un  $\lambda$ -termine,  $(M \cdot N)$  (di solito si scrive semplicemente  $(MN)$ ). L'astrazione funzionale di un  $\lambda$ -termine  $M$  rispetto ad una variabile  $x$  e' anch'essa un  $\lambda$ -termine,  $\lambda x.M$ . I  $\lambda$ -termini formalizzano il concetto di espressione della programmazione funzionale (sappiamo che le funzioni anonime sono anch'esse espressioni e che possono essere argomenti di funzioni o possono venir restituite come output di funzioni).

Il passo di computazione  $\rightsquigarrow_B$  e' formalizzato dalla relazione di  $\beta$ -riduzione

$$(\lambda x.M)N \rightarrow_{\beta} M[N/x]$$

dove  $M$ , ed  $N$  possono essere  $\lambda$ -termini qualsiasi e dove  $[N/x]$  indica la sostituzione del parametro attuale  $N$  al posto del parametro formale  $x$  nel corpo  $M$  della funzione anonima.

Un passo di computazione nel  $\lambda$ -calcolo corrisponde a trasformare un  $\lambda$ -termine contenente un sottoterminale della forma  $(\lambda x.M)N$  nello stesso  $\lambda$ -termine, ma in cui al posto di  $(\lambda x.M)N$  c'e'  $M[N/x]$ .

Dopo questo breve preludeo, si puo' iniziare a studiare il  $\lambda$ -calcolo.

---

<sup>14</sup>La cosa che sembra impossibile e' definire funzioni ricorsive senza poter dare nomi ad espressioni. Eppure si puo'. Con i pochi elementi indicati si puo', in un certo senso, rappresentare in concetto stesso di *ricorsione*.