## 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 è ovviamente più esplicita in `14` che in `(3+4)*2`.

Questa definizione di computazione è però troppo vaga. Dobbiamo renderla precisa, formalizzarla, attraverso la definizione di uno specifico modello computazionale. Un *modello computazionale* è una specifica formalizzazione matematica del concetto di "computazione" e delle nozioni ad asso 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 più modelli computazionali. In generale, "programmare" significa descrivere un particolare processo computazionale (un algoritmo) in un linguaggio di programmazione, che si basa su uno o più modelli computazionali[1].

I linguaggi di programmazione si possono classificare in base al modello computazionale su cui si basano[2]

**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 può "leggere" e "modificare" (cella di memoria)

[1] In realtà possiamo descrivere un algoritmo direttamente in un modello computazionale, che può venire inteso infatti come un modo di formalizzare il concetto stesso di algoritmo.

[2] Molti linguaggi appartengono a più di una classe. Per esempio C++ è sicuramente imperativo, ma è anche Object-Oriented.

- assegnamento
- istruzione
- iterazione

I ***Linguaggi di programmazione*** **funzionali**[3] (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[4] 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.
- ricorsione

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.

[3] I linguaggi funzionali sono sempre più utilizzati anche a livello industriale (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 cluod 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)*]

[4] 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 $x \in \Sigma^*$ è 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, $\widetilde{cbb}$:

$$\begin{aligned} & \widetilde{cbd} \\ = \; & \widetilde{b}dc \\ = \; & \widetilde{d}bc \\ = \; & \widetilde{\varepsilon}dbc \\ = \; & dbc \end{aligned}$$

Nei linguaggi di programmazione funzionali, l'attività di programmazione consiste nel definire funzioni che, come nell'esempio appena visto, descrivono implicitamente anche un metodo per calcolarle[5].

Nel linguaggio Haskell, per esempio, la funzione "inversa di una stringa"[6] 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"
```

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

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

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

[6]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 è simile a quella che abbiamo eseguito prima per calcolare *cbd*.

Per comprendere in dettaglio qual è 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 più "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 è una lista vuota, `False` altrimenti;

`tail` prende una lista come argomento e restituisce una lista unguale, 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*** è `True`, e al valore di ***expr3*** se il valore di ***expr1*** è `False`.
Inoltre, possiamo interpretare il senso di `inv =` come `inv` *è il nome di ...*, mentre possiamo interpretare il senso di `\y ->` come *questa è 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 è, 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 è una rappresentazione più comprensibile (zucchero sintattico, si usa dire) di

```
fact = \n -> if (n==0) then 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*[7] di funzioni precedentemente definite dall'utente (quando si definiscono funzioni ricorsive si può utilizzare il nome stesso della funzione che stiamo definendo). A partire dagli elementi indicati, l'espressione si costruisce utilizzando l'**applicazione funzionale**.[8]

2. costruire una *funzione anonima* (cioé 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 è

   ```
   \ ->
   ```

   Nell'esempio del fattoriale, dopo aver costruito l'espressione (il corpo delle 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 così definita è 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:

[7] I nomi non sono altro che variabili, a cui possiamo associare una espressione attraverso l'operatore `=`.

[8] Per esempio, l'espressione `x*x` è "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 qualsiari espressione, per esempio

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

Poiché un nome (una variabile) è un'espressione, se chiediamo ad Haskell di valutare l'espressione `pippo`, verrà valutata l'espressione associata a `pippo`, restituendo `35`

Dalla breve discussione fatta possiamo capire perché 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[9]

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 è qualcosa che aiuta a migliorare la leggibilità del nostro codice, associare espressioni a nomi non sia strettamente indispensabile. Questo però (almeno in apparenza) non è vero se consideriamo definizioni di funzioni ricorsive. Infatti, se è possibile utilizzare

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

in una espressione dove si utilizzi `square`, non possiamo certo utilizzare

[9]In realtà 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))
```

in un punto dove debba essere utilizzata 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[10] 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[11] 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"]
```

[10] ovviamente quella presentata è una semplificazione a scopo didattico.

[11] In una applicazione $(f\, e)$, l'argomento $e$ viene chiamato *parametro attuale* della funzione $f$; mentre in una definizione di funzione, per esempio $fx = x + 3$, il nome dell'argomento ($x$ nel nostro esempio), viene chiamato *parametro formale*.

`inv "cbd"`

*- al posto di* `inv` *mettiamo la funzione associata al nome* `inv` *-*

$\rightsquigarrow_A$ `(\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 -*

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

*- calcolo della funzione predefinita* `null` *-*

$\rightsquigarrow_C$ `if False then "cbd" else (inv (tail "cbd"))++[head "cbd"])`

*- valutazione dell'espressione* `if...then...else` *-*

$\rightsquigarrow_C$ `(inv (tail "cbd"))++[head "cbd"])`

*- calcolo della funzione predefinita* `tail` *-*

$\rightsquigarrow_C$ `(inv "bd")++[head "cbd"])`

*- al posto di* `inv` *mettiamo la funzione associata al nome* `inv` *-*

$\rightsquigarrow_A$ `((\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`*-*

$\rightsquigarrow_B$ `(if (null "bd") then "bd" else (inv (tail "bd"))++[head "bd"])++[head "cbd"])`

⋮ ⋮

$\rightsquigarrow_C$ `(if (null []) then [] else (inv []) )++[head "d"]++[head "bd"]++[head "cbd"]`

$\rightsquigarrow_C$ `(if True then [] else (inv []))++[head "d"]++[head "bd"]++[head "cbd"]`

$\rightsquigarrow_C$ `[]++[head "d"]++[head "bd"]++[head "cbd"]`

*- valutazione della funzione predefinita* `head`*-*

$\rightsquigarrow_C$ `[]++[d]++[head "bd"]++[head "cbd"]`

*- valutazione della funzione predefinita* `++`*-*

$\rightsquigarrow_C$ `[d]++[head "bd"]++[head "cbd"]`

$\rightsquigarrow_C$ `[d]++[b]++[head "cbd"]`

$\rightsquigarrow_C$ `[db]++[head "cbd"]`

$\rightsquigarrow_C$ `[db]++[c]`

$\rightsquigarrow_C$ `"bdc"`

Figure 1: Valutazione di `inv "bdc"`

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

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

Una *strategia di valutazione* è un algoritmo, una politica, che, quando in una espressione sono presenti più sottoespressioni "tasformabili" (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*[12], che riduce il sottotermine la cui valutazione è indispensabile per arrivare al risultato.

Nella valutazione di una espressione, potremmo in realtà anche trasformare tutte le sottoespressioni riducibili, in parallelo. Questo è 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 può 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. Può venire stampato il valore 7, oppure 8, e questo non dipende solo dall'argomento di `calculate`, che è 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 è un *side effect* della esecuzione di `calculate`. Il valore restituito da `calculate` dipende quindi

[12] questa strategia appartiene alla classe di strategie *call-by-name*.

anche dal numero di volte che viene eseguito il suo codice. Nella programmazione funzionale, invece, poiché le funzioni che definiamo sono vere funzioni matematiche, il valore di una sottoespressione è sempre lo stesso, indipendentemente da dove si trova e da quante volte viene valutata. Questa proprietà dei linguaggi funzionali si chiama *referential transparency*. La presenza di side effects fa intuire perché non è semplice comprendere cosa fa un programma imperativo o dimostrarne qualche proprietà.

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 è un valore esplicito) prima di valutare `(f arg)`[13]. Seguendo tale strategia, valutiamo ora

`(alwayseven (inf 3))`

dove le funzioni `alwayseven` e `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 così 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 proprietà è garantita dal Teorema di Confluenza del $\lambda$-calcolo, che studieremo.

### Funzioni di ordine superiore

Rispetto ai linguaggi imperativi, i linguaggi funzionali trattano le funzioni come un qualsiasi valore manipolabile dal linguaggio. Come conseguenza, è possibile definire funzioni che prendono altre funzioni come argomento o che restituiscono funzioni come risultato. Questo genere di funzioni vengono chiamate solitamente "funzioni di ordine superiore" per distinguerle da quelle che si utilizzano nei linguaggi non funzionali e che prendono come argomento - e restituiscono come risultato - solo elementi di tipi di dato che non sono

[13]Strategie di questo tipo appartengono alla classe di strategie *call-by-value*.

funzioni.

Un banale esempio in Haskell di funzione che prende un altra funzione come argomento e' il seguente:

```
atzero f = f 0
```

`atzero` prende come argomento una funzione e restituisce il valore di quest'ultima sul numero 0.
Vediamo come Haskell valuta l'espressione `(atzero square)`:

```
   (atzero square)
⇝A ((\f -> (f 0)) square)
⇝B (square 0)
⇝A ((\n -> (n*n)) 0)
⇝B (0*0)
⇝C 0
```

È possibile definire funzioni in cui sia gli argomenti che i risultati sono funzioni. Un semplicissimo esempio è la funzione `compose`, che prende due funzioni (unarie) come argomenti e restituisce la loro composizione.

```
compose f g = \x -> (f (g x))
```

Potremmo ora definire

```
inc n = n+1
```

e poi far valutare `((compose square inc) 3)`, ottenendo `16`.
Oppure definire

```
squareinc = (compose square inc)
```

ed utilizzare la funzione `squareinc` tutte le volte che abbiamo bisogno di fare il quadrato dell'incremento di uno.

La possibilità di definire funzioni di ordine superiore fornisce una grande flessibilità ed espressività ai linguaggi funzionali.

### Curryficazione

La nozione di funzione di ordine superiore ci permette di introdurre quella di "curryficazione", utilizzata in molti linguaggi funzionali. Curryficare significa trasformare una funzione

$$f : (A_1 \times \ldots \times A_n) \to B$$

in una funzione di ordine superiore

$$f_C : A_1 \to (A_2 \to (\ldots \to (A_n \to B)..))$$

tale che

$$f(a_1, \ldots, a_n) = f_C(a_1)(a_2) \ldots (a_n)$$

In Haskell le funzioni che definiamo vengono considerate implicitamente curryficate. Questo significa che potremmo definire `compose` semplicemente nel seguente modo

```
compose f g x = (f (g x))
```

e comunque poter far valutare `((compose square inc) 3)` e definire

```
squareinc = (compose square inc)
```

È importante notare che utilizzare le versioni currificate delle funzioni al posto delle funzioni stesse non è limitativo, poichè ogni volta che ci servisse il valore di $f(a_1, \ldots, a_n)$ potremmo utilizzare $f_C(a_1)(a_2) \ldots (a_n)$. Al contrario, avere le versioni curryficate delle funzioni fornisce maggiore flessibilità ad un linguaggio di programmazione.
Questo è il motivo per cui nel modello computazionale del $\lambda$-calcolo le funzioni che si introducono hanno solo un argomento.Al posto di definire funzioni a più argomenti, si possono definire, in modo equivalente, le loro versioni curryficate.

### Uso dell'induzione per dimostrare proprietà di programmi

Abbiamo accennato prima alla proprietà di referencial transparency propria dei linguaggi (puramente) funzionali. Questa permette l'uso di tecniche matematiche per manipolare programmi (per esempio per ottimizzarli rispetto a particolari parametri) oppure per dimostrarne proprietà, come ad esempio la correttezza.

La ricorsione è alla base della potenza computazionale dei linguaggi di programmazione funzionali. Si può notare come lo schema di una definizione

ricorsiva di una funzione sia analogo a quello utilizzato nelle dimostrazioni matematiche per induzione. Infatti in un programma ricorsivo l'output e' definito per gli elementi di base e poi è definito per un input generico assumendo che il programma funzioni correttamente su elementi più "piccoli" ("semplici") dell'input generico.
Similmente, in una dimostrazione matematica per induzione, la proprietà in oggetto è dimostrata per il caso base (solitamente 0) e poi per il caso generale assumendo che valga per numeri più piccoli.

Il principio di induzione è una regola logica che però, implicitamente, "rappresenta" anche un metodo di calcolo, e nella programmazione funzionale corrisponde alla definizione di funzioni numeriche per ricorsione, come il programma Haskell per il fattoriale visto in precedenza.

La stretta corrispondenza tra programmi ricorsivi e principio di induzione rende possibile l'uso di quest'ultimo per dimostrare proprietà di programmi ricorsivi. Possiamo usare l'induzione per esempio per dimostrare la seguente affermazione relativa alla funzione Haskell `fact`:

*Per ogni input* `n` *il calcolo (la valutazione) di* `(fact n)` *termina.*

Basta infatti mostrare che

1. la valutazione di `(fact 0)` termina;

2. per un generico `k` $> 0$, la valutazione di `(fact k)` termina, utilizzando l'ipotesi induttiva che `(fact (k-1))` termini.

Prendiamo quindi il programma che calcola il fattoriale e vediamo che

1. la valutazione di `(fact 0)` termina, poiche' corrisponde immediatamente a restituire 0;

2. se `k` $> 0$, la valutazione di `(fact k)` corrisponde a valutare il ramo `else` e quindi a valutare l'espressione `(k*(fact (k-1)))`. Poichè per ipotesi induttiva la valutazione di `(fact (k-1))` termina, e poichè il calcolo della sottrazione e della moltiplicazione su due numeri banalmente termina, l'intera valutazione dell'espressione `(fact k)` termina.

Per poter dimostrare proprietà di programmi che lavorano su strutture dati complesse avremo ovviamente necessità di utilizzare strumenti matematici più sofisticati, come ad esempio il principio di **induzione ben-fondata** (vedi [**VS**]).

### Preludio al $\lambda$-calcolo

Se il modello computazionale del $\lambda$-calcolo è il fondamento teorico dei linguaggi di programmazione funzionale esso dovrà 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 tutti indispensabili. In realtà 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 $\rightsquigarrow_B$[14]. Il modello computazionale del $\lambda$-calcolo è un formalismo estremamente semplice. E questa semplicità si riflette sul fatto che programmare nei linguaggi funzionali è relativamente semplice, così come dimostrare proprietà di programmi scritti in tali linguaggi.

Nel $\lambda$-calcolo le variabili sono rappresentate da un insieme infinito di elementi $x$, $y$, $z$, .... Le variabili sono $\lambda$-termini. L'applicazione è rappresentata con l'operatore $'\cdot'$. L'applicazione di due $\lambda$-termini $M$ e $N$ è 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 $x$ è 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 $\rightsquigarrow_B$ è 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.

[14]La cosa che sembra impossibile è definire funzioni ricorsive senza poter dare nomi ad espressioni. Eppure si può. Con i pochi elementi indicati si può, in un certo senso, rappresentare in concetto stesso di *ricorsione*.

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

Dopo questo breve preludio, si può iniziare a studiare il $\lambda$-calcolo.