Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

La Programmazione Funzionale nella Babele dei Linguaggi di Progr.

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **1**, 22 febbraio 2022

Lezione 0:

Informazioni Pratiche

Canali di Comunicazione

Lezioni:

Martedì 14-16: aula Alfa via Salaria
Venerdì 16–19 – aula Magna Regina Elena
modalità **BLENDED**.

Pagina web & classroom:

https://twiki.di.uniroma1.it/twiki/view/TPFI/WebHome

classroom.google.com/c/NDY5MzQyMjg4Njgz

Trovate:

- diario delle lezioni
- materiali vari (**registrazioni**)
- esempi di codice
- **queste slides**

Testi consigliati & consultazione

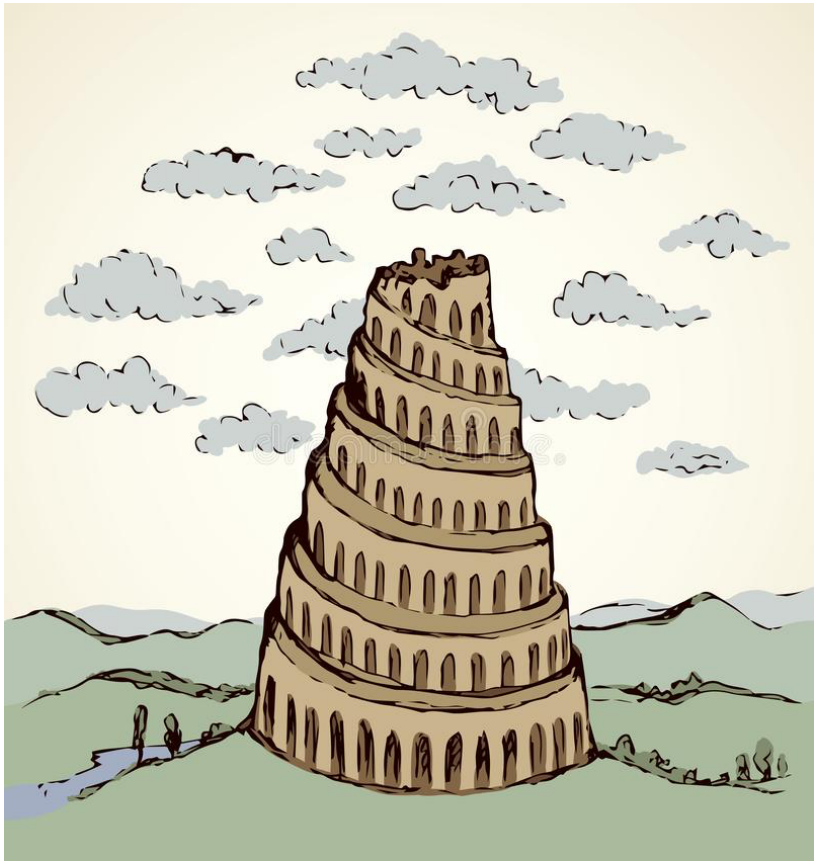
- **Testi Consigliati**
  - R. Bird: *Thinking Functionally with Haskell*. Cambridge University Press (2015).
  - B.W.Kernigham, D. Ritchie: *The C Programming Language* (2nd edition), Pearson (1988).
- **Testi di consultazione e di ispirazione**
  - E. W. Dijkstra: *A Discipline of Programming*, Prentice Hall (1976).
  - R. Bird: *Pearls of Functional Algorithm Design*, Cambridge University Press (2010).
  - R. Bird, P. Wadler: *Introduction to Functional Programming*, Prentice Hall (1988).
  - A. Oram, G. Wilson (eds.): *Beautiful Code*, O'Reilly (2008).
  - J. Bentley: *Programming Pearls* (1986) and *More Programming Pearls* (1988), Addison Wesley.

Esame

Idea:

- **homework**: assegnati circa ogni 2 settimane.
- **colloquio orale**: comprensivo di discussione homework

L'anno scorso ha funzionato bene ☺



Lezione 1.1
La Babele dei Linguaggi di Programmazione

Programmazione vs Algoritmi

Qual è la differenza tra la **programmazione** e gli **algoritmi**?

Prendete un algoritmo di cui avete dimostrato la correttezza:

scrivete il programma e testatelo: usualmente non funziona.

Ma cosa può andare storto?

Tipicamente:

- **Operazioni astratte non** implementate correttamente
- **side-effects** e altri problemi legati alla **memoria**
- Anche l'**efficienza** del programma potrebbe essere non coerente con la complessità asintotica calcolata

Esempio: riflettete su quante sottigliezze possano nascondere soprattutto le liste e altre strutture dinamiche.

Esempio (linguaggio C)

Esercizio 2 (10 punti) Considerare il problema di verificare se una lista di caratteri è palindroma, cioè se letta da sinistra a destra o da destra a sinistra dà la stessa sequenza di caratteri (come le parole 'non', 'otto', 'radar', 'ingegni', 'onorarono', 'ossesso', ...).

1. (**3 punti**) Considerate il seguente programma (dove la funzione `eqList` verifica se due liste sono uguali e `reverse` restituisce il puntatore alla testa di una lista che contiene gli elementi di `L` in ordine rovesciato):

```
int palindroma(charList L){
    if (eqList(L, reverse(L)) return 1;
       else return 0;
   }
```

   Sotto quali ipotesi sulla funzione `reverse` la funzione dà risultati corretti?

reverse **deve lasciare** L **immutata**. Cioè deve essere una funziona che **crea una nuova lista come risultato**.

Assolutamente **imprevedibili i risultati** se L viene modificata.

Testare cosa accade con il vostro linguaggio preferito (Python?)

Computabilità e Pratica della Programmazione

Da un punto di vista **puramente teorico**, bastano modelli di calcolo (~linguaggi di programmazione) estremamente minimali.

In un corso di Calcolabilità e Complessità probabilmente vi hanno introdotto (o introdurranno?) modelli di calcolo minimali (Macchine di Turing, Macchine a Registri, noi **λ-calcolo**)

Ma vogliamo questo?

I linguaggi di programmazione devono fornire **astrazioni sul controllo** (ad esempio: **cicli**, **funzioni** etc.) e **astrazioni sui dati** (definizione dei **tipi di dato**, con operazioni e regole di visibilità) e meccanismi per **costruire astrazioni**.

"**Un linguaggio deve fornire un piccolo numero di metafore uniformi con chiara semantica**"
[Dan Ingalls "**Design Principles behind Smalltalk**"]

Evoluzione dei Linguaggi di Programmazione 1

- **Linguaggi Macchina**
  - **Istruzioni**: trasferimento di memoria/aritmetiche, **identificate da stringhe di bit**
  - completo controllo del processore e della memoria
  - **Tipo di dato**: celle di memoria (**stringhe di bit!**)
  - **Controllo**: salto (in)condizionato
- **Linguaggi Assembler**
  - Uso di **nomi e label simbolici** per istruzioni e dati (`add %d1 3` invece di `000123 010012 0000011`)
  - 1:1 con istruzioni macchina
  - Uso di label simboliche: **astrazioni utile?** [se aggiungo/tolgo un'istruzione non devo modificare tutti i salti]

Evoluzione dei Linguaggi di Programmazione 2

- **Linguaggi Imperativi**
  - **Istruzioni**: Assegnazioni, cicli
  - **Tipo di dato**: **astrazione** delle celle di memoria: controllo sui tipi
  - Definizioni di **nuovi tipi di dato**.
  - **Controllo**: cicli + **astrazione procedurale/funzionale**.
- **Linguaggi Funzionali**
  - Definizione di **funzioni ricorsive**.
  - Controllo: **riduzione di espressioni**.
  - **Niente memoria**, ma ambiente di valutazione.
  - No alias/side-effects: **ricca teoria algebrica**.
- **Linguaggi Logici**
  - Definizione logica della specifica genera un meccanismo computazionale.
  - Programmare **cosa** invece di programmare **come**.

Famiglie di Linguaggi di Programmazione

- **Imperativi**: un programma è essenzialmente una **sequenza di comandi** (sequenza, salti, cicli) che modificano la **memoria** di una macchina (**assegnazione**). L'esecuzione è una sequenza di trasformazioni della memoria: **Algol**, **Fortran**, **C**, **Pascal**, ...
- **Orientati agli Oggetti**: il programma consiste nella definizione di **classi** (≈ tipi) da cui creare **oggetti** (≈ dati). L'esecuzione di un programma è costituito dall'interazione di oggetti (**scambio di messaggi**). Gli oggetti possono essere visti come dati o piccoli automi. **Smalltalk**, **Java**, **C++**, …
- **Funzionali**: un programma è una sequenza di definizione di **funzioni ricorsive**. L'esecuzione consiste nella **riduzione di espressioni** (analoga alla riduzione di un'espressione aritmetica/algebrica): **Haskell**, **ML**, **LisP**, …
- **Logici**: specificano una **formula logica** di **cosa** il programma deve calcolare. Il controllo è implicito: **Prolog**, …

In questo corso…

Graffieremo la superficie del paradigma **funzionale** e a **imperativo**:

- Vedremo le basi di **Haskell** come esempio di linguaggio funzionale;
- Vedremo qualche come alcuni comportamenti si possono mimare con un linguaggio imperativo (C).
- Confronteremo soluzioni in paradigmi diversi.

Lezione 1.2

Cos'è la Programmazione Funzionale?

Why Functional Programming Matters

Perché è **bella** ed elegante… ma soprattutto perché:

- pone l'enfasi sulle **funzioni** e la loro **applicazione**, invece che sui **comandi** e **trasformazioni di memoria**.
- usa **notazioni matematiche** "semplici" che permettono di descrivere soluzioni e problemi in modo **chiaro** e **conciso**.
- permette di valutare la **correttezza** (ed **equivalenza** di programmi) grazie a **ragionamento equazionale**.
- Molte delle sue caratteristiche sono state **incorporate in altri paradigmi**, o **pattern di programmazione** ed è istruttivo vederle nella loro purezza, per es.:
  - ricorsione
  - tipi generici
  - dati immutabili
  - composizionalità

Cos'è un programma funzionale?

Essenzialmente: un'**espressione** che si **riduce**.

Come alle medie.

```
7 + 5 * (4 + 3) =          { prima dentro le parentesi }
= 7 + 5 * 7          { prima i prodotti, regole di valutazione! }
= 7 + 35                   { poi le somme }
= 42                       { non ci sono più operazioni}
```

Osservate che **non c'è bisogno di memoria** e di **assegnazioni tutta l'informazione** necessaria **è contenuta dentro l'espressione** e le sue forme intermedie.

Osservate anche che **il risultato è implicito fin dall'inizio** e non dipende da fattori esterni.

Funzioni, tipi e applicazione

In Haskell si usa la notazione:

```
f :: X -> Y
```

che significa che `f` è una funzione il cui **dominio** (= argomento) è di tipo `X`, mentre il **codominio** (= risultati) è di tipo `Y`, del tutto analoga a quella usata in matematica (`::` indica che si tratta di un **tipaggio**)

Ecco alcune **funzioni predefinite** in Haskell (in **prima approssimazione**):

```
sin :: Float -> Float
logBase :: Float -> Float -> Float
div: Integer -> Integer -> Integer
```

Interagire con l'interprete

Haskell permette di compilare i programmi, ma soprattutto all'inizio può essere utile interagire con l'**interprete** `GHCi`: a prompt di comando possiamo **scrivere un'espressione** che viene **immediatamente valutata**. Ad esempio:

```
Prelude> 7 + 5 * (4 + 3)
42

Prelude> sin pi
1.2246467991473532e-16
```

dove `Prelude` è l'ambiente **standard** di funzioni predefinite a disposizione (tra cui `sin`, `+`, `pi` (costante pi greco) e molte altre…

Oppure potete chiedere il **tipo** di una **funzione** o di un'**espressione**:

```
Prelude> :type 7 + 5 * (4 + 3)
Integer
```

Alcuni tipi e costruttori di tipo

Oltre a `Integer`, `Float` , `Char` e altri tipi predefiniti, in Haskell ci sono costruttori di tipo:

Liste: sono il tipo principe dei linguaggi funzionali fin dal LISP. Sono sequenze di valori **omogenei** (= stesso tipo), **processabili sequenzialmente**. Si scrivono semplicemente tra parentesi quadre. Ad esempio:

```
[3, 7, 11] :: [Integer]
[[], [3]] :: [[Integer]]  --liste di liste
```

Stringhe: sono semplicemente **liste di caratteri**. Haskell offre dello **zucchero sintattico**: "`tpfi`" per `['t','p','f','i']::[Char]`

Coppie: *n*-uple di valori, anche **disomogenei**. Per esempio:

```
("pippo", 5) :: ([Char], Integer)
(3, 'c', (11, 3.14)) :: (Integer, Char, (Integer, Float))
                -- anche le tuple si possono annidare
```

Grandi assenti: **array mutabili**.

Operatori infissi e sezioni

Ovviamente Haskell permette **operatori infissi**, come `+` (somma tra valori numerici), `++` (concatenazione tra liste), `.` (composizione di funzioni) e molti altri.

Gli operatori, però, **sono funzioni**. Per motivi pratici spesso è utile avere operatori infissi come fossero funzioni: in tal caso si scrivono tra parentesi.

`3 + 3` è equivalente a `(+) 3 3`

`[1,2,3] ++ [4,5]` è equivalente a `(++)[1,2,3] [4,5]`

Notare che + e ++ sono una **funzioni di 2 argomenti** e non di una coppia di numeri, come probabilmente saresti tentati di scrivere.

Detto in altri termini:

```
(+) :: Integer -> Integer -> Integer,
```

e non `(+) :: (Integer, Integer) -> Integer`

Funzioni: come operatori infissi

Viceversa, data una funzione binaria può sempre essere scritta in modo infisso, usando gli apici rovesciati:

```
Prelude> 5 `div` 3
1
```

Osserviamo che in Haskell, possiamo fare **applicazioni parziali** di una funzione. Ad esempio, posso definire:

```
Prelude> piu3 = (+) 3
         -- piu3 ha già mangiato un argomento
Prelude> :t piu3
piu3 :: Integer -> Integer
Prelude> piu3 5
8
```

Si può anche scrivere **(+3)** per indicare la funzione unaria che somma 3.

Polimorfismo

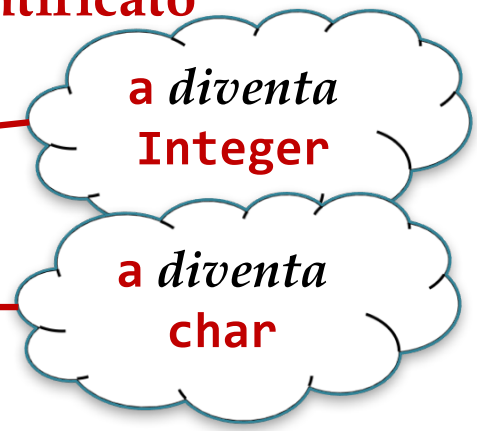
Ma qual è il tipo dell'operatore ++?

```
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
```

a è una **variabile di tipo**. Il tipo va inteso **quantificato universalmente** ("**per ogni tipo** *a*"). Infatti:

```
Prelude> [1,2,3] (++) [4,5]
[1,2,3,4,5]

Prelude> "I love " ++ "haskell"
"I love haskell"
```



Attenzione che il tipo di (++) in realtà impone un **vincolo**: i tipi delle due liste argomento e del risultato, possono essere qualsiasi, ma **le tre liste devono avere lo stesso tipo**. Infatti:

```
Prelude> [1,2,3] (++) "haskell"
<interactive>:20:2: No instance for (Num Char)
        arising from the literal '1'
```

cioè **errore di tipo**: non posso avere liste **miste** di caratteri e interi.

Polimorfismo "limitato": classi

Anche il tipo di `(+)` è leggermente più complicato:

```
Prelude> :t (+)
(+) :: (Number a) => a -> a -> a
```

che va letto: "**per ogni tipo *a* che è almeno un numero**": è simile **alle interfacce** in Java, e significa tutti i tipi su cui è definito `(+)`, cioè in questo caso, tutti i tipi numerici.

`Number` è una **classe**.

Due classi particolarmente importanti sono: **`Eq`** (cioè tipi che **ammettono uguaglianza**) e **`Ord`** (tipi che ammettono una **relazione d'ordine**)

Ad esempio, una ipotetica funzione di ordinamento, dovrebbe avere tipo:

```
sort :: (Ord a) => [a] -> (a -> a -> Bool) -> [a]
```

cioè "presa una lista con elementi ordinabili, una relazione d'ordine, restituisce la lista ordinata".

Il valore indefinito

In Haskell è predefinita una 'costante indefinita'

```
Prelude> undefined
*** Exception: Prelude.undefined
```

Più interessante il tipo di `undefined`:

```
undefined :: a
```

Quale valore mai può avere tipo $\forall \alpha.\ \alpha$? Ovviamente un **valore indefinito**, che non dà **nessuna informazione**, neanche il tipo a cui appartiene.

Volendo, in Haskell possiamo fabbricarcelo in casa, con un'equazione verificata da **qualsiasi valore**, vedremo più avanti.

Avendo tipo $\forall \alpha.\ \alpha$ ha qualsiasi altro tipo, perché tutti i tipi sono istanza di $\forall \alpha.\ \alpha$. anche `[Integer]`, `char`, `(a->b)->a`, etc.

È il rappresentante canonico delle computazioni indefinite, ad esempio non terminanti!

Composizionalità ed Equazioni

Un aspetto importante della programmazione funzionale è la facilità con cui si **compongono i programmi**.

Vediamo il **funzionale** `map` e un primo funzionale e un esempio di **ragionamento equazionale**.

Ci siamo convinti che `map (f . g) = (map f) . (map g)`

```
map :: (a -> b) -> [a] -> [b]
-- applica una funzione agli elementi di una lista
> map (*2) [1, 2, 3]
[2, 4, 6]
> map length [“I”, “love”, ”haskell”]
[1, 4, 7]
> map ((*2) . length) [“I”, “love”, ”haskell”]
[2, 8, 14] -- le funzioni si possono comporre!
> map (*2) (map length [“I”, “love”, ”haskell”])
[2, 8, 14]
> (map (*2) . (map length)) [“I”, “love”, ”haskell”]
[2,8, 14]
```

Definizioni di Funzioni

Una funzione viene definita essenzialmente dando un nome a una definizione. Usualmente si usa la **ricorsione**.

```
fatt n = if n==0 then 1 else n * fatt (n-1)
```

Osservazioni:

- parentesi: in Haskell l'**applicazione** di funzioni **associa più di tutto** (anche degli operatori infissi).
- il ramo `else` dell'`if` **non è opzionale**: si tratta di un'espressione e deve dare **un risultato in tutti i casi** (e sempre dello **stesso tipo** in tutti i rami)
- Sarebbe infatti **mal tipata** un'espressione come:

```
if n==0 then 1 else 'a'
```

Valutazione Lazy

Definiamo ora una funzione costante:

```
k x = 42
k :: (Num a) => t -> a
```

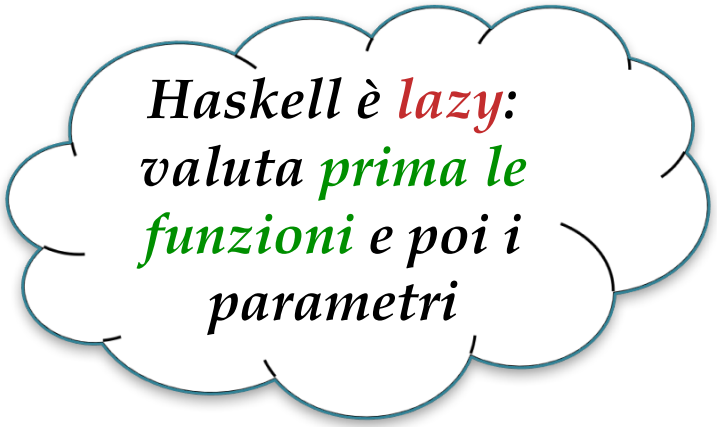
Definiamo ora la funzione:

```
omega x = omega x
omega :: t1 -> t2
```

che è una curiosa funzione che **trasforma un valore di un qualsiasi tipo** `t1` **in un valore di un qualsiasi altro tipo** `t2`.

Valutiamo k:

```
> k (omega 55)
42
> k undefined
42
```



Lezione 1

That's all Folks…

Grazie per l'attenzione…

…Domande?