Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Quegli Strani Tipi (Polimorfi)
Definizioni di Funzioni

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **3**, 1 marzo 2022

Lezione 3a:
Tipi base, costruttori di tipo e definizioni di funzioni

Tipi predefiniti I: Bool

Esistono una serie di tipi predefiniti (o **tipi base**): **Bool**, **Int**, **Integer** (interi a precisione **illimitata**) **Char**, **String**, **Float**, **Double** con significato (spero) chiaro.

Vediamo alcuni esempi, cominciando dal tipo **Bool**.

```
 -- usuali costanti per Bool
>:t False
False :: Bool
>:t True
True :: Bool
 -- e le usuali funzioni:
>not True
False
>True && False
False
 -- operator session:
>(&&) True False
False
 -- infatti:
>:t (&&)
(&&) :: Bool -> Bool -> Bool
```

```
-- undefined ha tutti i tipi
:t True && undefined
Bool
> False && undefined
False
-- ma:
> True && undefined
*** Exception: Pre.undefined
-- e:
> undefined && False
-- (suspense)
*** Exception: Pre.undefined
-- neanche in Haskell, && è
-- commutativo
```

Definizioni di Funzioni

Abbiamo già visto come definire funzioni con **equazioni ricorsive** e **lambda-notazione**, affidandoci al nostro intuito di informatici. Vediamo ora altri dettagli.

Ricordo che il **ramo else** nei linguaggi funzionali **non è opzionale**! infatti è un'espressione e deve sempre restituire un valore!

Qualora i rami siano più complessi (ma qui lo vediamo ancora con il `not`) si possono usare le **espressioni guardate** (analoghi ai **case**)

```
-- espressioni condizionali.
myNot x = if x then False else True

-- espressioni guardate:
myNot x
 | x         = False
 | otherwise = True
-- si valutano in ordine,
-- otherwise ≡ True
```

si prende il primo che valuta a `True`

i nomi delle funzioni devono cominciare con minuscola

Pattern Matching (1)

Il Vero Programmatore Haskell (VPH) però userà di preferenza il **pattern matching**: la funzione viene definita per ogni forma (**sintattica**) del parametro.

Anche nel pattern-matching l'ordine è importante (viene eseguita la **prima clausola** che fa **matching** col **parametro**).

Quando non utile nella definizione della funzione una parte del pattern si può usare _ (underscore, **parametro anonimo**)

```
-- Patten matching sui booleani:
myNot' False = True
myNot' True  = False

-- La seconda clausola fa SEMPRE matching…
-- ma vengono analizzate in ordine…
myNot'' False = True
myNot''   x   = False

-- Non è utile il nome del secondo parametro
myNot''' False = True
myNot'''   _   = False
```

Pattern Matching (2)

Vediamo qualche esempio più complesso con funzioni binarie.

Attenzione: il **pattern matching** non permette di fare `**unificazioni**', cioè di richiedere che parti del pattern **siano uguali tra loro**.

```
-- Pattern matching con funzioni binarie
myAnd True True = True
myAnd   _    _  = False
-- similmente con l'or logico
myOr False False = False
myOr   _    _    = True
-- Il Pattern matching non permette unificazioni!
xor x x = False
xor _ _ = True
  Conflicting definitions for 'x'
-- Meglio un'equazione guardata
xor x y
|    x/=y    = True
| otherwise = False
```

```
-- myAnd può essere poi usata
-- in forma binaria:
>True `myAnd` False
False
```

```
-- se piace, anche nelle def:
True `myOr` _ = True
  _  `myOr` x = x
```

x x non è un pattern accettabile

Pattern Matching sui naturali

Come per i booleani, possiamo **definire funzioni** sugli **interi** interi per **pattern matching**.

Attenzione che come già detto, l'**ordine conta**!

```
-- è necessario mettere prima la clausola per 0!
fatt 0 = 1
fatt n = n * fatt (n-1)
>:t fatt
(Num a, Eq a) => a -> a
-- oppure fibonacci:
-- la famigerata funzione inefficiente!
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
-- qui non abbiamo l'iterazione!
-- Come renderla efficiente?
```

n è solo una variabile e fa matching con tutto!

il pattern più generale va necessariamente per ultimo

Int e Integer

In Haskell ci sono due tipi per gli interi: Int (interi "classici" con valori contenuti in una **parola di memoria**) e Integer (interi illimitati, come in Python, per esempio).

Noi possiamo "forzare" il tipo di una funzione, dichiarandolo preventivamente.

```
fatt' :: Int -> Int
fatt' 0 = 1
fatt' n = n * fatt' (n-1)
-- osservate:
>fatt 20
2432902008176640000
>fatt' 20
2432902008176640000
-- ma:
>fatt 21
51090942171709440000
>fatt' 21
-4249290049419214848
```

Costruttori di tipo: Tuple

Ricordo che in accordo con il λ-calcolo, l'**applicazione** di funzione si scrive **(f v_1 … v_n)** (**senza virgole**) dove v_1, …, v_n sono gli **argomenti** ed `f` (cioè il primo termine) è una funzione (o meglio, il termine in **posizione funzionale**).

La scrittura tipica dei linguaggi imperativi **f(v_1, …, v_n)** significa un'altra cosa, e cioè la funzione `f` applicata a un **unico argomento** di tipo **tupla**.

```
>:t (False, True)
(False, True) :: (Bool, Bool)
 -- le tuple hanno dimensione arbitraria
 -- e possono essere non omogenee
>:t ("tpFi", 42, True, 'E')
("tpFi",42,True,'E')::(String, Integer, Bool, Char)
 -- sulle coppie ci sono le funzioni predefinite
 -- fst e snd che estraggono primo e secondo elemento
>:t fst
fst :: (a, b) -> a
>:t snd
snd :: (a, b) -> b
```

Pattern matching: tuple e lambda

Anche sulle tuple possiamo definire funzioni per pattern matching. Vediamo come si scrivono **fst** e **snd**.

Il pattern matching si può addirittura usare nelle lambda-espressioni (o **funzioni anonime**).

```
-- posso definire myFst e myFst usando pattern matching
-- per decomporre la tupla (distruttore).
myFst (x, y) = x
mySnd (x, y) = y
-- meno comune, ma possibile il pattern matching
-- nelle lambda espressioni
myFst’ = \(x,y) -> x
mySnd’ = \(x,y) -> y
-- versione non currificata di somma
somma (x, y) = x + y
```

Espressioni let

La semantica dell'espressione **let** x=N **in** M è del tutto **analoga** all'applicazione $(\lambda x.M)N$. Tuttavia ci sono **importanti differenze** nei **tipaggi**.

Ad esempio `let x = \x -> x in x x` è **tipabile in Haskell**, mentre ovviamente non lo è `(\x -> x x) (\x -> x)` semplicemente perché **non è tipabile** il sottotermine per l'**auto-applicazione**.

Ma qual è la differenza? Nel secondo caso, tipando l'applicazione `x x` **so già** che **x** è **legata all'identità**, che ha un tipo che si può auto-applicare (vedi lezione precedente).

In generale però questo non è vero e non si riesce a tipare `x x`, quando gli argomenti potrebbero essere qualsiasi.

```
-- uso di let per decomporre valori strutturati
myFst' z = let (x, _) = z in x
mySnd' z = let (_, y) = z in y
```

Espressioni let: fibonacci efficiente

L'espressione **`let`** può essere molto **comoda** nella **definizione di funzioni** quando queste hanno ricorsioni con valori strutturati.

Scriviamo una versione efficiente di Fibonacci che itera la trasformazione $(fib(i), fib(i\text{-}1)) \rightarrow (fib(i+1), fib(i))$ [idea già vista coi numerali di Church]

Questa tecnica è detta **tupling** (famigliare ai Python speaker).

L'espressione **`let`** in questo caso, ci serve per **decomporre la coppia** che ci arriva come risultato della chiamata ricorsiva, usando il **pattern matching**.

```
fibAux 0 = (0,0) -- il secondo valore non è importante
fibAux 1 = (1,0)
fibAux n =
  let (f, fPrec) = fibAux (n-1)
  in (f + fPrec, f)
-- mantenere l'interfaccia giusta per fib!
fib n = fst (fibAux n)
```

clausola where: fibonacci efficiente

Tuttavia, un **Vero Programmatore Haskell**, darebbe un'altra definizione, che fa leva su un'altra clausola: `where`.

La clausola where è ispirata al tipico **linguaggio matematico**, quando si definisce qualcosa, lasciandosi la libertà di definire qualcosa più tardi: "Sia $f = h \circ g$, **dove** h e g sono…"

Attenzione però alle **differenze**: `where` **non è un'espressione**, ma **qualifica dei valori** nella **parte destra** di una definizione.

```
myFst z = x where (x, _) = z

fibAux' 0 = (0,0)
fibAux' 1 = (1,0)
fibAux' n = (f+fPrec, f) where
    (f, fPrec) = fibAux' (n-1)

fib' n = fst (fibAux' n)

-- attenzione: where non è
-- un espressione standalone come let
> x x where x = \x -> x
<interactive>:94:5: parse error on input ‘where’
```

Costruttori di Tipo: Liste

Nella tradizione dei Linguaggi Funzionali, c'è predefinito il costruttore di tipo **lista: `[]`**.

Il **cons** (**inserzione in testa**) è un operatore infisso e si scrive **`:`**. Si chiama cons da '**constructor**', tradizione risalente al **Lisp**.

La scrittura `[1,2,3]` è zucchero sintattico per `1:2:3:[]`.

Le liste di caratteri (**stringhe**) invece posso usare i doppi apici.

```
 -- la costante lista vuota ha tipo polimorfo
>:t []
[] :: [t]
 -- il cons si scrive : (infisso)
>:t (:)
(:) :: t -> [t] -> [t] -- NB: 2do arg. è una lista!
 -- notazioni abbreviate per le liste
> 1:2:3:[]==[1,2,3]
True
> ”tpfi” = [’t’, ‘p’, ‘f’, ‘i’]
True
```

Amenità (istruttive) sulle liste

Vediamo un po' di istruttivi esempi di liste e tipaggi, che fanno capire alcune caratteristiche di Haskell.

Le liste si possono annidare. Attenzione a qualche tipaggio.

Posso avere anche liste di funzioni!

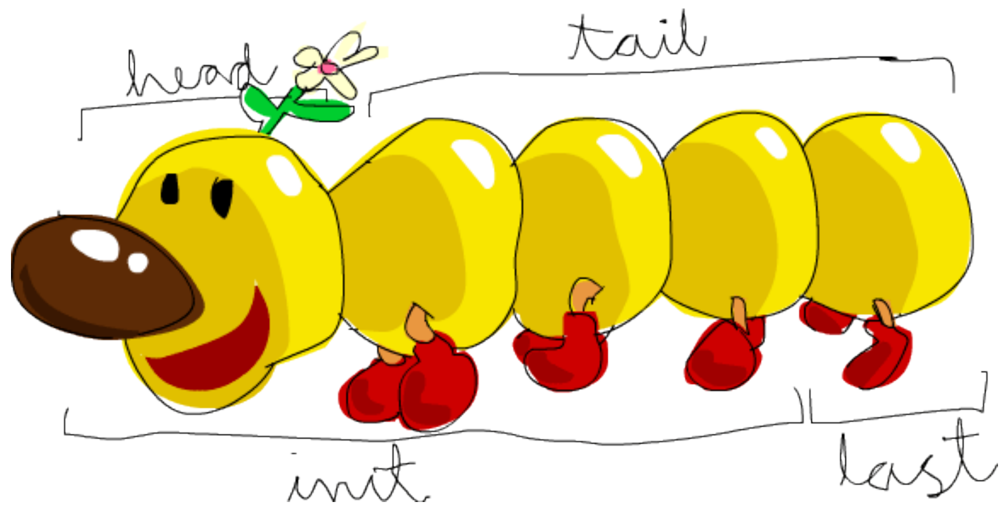
```
-- attenzione alla lista vuota!
>: t [[], [[]]]
[[], [[]]] :: [[[t]]]
-- la lista vuota è [t] per ogni t!

>:t [undefined]
[undefined] :: [t]

>:t [undefined, undefined]
[undefined, undefined] :: [t]

>:t [undefined:undefined]
[undefined:undefined] :: [[t]]

>:t [(+), (*)]
[(+), (*)] :: (Num a) => [a -> a -> a]
```



Lezione 3b:

Programmazione su Liste in Haskell

Definizioni di Funzioni su Liste

Tutte le liste hanno la forma **`x:xs`**, `x` **testa** e `xs` **coda**: gli Haskellioti usano un nome che finisce in s (**plurale**) come memo per dire che una variabile rappresenta una lista.

Usiamo questo fatto per scrivere funzioni per **pattern matching**.

Attenzione! l'applicazione di **funzione associa sempre più di tutto** e quindi `f x:xs` viene inteso `(f x):xs` e non `f (x:xs)`
Occorre mettere le parentesi se necessario.

```
 -- testa e coda si scrivono facilmente
testa (x:_) = x
coda (_:xs) = xs
> testa [1,2,3]
1
>:t testa
testa :: [t] -> t
> coda [1,2,3]
[2,3]
>:t coda
coda :: [t] -> [t]
```

```
 -- non c'è una clausola per lista vuota
 -- in testa e coda, infatti:
 -- ci sono argomenti che sfuggono:
> testa []
*** Exception: lezione3.hs:45:1-15:
Non-exhaustive patterns in function testa
-- mentre:
head []
*** Exception: Prelude.head: empty list
L'eccezione appare col suo significato logico
```

Last, Null, e Init

Vedremo la definizione di diverse funzioni che sono in realtà predefinite…

null verifica se una lista è vuota, **last** restituisce l'ultimo elemento, e **init** la lista menol'ultimo elemento.

```
nullBeginner []     = True
nullBeginner (x:xs) = False

myNull :: [a] -> Bool
myNull [] = True
myNull  _ = False

myLast :: [a] -> a
myLast [x] = x
myLast (_:xs) = myLast xs

myInit :: [a] -> [a]
myInit   [x]  = []
myInit (x:xs) = x: init xs
```

Pillole di laziness

```
-- Null è stretta, richiede
la valutazione del parametro…
> null undefined
*** Exception: Prelude.undefined

-- … ma solo fino a determinare
che si tratta di una lista
> null (undefined:undefined)
False
```

```
> last [1, 2, 3]
3
> init [1, 2, 3]
[1,2]
```

sum, length, maximum & minimum

Altre piccole funzioni, per prendere confidenza col mezzo.

sum sommatoria di una lista, **length** calcola la lunghezza, **maximum** (**minimum**) calcolano il massimo (minimo).

```
sum :: (Num a) => [a] -> a
mySum   []   = 0
mySum (x:xs) = x + mySum xs

length :: [a] -> Int
myLength [] = 0
myLength (x:xs) = 1 + myLength xs

and :: [Bool] -> Bool
myAnd [] = True
myAnd (x:xs) = x && myAnd xs

minimum :: (Ord a) [a] -> a
myMinL   [x]  = x
myMinL (x:xs) = min x (myMinL xs)
```

```
-- and di lista vuota?
> and []
True
> or []
False
> t min
min :: (Ord a) -> a -> a -> a
-- minimum di lista vuota?
> minimum []
*** Exception: Prelude.minimum:
empty list
```

++ e definizione di operatori

La giustapposizione di liste si scrive ++. Vediamo come si possono definire operatori.

++ è associativa:

```
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
```

```
(++) :: [a] -> [a] -> [a]
-- si fa ricorsione sul primo parametro
  []   ++ ys = ys
(x:xs) ++ ys = x:(xs ++ xs)

-- ma anche:
(++)  []   ys = ys
(++)(x:xs) ys = x:(xs ++ xs)

-- osservate e spiegate:
-- undefined ++ [1,2] = undefined
> undefined ++ [1,2]
***Exception: Prelude.undefined
-- ma: [1,2] ++ undefined = [1,2:undefined]
```

Pillole di laziness

reverse e palindromìa

Vediamo il rovesciamento di una lista: `reverse`.

Ma qual è la **complessità** di **`reverse`**? Quale versione usereste?

Gli haskelioti amano scrivere funzioni **senza decomporre strutture dati**, ma **componendo funzioni predefinite** (**one-liner**)

In Haskell, **`palindrome`** è sempre **corretta**.
Dimostrare (per **Esercizio** ☺)

```
reverse :: [a] -> [a]
myReverse [] = []
myReverse (x:xs) = myReverse xs ++ [x]
--oppure:
myReverse' [] = []
myReverse' (x:xs) =
   myReverse' (init xs) ++ last xs


--facciamo conoscenza con gli 'one-liner'
palindrome :: [a] -> Bool
palindrome xs = xs == reverse xs
```

osservatori di una lista & specifiche

Ora il giochino opposto: io do tipi, "specifica a parole ed esempi" e proprietà algebriche… e voi scrivete il codice (**Esercizio**)

`!!` seleziona un elemento in una certa posizione, **`take`** prende la prima parte di una lista, **`drop`** la seconda, **`splitAt`** fa una coppia con prima e seconda parte.

```
(!!) :: [a] -> Int -> a
> [1,2,3,4]!!0
1
> [1,2,3,4]!!3
4
> [1,2,3,4]!!4
*** Exception:Prelude.(!!):
index too large
```

```
take :: Int -> [a] -> [a]
> take 2 [1,2,3,4]
[1,2]
-- non si scompone se prendo "troppo"
> take 5 [1,2,3,4]
[1,2,3,4]
> take 2 (1:2:[]++undefined) -- lazy
[1,2]
> take 3 (1:2:[]++undefined)
*** Exception: Prelude.undefined

drop :: Int -> [a] -> [a]
splitAt :: Int -> [a] -> ([a], [a])
```

```
xs = take n xs ++ drop n xs
splitAt n xs = (take n xs, drop n xs)
```

'as pattern', liste ordinate & merge

Vediamo la classica funzione `merge` che **fonde due liste ordinate in una lista ordinata** come suona in Haskell.

Notare l'**as pattern** `xs@(x:txs)` che permette nella parte destra di nominare testa (`x`), coda (`txs`) e tutta la lista (`xs`)

Esercizio: scrivere **`mergeSort`** scorrendo una sola volta la lista (ovviamente `length` e `splitAt` devono entrambe scorrere la lista).

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge xs@(x:txs) ys@(y:tys)
    | x <= y    = x:merge txs ys
    | otherwise = y:merge xs tys

mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs  = mergeSort ls `merge` mergeSort rs
   where (ls, rs) = splitAt (length xs `div` 2) xs
```

Annidamento di liste

Le liste si possono **annidare**: è facile avere le **liste di liste**.

Scriviamo la funzione **lol** che sostituisce ogni elemento di una lista con la lista contenente quell'elemento…

… oppure la lista dei **suffissi** (ancora con as-pattern).

Ovviamente ci sono anche le liste di coppie… ecco lo **zip**.

Esercizi: e i prefissi? e la funzione inversa **unzip**?

```
zip :: [a] -> [b] -> [(a, b)]
-- si ferma quando finisce la più corta
myZip (x:xs) (y:ys) = (x, y):myZip xs ys
myZip [] xs = []
myZip xs [] = []

> zip [1,2,3] [‘a’,’b’,’c’,’d’]
[(1,’a’), (2,’b’), (3, ‘c’)]
```

```
lol :: [a] -> [[a]]
lol (x:xs) = [x]:lol xs
lol [] = []

> lol [1,2,3]
[[1],[2],[3]]

suffissi :: [a] -> [[a]]
suffissi xs@(_:txs) = xs:suffissi txs
suffissi     []     = []

> suffissi [1,2,3]
[[1,2,3],[2,3],[3]]
```

concat vs sum

La funzione **concat** 'scioglie' una lista di liste in un'unica lista, giustapponendo i valori.

Notate l'**estrema somiglianza** tra il codice di **concat** e quello di **sum** che sarà il tema della prossima lezione.

Basta sostituire `0` con `[]`, `++` con `+` etc.

Esercizio: dimostrare che **`myConcat (lol xs) = xs`**.

```
concat :: [[a]] -> [a]
myConcat (xs:xss) = xs ++ myConcat xss
myConcat [] = []

> myConcat [[1,2][3,4][2]]
[1,2,3,4,2]
```

That's all Folks…

Grazie per l'attenzione…

…Domande?