

Paradigma funzionale

Linguaggio Haskell

Paradigmi imperativo e funzionale

Paradigma **funzionale** si contrappone al paradigma **imperativo** dove:

- l'esecuzione programma comporta l'esecuzione di una sequenza di istruzioni
- istruzione base, modifica di una variabile, locazione di memoria

Paradigma imperativo usato nelle implementazioni hardware, linguaggi macchina

- macchina di von Neumann

Gran parte dei linguaggi di programmazione usano il paradigma imperativo

- linguaggi di von Neumann
- linguaggi ad oggetti

Paradigma funzionale

Paradigma di programmazione:

- dichiarativo
 - il programma consiste nella definizione di una serie di funzioni e di costanti

Differenze rispetto all'imperativo;

- assenza di stato
 - non si accede direttamente alla memoria (store)
 - non esistono variabili modificabili
- diverso meccanismo di computazione
 - non si esegue una di sequenza di istruzione
 - si valuta un'espressione
 - dal punto di vista astratto: sequenza di riscritture
 - concreto: macchina a stack, attivo tanti RdA

Data la definizione:

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

la valutazione di `fatt 3` porta a valutare nell'ordine:

```
if 3 == 0 then 1 else 3 * (fatt (3 - 1))
3 * (fatt 2)
3 * (if 2 == 0 then 1 else 2 * (fatt (2 - 1)))
3 * 2 * (fatt 1)
...
```

passi di computazione:

- sostituisco una chiamata di funzione con il suo corpo istanziato
 - `fatt 2` diventa `if 2 == 0 then 1 else 2 * (fatt (2 - 1))`
- applico le funzioni base
 - `3 - 1` diventa `2`

Meccanismo di valutazione implementato

Uso degli ambienti, la valutazione di:

```
fatt 3
```

diventa la valutazione dell'espressione

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

nell'ambiente

```
n ==> 3
```

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

che a sua volta per essere valutata, porta alla valutazione di

```
n == 0
```

nell'ambiente

Si crea un pila di espressioni da valutare, con relativi ambienti di valutazione.

Struttura complessa:

- macchina SECD di Landin (1964)
- base per le implementazioni attuali

Paradigma funzionale

- l'assenza di stato comporta l'assenza di cicli (`while`)
 - la valutazione di un'espressione porta sempre allo stesso risultato, non ha senso ripeterla
 - cicli sostituiti da funzioni ricorsive, si valuta la stessa funzione con argomenti diversi,
- funzioni oggetti del primo ordine,
 - funzioni di ordine superiore
funzioni come argomento o risultato di altre funzioni
 - funzioni senza nome
`(lambda (x) (+ x 1))`
espressioni che rappresentano una funzione
- polimorfismo
 - riuso del codice attraverso funzioni che si adattano a tipi di dato diversi
 - type checking dinamico: Scheme
 - type checking statico: Haskell, schemi di tipo, type checking sofisticato

- L'assenza di stato semplifica il ragionamento
 - non è possibile l'aliasing
- Un esempio di errore creato dall'aliasing, modello per riferimento, errore che era presente in una libreria Java)
 - Classe per l'accesso protetto alle risorse
 - accedo alla risorsa con il metodo `useTheResource`
 - il metodo controlla che solo gli utenti nella lista `allowedUsers` possano accedere
 - è possibile ricevere la lista `allowedUsers` per controllare la propria priorità.

```
class ProtectedResource {
    private Resource theResource = ...;
    private String[] allowedUsers = ...;
    public String[] getAllowedUsers() {
        return allowedUsers;
    }
    public String currentUser() { ...}
    public void useTheResource() {
        for(int i=0; i < allowedUsers.length; i++) {
            if(currentUser().equals(allowedUsers[i])) {
                ...// access allowed: use it
            }
        }
        return;
    }
    throw new IllegalAccessException();}
}
```

Far restituire da `getAllowedUsers()` una copia della lista interna

```
public String[] getAllowedUsers() {  
    String[] copy = new String[allowedUsers.length];  
    for(int i=0; i < allowedUsers.length; i++)  
        copy[i] = allowedUsers[i];  
    return copy;  
}
```

In programmazione funzionale

- il problema non si pone
- accedo sempre ad una copia

Presentazione del paradigma attraverso un esempio concreto

- Haskell

Confronto con Scheme, funzionale ma con scelte diametralmente opposte

- Origini

- Scheme: uno dei tanti dialetti Lisp (1958, John McCarthy) come Clojure, Common Lisp, Emacs Lisp, Standard Lisp, Racket
- Haskell: ML (1973, David Milner, Edimburgo), come Standard ML, Caml, OCaml,

- Sintassi
 - Scheme: pochi costrutti, semplici a scapito delle concisione del codice, tante parentesi
 - Haskell: non molti costrutti base, molto zucchero sintattico, programmi molto sintetici,
- Esempio, crivello di Eratostene in Haskell

```
primes = filterPrime [2..]  
where filterPrime (p:xs) =  
      p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

Sistema di tipi:

- Scheme: controllo di tipo **dinamico**
- Haskell: controllo **statico**
 - inferenza di tipo
- entrambi fortemente tipati

Meccanismo di valutazione:

- Scheme: linguaggio **eager, call-by-value** (passaggio per valore)
 - gli argomenti di una funzione vengono valutati prima di essere passati nel corpo,
 - valuto tutto appena possibile
- Haskell: **lazy, call-by-need** (passaggio per nome)
 - gli argomenti di una funzione vengono passati così come stanno
 - valuto un argomento se proprio non posso farne a meno
 - posso gestire strutture dati infiniti, stream

Confronto eager – lazy

- esistono espressioni che convergono con la valutazione lazy, e divergono nella eager
- ma non il viceversa
- sui tipi semplici, se convergono, convergono sullo stesso valore
- valutazione eager più semplice da implementare
- valutazione lazy, implementata in maniera semplice, porta a valutare lo stesso argomento molte volte con perdite di efficienza

- Vantaggi

- evito lavoro inutile `fst (2, (fact 1000))`
- più programmi terminano `fst (2, (diverge 0))` `diverge x = diverge x`
- più programmi non generano errori `fst (2, error)`

- Svantaggi

- potenziale ripetizione delle esecuzione
 - `(\x -> x + x) (fact 1000)`
 - lo si evita con il [call-by-need](#), valuto argomenti replicati una volta sola
- espressioni con side-effect complicate da gestire
 - side-effect modifico lo stato: memoria, I/O
 - se la valutazione di un argomento provoca un side effect, diventa difficile capire quando questo avviene
 - soluzione Haskell funzionale puro

In un linguaggio funzionale, soprattutto eager, è importante definire

- l'insieme dei valori le espressioni che si considerano essere completamente valutate
in Scheme:
 - le costanti numeriche 1, 2, ... sono valori, tutti le alte espressioni di tipo intero no
 - coppia (e1, e2) è un valore se e solo se le sue componenti, e1, e2, sono valori
 - le lambda astrazioni sono valori
(lambda (x) (+ 3 1)) funzione costante
(lambda () (+ 3 1)) funzione senza argomento, **thunk**
 - meccanismo per sospendere la valutazione
 - ...

- Purezza

- Scheme: non e' un linguaggio funzionale puro
esistono effetti collaterali, posso modificare lo stato

```
(define b 1)
(define succ (lambda (x) (+ x b)))
(set! b 5)
```

- Haskell puramente funzionale
come (quasi) tutti i linguaggi lazy
- la gestione dei side-effect nei linguaggi lazy più complessa,
non è chiaro in quale istante un'espressione è valutata e quante volte
problematico se la valutazione ha effetti collaterali

- in generale
 - in Haskell molti dei concetti di Scheme, ma sotto una veste diversa
 - Haskell più sofisticato, nuovi costrutti, meccanismi per definire programmi

- Sito haskell.org, dispone di tutte le informazioni
 - tutorial: [A gentle introduction to Haskell 98](#) , qualche discrepanza con la versione attuale Haskell2010, (sistema di tipi più ricco) non completamente retrocompatibile
 - download: ghc: Glasgow Haskell Compiler (linguaggio compilato)
 - ghci: un interprete Haskell, esecuzioni interattiva.

Tipi scalari:

- Numerici: varie classi di numeri

```
5 - 5  :: 5  :: Num t => t  --- un generico tipo numerico
5.0   :: Fractional t => t --- un generico tipo frazionario
pi    :: Floating a => a   --- un generico tipo floating point
```

sistemi di tipi più complesso rispetto a Haskell98, tutorial diverse "classi" di tipi numerici.

- Overloading, la giusta interpretazione è scelta dal contesto

```
(rem 7 4) :: Integral a => a  --- generico tipo intero
1 + 2    :: Num a => a
(1 + 2) + pi      --- accettato, a (1+2) associato un tipo
(rem 7 4) + pi    --- genera errore, Integral incompatibile
```

Sistema di inferenza dei tipi:

- non serve (quasi mai) dichiarare il tipo di un identificatore
- il compilatore (interprete) lo capisce dal contesto, in maniera piuttosto complessa.
 - algoritmo di inferenza di tipi
cerca lo schema di tipo più generale associabile ad un espressione
- `ghci` fornisce il tipo di un'espressione e con `:type` e

```
Prelude> :type (rem 7 4)  
(rem 7 4) :: Integral a => a
```

- Caratteri (Unicode set)

```
'a' :: Char
```

- Boolean

```
True False :: Bool
```

- enuple di tipi diversi

```
('b', 5, pi) :: (Num t, Floating t1) => (Char, t, t1)
('b', ((5, pi), (1,2,3))) --- naturalmente posso iterare
```

- esistono destruttori ma solo per le coppie

```
fst (2,4)
snd (2,4)
fst (2,3,5) --- genera errore di tipo
```

- le componenti di altre enuple selezionate per pattern matching

Sequenze di elementi dello stesso tipo:

```
[1,2,4] :: Num t => [t]
```

```
[1,2,pi] :: Floating t => [t]
```

```
[1,2,'a'] --- type error
```

```
['a','b','d'] == "abd" --- :: [Char] stringhe identificate
```

- liste formate da due costruttori

```
[] :: [a] --- lista vuota, nil di Scheme
```

```
(:) :: a -> [a] -> [a] --- concatenazione, infisso, cons
```

```
1:2:4:[] --- senza parentesi, (:) associa a destra
```

```
(1:(2:(4:[]))) ---- posso aggiungere parentesi, a differenza
```

```
1:2:4:[] == [1,2,4] --- zuccherare sintattico
```

```
'a': 'b': 'c': [] == "abc" --- zucchero sintattico
```

- costruttori per liste di sequenze aritmetiche, presente in Python, e altri linguaggi

```
[1..9]      --- definizioni compatte  
[1,3..20]  --- schema più sofisticato  
           --- definisce una progressione aritmetica
```

- distruttori: `head`, `tail`, equivalente di `car`, `cdr`, ma con nomi accettabili

```
head [2,4,6]  
tail [2,4,6]  
head (tail [2,4,6])
```

- posso costruire liste di liste, ma tipi omogenei

```
[[1,2], [1,2,3], [1]] :: [[Integer]]  
[[1,2], [1,2,3], 1]  --- genera errore di tipo
```

Tipi definiti dall'utente

Meccanismo piuttosto sofisticato per definire un ricco insieme di tipi

- Tipi Enumerazione

```
data Colore = Rosso | Verde | Marrone
y = Rosso
```

- Haskell case sensitive,

- nome dei tipi, costruttori, costanti: iniziano per maiuscola
- nome variabili (normali o di tipo), funzioni: iniziano per minuscola

```
data Colore = rosso | Verde | Marrone --- genera errore
data colore = Rosso | Verde | Marrone --- genera errore
```

- tipo Bool enumerazione predefinito (ma ridefinibile)

```
data Bool = True | False
data NBool = True | False
```

- dichiarazioni multiple: non ammesse nello stesso ambiente
possibili in un nuovo ambiente locale
con l'interprete ogni riga di input genera un nuovo ambiente locale

Tipi record, con costruttori

```
data IntPoint = IPt Integer Integer  
IPt 3 6
```

- definisco un tipo con più componenti, alle singole componenti accedo in base alla posizione,
- mancano le etichette sulle componenti, ma posso introdurle con sintassi alternativa
- etichetto la struttura IPt

Tipi unione, più alternative

```
data Point23Dim = Pt2 Integer Integer  
                | Pt3 Integer Integer Integer
```

```
Pt2 1 2
```

```
Pt3 1 2 3
```

- definisco un elemento che può avere 2 o 3 componenti,
- più strutture alternative
- un costruttore per ogni alternativa

Tipi parametrici

- Tipi parametrici, un tipo che dipende da un altro tipo

```
data Point a = Pt a a
```

- a parametro, variabile di tipo
- Pt costruttore generico di tanti tipi diversi
- generic (parametrized) types in Java, implementano idee simile ma con tipi espliciti

```
Pt 2 3      :: Num a => Point a  
Pt 'a' 'b'  :: Point Char
```

- Point: **type constructor**
- Pt : **data constructor**

Java Generic Classes

Tipi parametrici analoghi ai tipi generici in Java

```
class Point<T> {  
    private T x, y;  
  
    public void pt(T x, T y) {  
        this.x = x;  
        this.y = y; }  
  
    public T getX() {  
        return x; }  
  
    public T getY() {  
        return y; } } }
```

Haskell non serve istanziare esplicitamente i tipi parametrici, in Java sì.

Posso anche definire:

```
data Point a = Point a a
Point 2 3 :: Point Integer
```

- uso lo stesso nome per due oggetti in categorie sintattiche differenti,
- **name set** diversi, il contesto dirime l'ambiguità

Viste come caso particolare di tipo record parametrico

```
(3, 'a', ['b']) :: Num t => (t, Char, [Char])
```

- nota virgola per separare i campi,
- parentesi tonde, obbligatorie, con significato diverso dal solito

Equivalente alla definizione

```
data ( , , ) a b c = ( , , ) a b c
```

- più zucchero sintattico ad hoc.

Posso definire enuple di lunghezza arbitraria

- concettualmente: tanti costruttori
 - ognuno con un numero di argomenti differenti
 - con struttura simile

Tipi ricorsivi (parametrici)

Definizione ricorsive sono lecite

```
data Tree a    = Leaf a | Branch (Tree a) (Tree a)
data List a    = Nil     | Cons  a (List a)
```

```
Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)) :: Tree Integer
Cons 'a' (Cons 'z' (Cons '8' Nil))      :: List Char
```

Definizione di un nuovo tipo T

- vengono definiti un insieme di **costruttori**
- costruttori: funzioni che restituiscono elementi di T.
- di ogni costruttore viene definito il dominio, la lista degli argomenti
- definizioni parametriche rispetto ad uno o più tipi a, b
- definizioni ricorsive sono lecite

Tipi di dato ricorsivi in Scheme.

Usando le liste (arbitrarie) di Scheme posso definire diversi modi per rappresentare dati di tipi ricorsivi

- un approccio canonico

```
(define (Leaf a) (list 'LTree a))
(define (Branch tl tr) (list 'BTree tl tr))
(define (leftSon t) (car (cdr t)))
(define (BTree-? t) (eq? (car t) 'BTree))
```

- definisco
 - i costruttori Leaf Branch
 - i distruttori leftSon rightSon
 - funzioni di controllo BTree
- nessuna definizione di tipo o controllo di tipo
 - il programmatore usa uno schema canonico per definire gli oggetti
 - controlli a tempo di esecuzione

Tipi predefiniti come caso particolare

Tipi di dato predefiniti sono quasi tutti un'istanza dello schema generale di definizione di tipo:

- `Boolean`: tipo di dato predefinito in maniera perfetta
- `Enum`, `Liste`: predefiniti ma usano una sintassi ad hoc
- `Int`: seguono il pattern base, ma contengono troppi elementi per essere definibili in maniera standard
- `Float`: non sono tipi di dato ricorsivi, non sono ordinali

Equazioni di tipo, equivalenze tra tipi

- Posso dare un nome ad espressioni di tipo

```
type String = [Char]    --- predefinito
```

```
type Person = (Name, Address)
```

```
type Name = String
```

```
data Address = None | Addr String
```

```
type IntTree = Tree Integer
```

```
type BinaryFunction a = a -> a -> a
```

```
type BinaryIntFunction = BinaryFunction Integer
```

- **Equivalenza strutturale** tra i tipi:

- [String] e [[Char]] sono espressioni di tipo equivalenti.
- come Person -> Name e (Name, Address) -> Name e (Name, Address) -> [Char]

Nota: tipi ricorsivi con nomi diversi, con stessa struttura, non sono equivalenti

```
data Point1 = Point1 Integer Integer  
data Point2 = Point2 Integer Integer
```

usano, per i costruttori, nomi differenti

Posso associare valori a variabili

$$x = 5 + 3$$

- legame immutabile, identificatori (variabili, funzioni) non possono essere ridefiniti se non in ambienti locali, mascheramento
 - o usando l'interprete: ogni interazione con l'interprete genera un nuovo ambiente locale
- legame **lazy, per nome**
a x associo l'espressione $5 + 3$ **senza valutarla**
- l'ordine non ha importanza

$$x = y + 3$$
$$y = 0$$

- possibili definizioni mutuamente ricorsive

$x = y + 3$

$y = x - 3$

$z = z + 1$

- il programma diverge solo quando viene forzata la valutazione di x
o y o z
legame lazy

Le dichiarazioni

```
x = 5
```

```
succ x = x + 1
```

Definiscono legame immutabile,

- gli identificatori non possono essere ridefiniti,
- validi su tutto il programma.

Utile avere ambienti locali,

- variabili locali, funzioni ausiliare, per definire altre funzioni.

Due costruttori di ambienti locali

let

```
y = 5
let y = a + 3
    a = 5
in y + a
```

- let definisce un'espressione con ambiente locale
- mutua ricorsione
- nell'ambiente locale posso ridefinire le variabili

where

```
y = 2 + z
where z = x*x
```

- usata solo dopo una definizione
valori ausiliari necessari per la definizione principale
- post-fisso, mutua ricorsione

Oggetti del primo ordine

- lambda astrazione, per costruire funzioni nameless

```
\ x -> x + 1
```

il simbolo `\` ricorda la lettera greca lambda λ

- definizione di funzioni (associamo un identificatore ad una funzione)
 - due alternative

```
inc x = x + 1
```

```
inc = \ x -> x + 1 --- definizione equivalente
```

- definizione per casi, si applica il primo caso lecito

```
fact 0 = 1
```

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

```
fib 1 = 1
```

```
fib 2 = 1
```

```
fib n = fib (n-1) + fib (n-2)
```

```
fibAux 1 = (0, 1)
```

```
fibAux n = (snd prefPair, fst prefPair + snd prefPair)  
  where prefPair = fibAux(n-1)
```

```
fibAux2 1 = (0, 1)
```

```
fibAux2 n = (j, i+j)
```

```
  where (i, j) = fibAux2(n-1)
```

Currying

- da Haskell Curry,
- funzioni su più argomenti sono viste come funzioni che restituiscono funzioni
- isomorfismo tra $(A * B) \rightarrow C$ e $A \rightarrow (B \rightarrow C)$
- currying significa passare dalla prima alla seconda rappresentazione dello spazio di funzioni

Esempio: definisco una funzione somma come (tre definizioni equivalenti):

```
add = \ x -> ( \ y -> (x + y) )
```

```
add = \ x y -> x + y
```

```
add x y = x + y
```

il currying permette

- una gestione elegante di funzione a più argomenti, non serve costruire coppie di valori
- sintetizza la definizione di alcune funzioni

```
succ = add 1
```

```
succ = \x -> add 1 x
```

- per i nostalgici, la versione uncurried è comunque possibile

```
uncurriedAdd :: Num a => (a, a) -> a
```

```
uncurriedAdd (x, y) = x + y
```

```
dodici = uncurriedAdd (5,7)
```

- notare che le parentesi sono quelle dei tipi enupla,
- non quelle che racchiudono i parametri di una procedura

Regole sintattiche per evitare parentesi

- per applicare una funzione e sufficiente la giustapposizione
 - $f\ x$ l'equivalente della notazione matematica $f(x)$
 - o nella sintassi di Scheme ($f\ x$)
 - può essere usata in Haskell,
posso inserire un numero arbitrario di parentesi forzano solo l'ordine di valutazione
- l'applicazione associa a sinistra
 - $add\ x\ y$ abbreviazione per $(add\ x)\ y$
 - posso anche scrivere $(add\ x\ y)$, come in Scheme,
o $((add\ x)\ y)$
- nelle espressioni di tipo, la freccia \rightarrow , associa a destra
 $add :: Integer \rightarrow Integer \rightarrow Integer$
abbreviazione per
 $add :: Integer \rightarrow (Integer \rightarrow Integer)$

Funzioni definite per casi (pattern matching)

- modo comodo e elegante per definizioni chiare e sintetiche di funzioni
- zucchero sintattico, si può ridurre ad altri meccanismi base, costruito `case`

```
head (x:xs) = x
tail (_:xs) = xs
```

- wildcard `_`, sostituisce le variabili nei pattern,
 - rende esplicito il **non uso** della variabile,
 - tutti i wildcard sono considerati variabili distinte

```
empty :: [a] -> Bool
empty (_:_) = False
empty []    = True
```

Esempi

```
length :: [a] -> Integer
length [] = 0
length (_:xs) = 1 + length(xs)
```

```
map :: (a->b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

```
squares = map (\x -> x * x) [1..10]
```

Cosa contiene `map add [1..10]`?

Alcuni funzionali standard

- `curry`: trasforma una funzione binaria nella sua versione curryficata

```
curry :: ((a, b) -> c) -> a -> b -> c
```

```
curry f = \ x -> \y -> f (x, y)
```

```
curry f = \ x y -> f (x, y)
```

```
curry f x y = f (x, y)
```

- operazione inversa

Alcuni funzionali standard

- `curry`: trasforma una funzione binaria nella sua versione curryficata

```
curry :: ((a, b) -> c) -> a -> b -> c
```

```
curry f = \ x -> \ y -> f (x, y)
```

```
curry f = \ x y -> f (x, y)
```

```
curry f x y = f (x, y)
```

- operazione inversa

```
uncurry :: (a -> b -> c) -> (a, b) -> c
```

```
uncurry f = \ (x, y) -> f x y
```

```
uncurry f (x, y) = f x y
```

- `compose`, composizione di due funzioni

Alcuni funzionali standard

- `curry`: trasforma una funzione binaria nella sua versione curryficata

```
curry :: ((a, b) -> c) -> a -> b -> c
```

```
curry f = \ x -> \ y -> f (x, y)
```

```
curry f = \ x y -> f (x, y)
```

```
curry f x y = f (x, y)
```

- operazione inversa

```
uncurry :: (a -> b -> c) -> (a, b) -> c
```

```
uncurry f = \ (x, y) -> f x y
```

```
uncurry f (x, y) = f x y
```

- `compose`, composizione di due funzioni

```
compose :: (b -> c) -> (a -> b) -> a -> c
```

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

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

predefinita . in notazione infissa `f . g`

Meccanismo di valutazione, passaggio dei parametri

Valutazione lazy, call-by-need

- argomenti di funzioni e valori associati a variabili, **non vengono valutati**, al momento del passaggio dei parametri, definizione, si passa la loro **closure** (espressione, ambiente di valutazione)
- valutazione avviene all'interno del corpo, se necessario

La definizione

$v = 1/0$

- associa a v l'espressione $1/0$
- nessun errore fino a che si forza la valutazione di v
- esempio scrivo, nell'interprete, `v 0 if v < 0 then 0 else 1`
 - `if v < 0 then 0 else 1` viene passato come argomento ad una funzione `show` che determina la stringa della sua rappresentazione
 - i singoli caratteri di `show (if v < 0 then 0 else 1)` vengono valutati e stampati

- la valutazione differita, anche per i costruttori di dato, permette di definire **dati potenzialmente infiniti**,
- di volta in volta viene valutata solo la parte richiesta dal resto del codice
per esempio, per applicare pattern matching

```
ones = 1 : ones :: Num a => [a]
```

- definizione ricorsiva,
- viene espansa solo quando necessario, per esempio esamino il secondo elemento di `ones`

```
head(tail(tail ones))
```

Costruzione della lista di tutti i naturali

```
numsFrom n = n : numsFrom (n + 1)  
nums = numsFrom 0
```

Costruzione della lista dei quadrati

```
squares = map (\x -> x^2) (numsFrom 0)
```

La sequenza di Fibonacci, vista come stream,

```
addList (x:xs)(y:ys) = (x + y) : addList xs ys
```

```
fib = 1:(addList fib (0 : fib))
```

- calcolata in maniera efficiente grazie alla meccanismo call-by-need, `fib` replicata ma valutata una volta sola
- corretto perché la somma della sequenza di Fibonacci con la sua traslata, di una posizione a destra, restituisce la sequenza di Fibonacci senza il primo elemento (correttezza non ovvia)

Estrarre parte finite da dati lazy

- funzione che seleziona la parte iniziale finita (funzione predefinita nel Prelude)

```
take 0 _ = []
```

```
take _ [] = []
```

```
take n (x:xs) = x : take (n-1) xs
```

- utile per interagire con l'interprete `ghci`
 - quando scrivo `exp` l'interprete forza la valutazione completa di `show exp`:
 - `exp` viene convertito in una stringa
funzione `show`
 - ogni elemento della stringa viene stampato
 - se inserisco `ones` stampa la sequenza infinita di 1
- funzione per selezionare l'iesimo elemento di uno stream

```
takeEl 0 (x:_) = x
```

```
takeEl n (_,xs) = takeEl (n-1) xs
```

Stream in linguaggi funzionali eager

Attraverso l'uso di **thunk** posso rappresentare strutture dati infinite in linguaggi eager.

Gli esempi precedenti tradotti in Scheme come:

```
(define (numsFrom n) (lambda () (cons n (numsFrom (+ n 1)))))
```

```
(define nums (numsFrom 0))
```

```
(define (take n xs)
  (cond
    [(equal? n 0) null]
    [(equal? (xs) null) null]
    [true (cons (car (xs)) (take (- n 1) (cdr (xs))))]))
```

```
(take 20 nums)
```

La funzione `map` sugli stream di Scheme diventa:

```
(define (lazymap f xs)
  (if (equal? (xs) null)
      xs
      (lambda () (cons (f (car (xs)))
                        (lazymap f (cdr (xs)))))))

(take 20 (lazymap (lambda (x) (* x x)) nums))
```

Sintassi infissa

- Diverse funzioni predefinite usano la sintassi infissa

+ * ; ^ -

- ne esistono altre:

- ++ concatenazione di stringhe
- . composizione di funzioni $f . g = \lambda x \rightarrow f (g x)$

- le operazioni non rappresentate da identificatori standard usano la forma infissa

(#) $a b = \text{rem } a b$

(\\) $a b = a + b$

- possibile specificare tipo di associatività e priorità

```
infixl 9 #    --- infisso, assoc. a sinistra, priorità 9
infixr 5 \\   --- infisso, assoc. a destra, priorità 5
7 # 4 # 3 \\ 2 \\ 5 --- ((7 # 4) # 3) \\ (2 \\ 5)
```

Passaggio tra notazioni infissa – prefissa

Le parentesi () permettono di passare alla notazione prefissa

`(+) :: Num a => a -> a -> a`

- `(+)` coincide con la funzione `\ x y -> x + y`

inoltre

- `(+ 4)` è zucchero sintattico per `\ x -> x + 4`
- `(5 +)` è zucchero sintattico per `\ y -> y + 5`

Gli apici permettono la trasformazione opposta:
scrivere una funzione binaria in forma infissa

`5 `add` 4`

- chiave di lettura: c'è uno scambio di posizione tra
funzione e primo argomento

`add3Values x y z = x + y + z :: Num a => a -> a -> a -> a`
`sei = (1 `add3Values` 2) 3`

List comprehension

- liste sono pervasive di Haskell, utile avere zucchero sintattico

```
[ f x | x <- xs ]
```

applico la funzione f a tutti gli elementi della lista xs ,

si richiama una notazione insiemistica

```
[ x + 1 | x <- [1..6] ]  
map (+1) [1..6]
```

La parte $x <- xs$ viene detta **generatore**,

possibili più generatori, si prendono tutte le combinazioni

```
[ (x * y) | x <- [1..10], y <- [1..10] ]  
[[ (x * y) | x <- [1..10]] | y <- [1..10] ]
```

posso introdurre **guardie**, espressioni boolean per filtrare

```
[ (x * y) | x <- [1..10], y <- [1..10], (mod (x*y) 2) == 1 ]
```

Quicksort

non efficiente come la versione imperativa

```
quickSort [] = []  
quickSort (x:xs) =  
    quickSort[ y | y <- xs, y < x] ++  
    (x : quickSort[ y | y <- xs, x <= y])
```

```
quickSort ([3,7..40]++[2,5..30])  
quickSort "Hello World"
```

- stringhe liste di caratteri
- sui caratteri esiste una relazione d'ordine

```
quickSort :: Ord a => [a] -> [a]
```

- Haskell ha un meccanismo di [typeclass](#), parente delle [interface](#) di Java che permette un più sofisticato polimorfismo parametrico

Haskell, attraverso il modulo precaricato Prelude, mette disposizione:

- le classiche operazioni aritmetiche e logiche, con usuale notazione infissa, implementate in linguaggio macchina,

```
+ - * / || && < > <= >= == /= ...
```

- una serie funzioni analitiche, implementazione interna

```
sin cos tan exp ...
```

- funzioni di I/O

- funzioni Haskell predefinite su liste, numeri

```
head tail take sum mcd lcd max
```

per le funzioni in Prelude

- pratica “A tour of Prelude”

<http://www.cse.chalmers.se/edu/course/TDA555/tourofprelude.html>

- più completa: tramite Hackage,

<http://hackage.haskell.org/package/base-4.11.0.0/docs/Prelude.html>

- in generale una Hackage offre una panoramica dei package, librerie, disponibile per Haskell

reverse: invertire un stringa

- semplice ma poco efficiente
- accumulatore e ricorsione di coda
`reverseAux xs ys` coincide con `(reverse xs) ++ ys`
- usando la `foldl` o `foldr`

reverse: invertire un stringa

- semplice ma poco efficiente
- accumulatore e ricorsione di coda
reverseAux xs ys coincide con (reverse xs) ++ ys
- usando la foldl o foldr

```
reverse xs = reverseAux xs []  
  where reverseAux [] ys = ys  
         reverseAux (x:xs) ys = reverseAux xs (x:ys)
```

Fold left

Definire fold left, tipo

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

`foldl f z l`, usando come valore iniziale `z`, applica `f` a tutti gli elementi della lista da sinistra a destra, ossia:

```
foldl f z [a1, a2, ... , an] = f ( ... (f ( f z a1 ) a2) ... )
```

Fold left

Definire fold left, tipo

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

`foldl f z l`, usando come valore iniziale `z`, applica `f` a tutti gli elementi della lista da sinistra a destra, ossia:

```
foldl f z [a1, a2, ... , an] = f ( ... (f (f z a1) a2) ... )
```

```
foldl f z [] = z
```

```
foldl f z (x:xs) = foldl f (f z x) xs
```

Definire `reverse` da `foldl`

Fold left

Definire fold left, tipo

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

foldl f z l, usando come valore iniziale z,
applica f a tutti gli elementi della lista da sinistra a destra, ossia:

```
foldl f z [a1, a2, ... , an] = f ( ... (f (f z a1) a2) ... )
```

```
foldl f z [] = z
```

```
foldl f z (x:xs) = foldl f (f z x) xs
```

Definire reverse da foldl

```
reverse xs = foldl (\ys y -> (y:ys)) [] xs
```

alternativamente

```
reverse xs = foldl (\ys -> \y -> (y:ys)) [] xs
```

```
reverse xs = foldl (\ys -> (:ys)) [] xs
```

Definire fold right, tipo

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

come foldl ma in ordine opposto, si parte dall'ultimo elemento della lista ossia:

```
foldr f z [a1, a2, ... , an] = f a1 (f a2 ... (f an z) ... )
```

Definire fold right, tipo

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

come foldl ma in ordine opposto, si parte dall'ultimo elemento della lista ossia:

```
foldr f z [a1, a2, ... , an] = f a1 (f a2 ... (f an z) ... )
```

```
foldr f z [] = z
```

```
foldr f z (x:xs) = f x (foldr f z xs)
```

Definire reverse con foldr

```
reverse xs = foldr (\ y ys -> (ys ++ [y])) [] xs
```

Come vengono interpretate, eseguite le definizioni per pattern matching?

```
take 0 _ = []  
take _ [] = []  
take n (x:xs) = x : take (n-1) xs
```

zucchero sintattico per un costrutto case sull'enupla degli argomenti

```
take n xs = case (n, xs) of  
    (0, _) -> []  
    (_, []) -> []  
    (n, (y:ys)) -> y : take (n-1) ys
```

Case – definizione per casi

Haskell permette definizione per casi `case` che considerano più valori contemporaneamente

Possono essere visti come zucchero sintattico per una serie di case sulle singole componenti

```
take n xs = case n of
  0 -> []
  _ -> case xs of
    [] -> []
    (y:ys) -> y : take (n-1) ys
```

- le definizioni per pattern matching rendono il codice più leggibile
- permettono di scrivere in maniera sintetica, chiara, un insieme costrutti case annidati tra loro

Come viene eseguito un case? Es.

```
case x of  
  [] ->  
  y: ys ->
```

forza la valutazione dell'argomento x , passaggio per nome, alle variabili
associa closure (espressioni, environment)

come avviene la valutazione di x ?

Valutazione dell'argomento del case

dipende dal tipo di x

- x **scalare**, viene valutato completamente il suo valore comparato con quello dei casi
- x tipo **data**
 - viene valutato fino a determinare il suo costruttore
 - es. tipo `Tree Leaf n 0 Branch x1 x2`
 - valutazione **lazy**, si valuta lo stretto necessario, le componenti restano espressioni (closure)
 - i rami di case marcati con i costruttori

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
height = \ t -> case t of
  Leaf n -> 0
  Branch t1 t2 -> max (height t1) (height t2)
```

- possibili definizioni non esaustive (non tutti i casi considerati),
- casi non previsti generano errori a run-time.

Pattern matching, risultati per un pattern

- i pattern possono essere piuttosto complessi
- dato un pattern (es. $x : 0 : xs$) ed un'espressione e la valutazione del pattern su e può
 - fallire, l'espressione valutata non rispetta il pattern, (es. se e diventa $3:1:e2$ o se diventa $4: []$)
 - avere successo, es. e riduce $5:0:e1$
 - in questo caso
 - x viene legata a 5
 - xs alla closure $e1$
 - divergere (o generare errore), la computazione non produce sufficienti risultati (es. e riduce a $1:e3$, e la valutazione di $e3$ diverge)

- cosa posso inserire in un espressioni pattern, es `3:x:y:[]`
 - costanti
 - costruttori
 - variabili
 - non possono essere ripetute
 - considerate nuove, variabili locali del pattern, nessun collegamento con variabili preesistenti

```
x = 10
```

```
take 0 _ = []
```

```
take n (x:xs) = x : take (n-1) xs
```

- wildcard: caso particolare di variabile, rimarco il disinteresse per il valore istanziato

Come viene forzata la valutazione degli argomenti

- in base all'ordine di scritte delle regole, dalla prima all'ultima
- sulla singola regole, i pattern vengono testati da sinistra a destra
 - se un pattern fallisce: passo alla regola successiva
 - se un pattern diverge, la computazione diverge
 - se un (la sequenza di) pattern ha successo si valuta il corpo, con i legami creati dal pattern

Esempio: le seguenti definizioni:

```
take 0 _ = []
```

```
take _ [] = []
```

```
take n (x:xs) = x : take (n-1) xs
```

e

```
take _ [] = []
```

```
take 0 _ = []
```

```
take n (x:xs) = x : take (n-1) xs
```

non sono equivalenti

Quando possono dare risultati differenti?

Pattern con guardie

Ai pattern posso far seguire una lista di guardie

- espressioni booleane
- esaminate dalla prima all'ultima, se il pattern ha successo
- se una guardia valuta `True`, valuto il corpo corrispondente
- se una guardia valuta `False`, si passa alla successiva
- se tutte `False`, il pattern fallisce e si passa al successivo

```
sign x | x > 0 = 1  
      | x == 0 = 0  
      | x < 0 = -1
```

- posso usare le guardia anche con il costrutto `case`

```
sign x = case x of  
  y | y > 0 -> 1  
    | y == 0 -> 0  
    | y < 0 -> -1
```

Guardie: meccanismo generale di programmazione

If then else

Posso usare il costrutto

```
if b then e1 else e2
```

equivalente a

```
case b of
  True  -> e1
  False -> e2
```

Forma generale del pattern con guardia

Definizione per pattern matching

```
funName pattern1 | guardia11 = exp11
                  | ...
                  | guardia1n = exp1n
funName pattern2 | guardia21 = exp21
                  | ..
                  | guardia2m = exp2m
funName pattern3 ...
    ...
```

Forma generale del pattern con guardia

Posso usare le guardie anche col costrutto case

```
case exp of
  pattern1 | guardia11 -> exp11
           | ...
           | guardia1n -> exp1n
  pattern2 | guardia21 -> exp21
           | ..
           | guardia2m -> exp2m
  pattern3 ...
  ...
```

Esempio:

Calcolare la somma dei numeri pari di una lista, definizione per pattern matching.

Esempio:

Calcolare la somma dei numeri pari di una lista, definizione per pattern matching.

```
sommaPari [] = 0
sommaPari (x:xs) | (mod x 2) == 0 = x + sommaPari xs
                  | otherwise     = sommaPari xs
```

Calcolare la somma dei numeri dispari di una lista, definizione via case

Esempio:

Calcolare la somma dei numeri pari di una lista, definizione per pattern matching.

```
sommaPari [] = 0
sommaPari (x:xs) | (mod x 2) == 0 = x + sommaPari xs
                  | otherwise      = sommaPari xs
```

Calcolare la somma dei numeri dispari di una lista, definizione via case

```
sommaDispari = \ ys ->
  case ys of [] -> 0
             (x:xs) | (mod x 2) == 0 -> sommaDispari xs
                    | True           -> x + sommaDispari xs
```

Separatori e blocchi possono essere usati esplicitamente (come in Python)

```
z = let x = 5 + y
      y = 6
      in x + y
```

può essere scritto con blocco e separazione delle dichiarazioni espliciti

```
z = let { x = 5 + y; y = 6 } in x + y
```

Una tabulazione imprecisa a porta a errori di compilazione. Es:

```
z = let x = 5 + y
      y = 6
      in x + y
```

- in questi casi i messaggi di errore del compilatore possono essere poco chiari: non si evidenzia l'errore di tabulazione

Record con campi etichettati

Le componenti di un data type sono identificate dalla loro posizione

```
data Point = Pt Float Float
p = Pt 3 4
```

Posso accedere alle componenti per pattern-matching

```
xCoord (Pt x _) = x
```

Alternativamente, come nei record, posso accedere alle componenti attraverso etichette **field labels**

```
data Point = Pt {xCoord, yCoord :: Float}
```

```
data Point = Pt {xCoord, yCoord :: Float}
p = Pt{ xCoord = 3, yCoord =5}
a = xCoord p
q = p{ xCoord = 5}
a2 = xCoord p
b = xCoord q
```

- attraverso le etichette specifico le componenti di un nuovo Pt
- le etichette diventa funzioni per accedere alle componenti (distruttrici) `xCoord :: Pt -> Float`
- posso definire un nuovo punto q, a partire da uno esistente p, modificando solo alcune campi, gli altri vengono copiati.

Una stessa etichetta può essere utilizzata da più costruttori.

```
data Point23 = Pt2 {xCoord, yCoord :: Float}
              | Pt3 {xCoord, yCoord, zCoord :: Float}
```

Etichette con tipi union rendono possibili errori di tipo rilevabili solo a run time,

- la dichiarazione seguente accettata,

```
p = Pt2{xCoord = 3, yCoord = 5}
a = zCoord p
```

- genera errore solo se si valuta a.

Definire `sommaPari` e `map` usando più “idiomi”, metodi di scrittura

- pattern matching
- costrutto `case`
- ricorsione di coda
- via `foldl`
- via `foldr`

Definire tipo di dato albero binario, `BTree` con tutti i nodi etichettati,

Definire le seguenti funzioni sui `BTree`

- `sumBTree`
- `depthBTree`
- `takeDepthBTree`

Definire le seguenti funzioni su alberi binari di ricerca, `BST`

- `insert`
- `search`
- `toList`

Ordinare una lista usando i `BST`.

Meccanismi per incrementare il **polimorfismo parametrico**

Nella versione semplice il polimorfismo parametrico

- usa espressioni di tipo con variabili (parametri) per descrivere il comportamento di programmi.

Esempio:

```
map _ []      = []  
map f (x:xs) = f x : map f xs
```

- `map :: (a->b) -> [a] -> [b]`
- tipo derivato mediante un algoritmo di **inferenza di tipo**
 - associa a `map` in suo tipo parametrico più generale,
 - tutti gli altri possibili tipi per `map`,
es `(a->a) -> [a] -> [a]` oppure `(Char->b) -> [Char] -> [b]`
ottenibili per istanziazione del tipo più generale
- esempio di derivazione di tipi per `map ...`

Esistono funzioni polimorfe non descrivibili in questo modo. Esempio

```
quickSort [] = []  
quickSort (x:xs) = quickSort [ y | y <- xs, y < x]  
                  ++ x : quickSort [ y | y <- xs, x <= y]
```

ha tipo `Ord a => [a] -> [a]`

`quickSort` applicabile ad una lista generica, `[a]`,
a condizioni che i suoi elementi,

- possano essere applicate le funzioni `<` `<=`
- condizione espressa mediante la condizione `Ord a`:
il tipo `a` appartenete alla typeclass `Ord`

Affinché `quickSort` sia polimorfa bisogna:

- permettere che le funzioni `<` `<=` appartengano a più tipi
 - eventualmente con implementazione diverse, più metodi, codici, per confrontare oggetti
- ossia: permettere il **polimorfismo di overloading**

Mostreremo dichiarazioni per:

- specificare insiemi di tipi, **typeclass**, (esempio `Ord`) con un insieme di funzioni associate
- specificare che un **tipo** appartiene ad una **typeclass** (esempio `Char` appartiene a `Ord`)

Typeclass - dichiarazioni e istanziazioni

Le typeclass definiscono insiemi di tipi che posseggono le necessarie funzioni, chiamate **metodi**.

```
class Eq a where
  (==)    :: a -> a -> Bool
```

- Un tipo T per appartenere alla typeclass Eq deve avere una funzione == (usata in notazione infissa).

L'associazione di un tipo ad una typeclass,

- devo essere dichiarata esplicitamente
- bisogna fornire un implementazione per i metodi della typeclass

```
instance Eq Integer where
  x == y    = integerEq x y
```

- Alla funzione polimorfa

```
elem x [] = False
```

```
elem x (y:ys) = x == y || (elem x ys)
```

viene associato tipo

```
elem :: (Eq a) => a -> [a] -> Bool
```

- il simbolo => da intendersi come implicazione logica:
 - se la condizione ([constraint](#)) `Eq a` è soddisfatto (il tipo `a` appartiene alla typeclass `Eq`)
 - allora `elem` ha tipo `a -> [a] -> Bool`.

Implementazione di default dei metodi

La “vera” definizione di `Eq`, typeclass predefinita in `Prelude`,

```
class Eq a where
  (==), (/=)    :: a -> a -> Bool
  x /= y       = not (x == y)
  x == y       = not (x /= y)
```

- fornisco una implementazione di default per alcuni metodi, basata sugli altri metodi,
- nelle definizioni `instance`
 - posso definire tutti i metodi
 - definirne solo alcuni e sfruttare le implementazioni default per gli altri
 - nella specifica di una typeclass si definisce anche una lista di definizioni minimali

Implementazione di default dei metodi

```
instance Eq Integer where
  x == y      = integerEq x y
  x \= y      = False
```

oppure

```
instance Eq Integer where
  x == y      = integerEq x y
```

oppure

```
instance Eq Integer where
  x \= y      = False
```

- definizione semanticamente scorrette accettate da Haskell,
- a livello di documentazione, nei package, vengono specificate le uguaglianze attese ma
- Haskell controlla solo:
 - il tipo dei metodi
 - che i metodi forniti sia sufficienti a determinare tutti gli altri

Definizione di instance polimorfe

```
instance (Eq a) => Eq (Tree a) where
  Leaf l1          == Leaf l2          = l1 == l2
  Branch t11 tr1 == Branch t12 tr2 = t11 == t12 && tr1 == tr2
  -                == -                = False
```

- per definire l'equivalenza su alberi, serve l'equivalenza sugli elementi base, ossia
- sotto la condizione Eq a posso dichiarare Eq (Tree a)
- nella definizione delle metodo == su Tree a uso:
 - l1 == l2 definita da Eq a
 - t11 == t12 chiamata ricorsiva

Queste definizioni strutturali possono essere create automaticamente da Haskell usando `deriving` nella definizione di tipo

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
  deriving (Eq)
```

Relazione di sub-typeclass

- Posso dichiarare una typeclass sub-typeclass (estensione) di un'altra

```
class (Eq a) => Ord a where
  (<), (<=), (>), (>=)  :: a -> a -> Bool
  min, max              :: a -> a -> a
  x <= y                = x < y || x == y
  ...
```

- `Ord` e' sub-typeclass di `Eq`
- `Eq` e' super-typeclass di `Ord`

- dichiara che la typeclass `Ord` possiede tutti i metodi della typeclass `Eq`
quindi `==` e `\=`
- nel definire tipo instance di `Ord` devo fornire anche in metodi di `Eq`
- un tipo `T` instance di `Ord` e' automaticamente instance di `Eq`

- Posso definire un tipo sub-typeclass di più typeclass

```
class (Eq a, Show a) => EqAndShow a where
```

```
...
```

- In tipo instance di EqAndShow deve possedere i metodi delle due sottoclassi Eq e Show
più eventuali metodi aggiuntivi

- typeclass e metodi definiti a livello globale, visibili in tutto il programma
- uno stesso nome di metodo non può essere dichiarato su più typeclass
- metodi comuni possibile solo attraverso la definizione di sub-typeclass
- metodi, variabili, funzioni sono nello stesso **name space**
 - non posso usare lo stesso nome per un metodo ed una funzione

Typeclasses e Interfaces.

Pur con alcune differenze, possiamo paragonare Type Java.

Haskell	Java
Types	Classes
Typeclasses	Interfaces
Elementi	Oggetti

`instance Schema Type | class Type implements Schema {...}`
Polimorfismo parametrico | Tipo generici con con typeclass | relazione di sottotipo

Naturalmente Haskell non OO, quindi una lunga serie di differenze

- i metodi non sono collegati agli oggetti, non si usa la sintassi `object.method`
- nessun mascheramento, stato nascosto (nomi privati, pubblici)
- non c'è ereditarietà, ma vengono definite implementazioni di default

Una catalogazione delle espressioni Haskell

- Espressioni semplici,

```
5 + 2
\ x -> x + 1
```

- Espressioni di tipo,

Integer

Integer -> Integer

- costruttori di tipo,

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

```
data BTree a b = LLeaf a | BBranch b (BTree a b) (BTree a b)
```

- typeclass, proprietà di tipi

Eq

Ord

Struttura piuttosto complessa:

- necessario mettere ordine
- nozione di `kind`
- estensione della nozione di tipo, applicabile ad espressione che non sono elementi e non hanno tipo (tipi, costruttori di tipo, `typeclass`)
- ai vari identificatori associa un etichetta che indica a che categoria appartengono
- `kind` dedotto dal compilatore, possibile interrogare l'interprete per sapere il `kind` associato ad un'espressione

- alle espressioni semplici associ un tipo

```
λ> :type (+1)
```

```
(+1) :: Num a => a -> a
```

- a tipi, il kind *,

```
λ> :kind Integer -> Integer
```

```
Integer -> Integer :: *
```

- ai costruttori di tipo, kind più complessi, esprimono il fatto di essere funzioni da tipo a tipo

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
data BTree a b = LLeaf a | BBranch b (BTree a b) (BTree a b)
λ> :kind Tree
BTree :: * -> *
λ> :kind BTree
BTree :: * -> * -> *
λ> :kind (->)
(->) :: TYPE q -> TYPE r -> *
λ> :kind (->) Integer
(->) Integer :: * -> *
```

- alle typeclass, proprietà di tipi

```
λ> :kind Eq
```

```
Eq :: * -> Constraint
```

```
λ> :kind Functor
```

```
Functor :: (* -> *) -> Constraint
```

valutazione dei kind fatta dal compilatore,
un type checking di secondo livello

In Prelude

<https://hackage.haskell.org/package/base-4.6.0.1/docs/Prelude.html>

vengono definite

- un insieme di typeclass
 - ricca relazione di sub-typeclass
 - principalmente ordini e numeriche
- tipi, costruttori di tipo, definiti anche come istanze delle opportune typeclass,

Esempi di typeclasss - Functor

```
class Functor f where
```

```
  fmap :: (a -> b) -> f a -> f b
```

- Functor non definisce un vincolo sui tipi ma sui costruttori di tipo
- Esempio, il costruttore di lista

```
instance Functor [] where
```

```
  fmap = map
```

- ci si aspetta che le seguenti regole siano soddisfatte

```
fmap id == id
```

```
fmap (f . g) == fmap f . fmap g
```

Costruttori di tipo canonici, Maybe

- Maybe: serve a rappresentare eccezioni, errori

```
data Maybe a = Nothing | Just a
```

- `Nothing` eccezione
- `Just 5` computazione corretta

- Esempio di uso

```
myhead :: [a] -> Maybe a  
myhead []      = Nothing  
myhead (x:xs) = Just x
```

- Maybe è un funtore

```
instance Functor Maybe where  
  fmap f Nothing = Nothing  
  fmap f (Just x) = Just (f x)
```

- Either, unione tra due valori

```
data Either a b = Left a | Right b
```

- Either non è un instance of Functor (kind non corretto) Either Int sì.

- Haskell ha un ricco insieme di tipi numerici, ereditato da Scheme
 - interi (dimensione fissa `Int` e arbitraria `Integer`)
 - frazionari (coppie di interi)
 - floating point (precisione singoli e doppie)
 - complessi (coppie di floating point)
- Catalogato con un ricco insieme di typeclass.

Typeclass numeriche

- Num typeclass più generale,
non sub-typeclass di Ord perché i complessi non sono ordinati

```
class (Eq a) => Num a where
  (+), (-), (*)  :: a -> a -> a
  negate, abs   :: a -> a
  fromInteger   :: Integer -> a
```

- Real la typeclass più generale di numeri ordinati

```
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
```

Typeclass numeriche

- Real non possiede la divisione, le sue sub-typeclass si dividono per il tipo di divisione
- Integral numeri con la divisione intera (modulo e resto)

```
class (Real a, Enum a) => Integral a where
quot, rem  :: a -> a -> a
-- arrotonda verso 0
-- (x `quot` y)*y + (x `rem` y) == x
div, mod   :: a -> a -> a
-- arrotonda al numero piú piccolo
-- (x `div` y)*y + (x `mod` y) == x
toInteger  :: a -> Integer
```

- Fractional numeri con divisione esatta

```
class Num a => Fractional a where
(/)      :: a -> a -> a
recip    :: a -> a
```

- Floating fornisce le funzioni analitiche

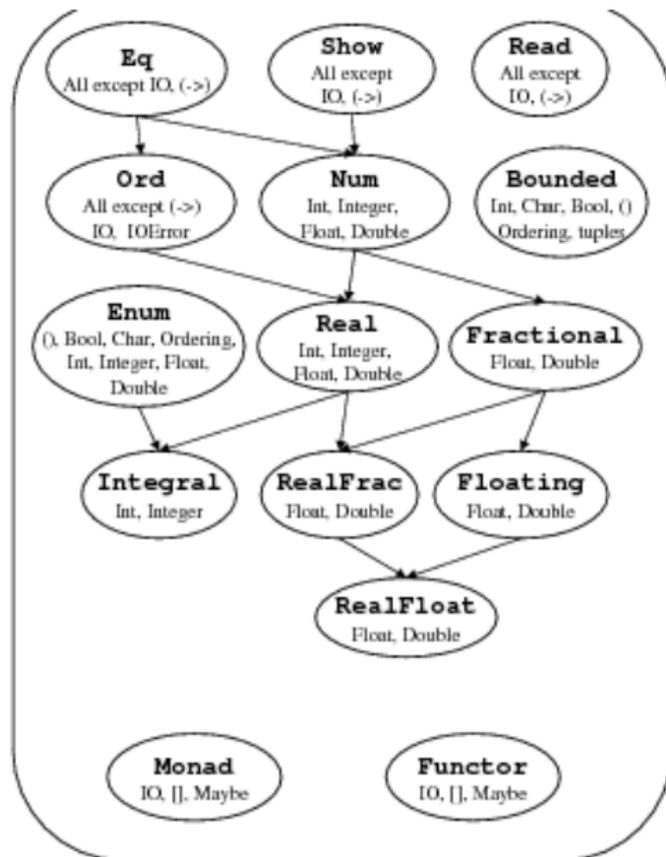
```
class Fractional a => Floating a
```

- RealFrac fornisce funzioni di arrotondamento (all'intero più vicino)

```
class (Real a, Fractional a) => RealFrac a
```

- con le relative istanze di tipo Integer, Double, Float, Int32, Natural
 - Num conta una 40ina di possibili istanze

Schema



Casting esplicito

- per passare da un tipo numerico all'altro devo esplicitare nel codice la conversione
- un insieme di funzioni ad hoc

```
fromIntegral :: (Num b, Integral a) => a -> b
```

```
fromInteger :: Num a => Integer -> a
```

```
realToFrac :: (Real a, Fractional b) => a -> b
```

```
fromRational :: Fractional a => Rational -> a
```

```
...
```

```
ceiling    :: (RealFrac a, Integral b) => a -> b
```

```
floor      :: (RealFrac a, Integral b) => a -> b
```

```
truncate   :: (RealFrac a, Integral b) => a -> b
```

```
round      :: (RealFrac a, Integral b) => a -> b
```

Nell'uso pratico IO haskell non troppo complesso

- definisco un variabile `main` di tipi IO ()
- codice di `main` simile ad una sequenza di comandi
 - leggono dati da input
 - chiamano altre funzioni per valutare il risultato
 - generano in uscita il risultato finale

l'apparente semplicità nasconde concetti più complessi

- una gestione pulita degli effetti collaterali

Haskell linguaggio funzionale puro.

- non esiste memoria, stato,
- definisco solo funzioni da un tipo ad un altro, senza effetti collaterali

Questo vincolo rende complesso gestire l'input-output

- anche in una versione semplice: stringa di input e stringa di output
- la coppia di stringhe costituiscono uno stato, S , che può
- viene modificato dalle operazioni di input - output
 - **input**: consuma caratteri nel file di input per generare un valore
 - **output**: da un valore genera caratteri inseriti nel file di output
- si creano **effetti collaterali**

Due alternative

- permetto che la valutazione di espressioni abbia effetti collaterali (side effect), senza segnalare questi effetti collaterali nel tipo dell'espressione
 - linguaggio funzionale non puro
 - soluzione di ML (Scheme)
 - il comportamento del programma dipende dall'ordine di valutazione
- introduco esplicitamente lo stato S nel tipo delle funzioni
 - una funzione $f :: A \rightarrow B$ che modifica lo stato, rappresentata da:
 - $fs :: (A, S) \rightarrow (B, S)$
 - curryficando:
 - $fs :: A \rightarrow (S \rightarrow (B, S))$
- seconda soluzione scelta da Haskell, funzionale puro:
 - usando funzioni con effetti collaterali, definisco nel modulo principale una funzione
 - `main :: S -> ((), S)` `main :: IO()`
 - l'esecuzione del programma, provoca la valutazione di `main` che modifica dello stato

- In Haskell esiste un costruttore di tipi predefinito IO

- che posso interpretare, intuitivamente, come

```
type IO a = (S -> (a,S))  --- pseudo definizione
```

- dove S rappresenta lo stato
- espressioni con tipo IO a rappresentano **action**
 - **comandi**, se hanno tipo tipo IO ()
 - **espressioni**, di tipo a, con **effetti collaterali**, se hanno tipo IO a
- la definizione di tipo IO nascosta,
 - non esistono costruttori espliciti
 - ma esistono funzione predefinite sui tipi IO
getChar, putChar ...

Funzioni di input output

- `getChar :: IO Char`
ossia `S -> (Char, S)`
dato un stato, mi restituisce un carattere e lo stato modificato,
(con un carattere in meno nel file di input)
- `putChar :: Char -> IO ()`
ossia `Char -> S -> ((), S)`
 - `()` è il tipo delle enuple con nessun elemento, contiene un unico valore, la enupla vuota `()` equivalente al `void` in C
 - `putChar`, dato un carattere ed uno stato restituisce lo stato modificato (con un carattere in più nel file di output)

- Come già scritto, il tipo IO nascosto, possiamo immaginare

```
type IO a = (S -> (a,S))
```

- Cosa sia esattamente lo stato S non è definito esplicitamente, intuitivamente contiene
 - i file di input e output
 - gli altri file manipolabili
 - lo stato che rappresenta eventuali condizioni di errore, eccezioni,

Espressioni di tipo `IO` hanno:

- solo un numero limitato di costrutti per manipolarle
la parti imperativa, non viene mescolato con quella funzionale
- le uniche funzioni f definibili di tipo `(IO a) -> Integer` sono funzioni costanti
funzioni che preso in input un comando (action) `c` lo ignorano, non lo eseguono

Posso comporre i comandi base con

- $(\gg) :: IO\ a \rightarrow IO\ b \rightarrow IO\ b$
 $(S \rightarrow (a, S)) \rightarrow (S \rightarrow (b, S)) \rightarrow (S, (b \rightarrow S))$
 - composizione di due comandi
 - trascurando in valore in a generato dal primo
 - ; nei linguaggi imperativi
- $(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$
 $(S \rightarrow (a, S)) \rightarrow ((a, S) \rightarrow (b, S)) \rightarrow (S, (b \rightarrow S))$
 - compone i due comandi (azioni)
 - tenendo conto del risultato generato dalla prima

Posso creare un comando, banale, con:

- $return :: a \rightarrow IO\ a$ $(a \rightarrow (S \rightarrow (a, S)))$
 - presa un'espressione v di tipo a
 - restituisce un'azione che lascia lo stato inalterato e ritorna v

Gli operatori (`>>`), (`>>=`), `return` sono motivati dalla nozione di **monade**

Nozione presa dalla matematica (teoria delle categorie)

- utile per descrivere diverse costruzione della matematica (insieme delle parti, grammatica libera su un alfabeto)
- in informatica utile per descrivere diversi costruttori di tipo (liste, `IO`, `Maybe`)
- in Haskell, esiste una `type class` per costruttori di tipo `Monad m` :
`* -> *`
 - `IO`, `[]`, `Maybe` sono `instance of Monad`, esempi di monadi
 - tra le funzioni caratterizzanti delle monadi troviamo
 - `(>>=) :: m a -> (a -> m b) -> m b`
 - `return :: a -> m a`

Monadi esempi

Le funzioni caratteristiche delle monadi possono essere usate come base per la programmazione:

```
[(x,y) | x <- [1,2,3] , y <- [3,2,1], x /= y]
```

```
[1,2,3] >>= (\ x -> [3,2,1] >>= (\y -> return (x/=y) >>=
  (\r -> case r of True -> return (x,y)
              _       -> fail "")))
```

```
do x <- [1,2,3]
   y <- [3,2,1]
   True <- return (x /= y)
   return (x,y)
```

La sintassi (\gg), ($\gg=$) poco intuitiva nel caso di composizione di comandi

- $c1 \gg c2 \gg c3$ associa a destra

$c1 \gg (c2 \gg c3)$

può essere scritto come `do { c1; c2; c3 }`

o come

```
do c1
   c2
   c3
```

Esempio

```
do getChar
   putChar 'h'
   putChar 'e'
```

- `c1 >>= (\ x -> (c2 >>= (\ y -> c3)))`

lo posso scrivere come

```
do x <- c1
   y <- c2
   c3
```

Esempio

```
do c1 <- getChar
   c2 <- getChar
   putChar c2
   putChar c1
```

Separazione tra parte funzionale e IO

- Su comandi, espressioni di tipo `IO a`, posso usare solo gli operatori `>>` e `>>=`
 - posso comporre comandi
 - un comando può passare dati ad un altro comando
 - un'espressione di tipo standard (`Integer`) non può forzare l'esecuzione di un comando, o estrarre informazioni da questo
- Un comando `IO a` è costituito dalla composizione mediante
 - `>>`, `>>=`, `return`, `do`
 - di comandi elementari predefiniti, ma anche comandi composti
 - posso inserire funzioni da valutare

Riassumendo:

- dentro un comando posso chiamare una funzione
- nella valutazione di una funzione pura non posso inserire un comando

Programmi compilati, entry point

In un programma compilato, devo definire l'**entry point**

- l'espressione che deve essere valutata come risultato dell'esecuzione del programma
- questa espressione deve avere:
 - nome `main`
 - tipo `I0`, tipicamente `I0 ()`
se avesse tipo diverso l'esecuzione del programma non avrebbe nessun effetto
- a volte necessario introdurre dichiarazione di tipo per `main`

```
main :: I0()
main = do c <- getChar
         putChar c
         putChar '\n'
```

- nel caso di programma diviso in moduli (vedi più avanti), `main` deve essere inserita nel modulo `main`

Posso costruire comandi più complessi di `getChar` e `putChar`
combinazioni più comandi

```
getLine2 :: IO String
getLine2 = do c <- getChar
              if c == '\n'
                then return ""    --- alias per []
                else do l <- getLine2
                       return (c:l)
```

- `getLine` presente in Prelude
- funzione `if then else` in notazione infissa

Esercizio: controllare la correttezza di tipo

```
putString :: String -> IO ()  
putString [] = return ()  
putString (x:xs) = do putChar x  
                      putStrLn xs
```

- in Prelude `putStr, putStrLn :: String -> IO ()`

Posso trattare le action (comandi), come un qualsiasi altro dato es, costruisco un vettore di comandi

```
todoList = [putChar 'H',  
            do c <- getChar  
              putChar c,  
              putStrLn "ello" ]  
todoList :: [IO ()]
```

- non posso eseguire un vettore di comandi,
- solo il singolo comando definito in main è valutato
tutto il resto sono espressioni, che possono essere manipolate,
- devo combinare i comandi in uno, e valutarlo nel main

```
sequence2 [] = return ()  
sequence2 (c:cs) = do c  
                      sequence2 cs
```

Tipo e definizioni alternative

```
sequence2 :: Monad m => [m a] -> m ()
sequence2 :: [IO a] -> IO ()

-- foldr :: (a -> b -> b) -> b -> [a] -> b
sequence3 :: [IO a] -> IO ()
sequence3 = foldr (>>) (return ())

-- foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
sequence4 :: [IO ()] -> IO ()
sequence4 = foldl (>>) (return ())

c2 = sequence2 toDoList
c3 = sequence3 toDoList
c4 = sequence4 toDoList
```

una seconda definizione per putStrLn

```
putStrLn2 = sequence . map putStrLn
```

- Gestione semplice dei file attraverso le funzioni

```
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

dove

```
type FilePath = String -- Defined in 'GHC.IO'
```

Posso costruire comandi per copiare un file in un altro

```
copy fileS fileD = do s <- readFile fileS
                      writeFile fileD s
```

```
c5 = copy "sommaPari.hs" "prova"
```

Oppure comandi per copiare il contenuto il contenuto dopo averlo modificato

```
modify fSource fDestin trasf = do s <- readFile fSource  
                                writeFile fDestin (trasf s)
```

```
c6 = modify "sommaPari.hs" "prova" tail
```

sorgente e destinazione devono essere diversi.

Typeclass Show Read

Definiscono funzioni standard per stampare e leggere valori da IO

```
class Show a where
  show :: a -> String
  shows :: a -> String -> String
  showsPrec :: Int -> a -> String -> String
  showList :: [a] -> ShowS
  {-# MINIMAL showsPrec | show #-}
```

- `show` funzione usata dall'interprete per visualizzare espressioni
 - insieme a `putString` mi permette di stampare dei dati da programma

- `shows :: a -> String -> String`
funzione con accumulatore per migliorare l'efficienza, evito di dover concatenare liste

```
putStr ("i num. " ++ show x ++ ' ': show y ++ " sono uguali ")
```

diventa

```
putStr ("i num. " ++ shows x (' ': shows y " sono uguali "))
```

Esempio

```
data BTree a = Null | BTree a (BTree a) (BTree a)
```

```
showsBTree Null = ('_':)
```

```
showsBTree (BTree n tl tr) =  
  ('(':) . showsBTree tl . sh
```

```
showsBTree :: Show a => BTree a -> String -> String
```

```
instance Show a => Show (BTree a) where
```

```
  showsPrec _ = showsBTree
```

In alternativa:

```
data BTree a = Null | BTree a (BTree a) (BTree a)
```

```
  deriving (Eq, Show)
```

Una type typeclass per leggere dati
trasformare una stringa di caratteri in un dato

Implementa un parser, usando back-tracking
semplice ma inefficiente

```
class Read a where
  readsPrec :: Int -> ReadS a
  readList  :: ReadS [a]
  ...
  ...
  {-# MINIMAL readsPrec | readPrec #-}

type ReadS a = String -> [(a, String)]
```

Esempio

```
readsBTree ('_':s) = [(Null,s)]
readsBTree ('(':s) =
  [((BTree n t1 tr), s3) | (t1, s1) <- readsBTree s,
                          (n, s2) <- reads s1,
                          (tr, ')':s3) <- readsBTree s2 ]

readsBTree :: Read a => String -> [(BTree a, String)]

instance Read a => Read (BTree a) where
  readsPrec _ = readsBTree
```

Sistema per gestire la visibilità dei nomi.

- Quadro generale
 - codice distribuito su più file
 - ogni file contiene una serie di definizioni
 - tipi, typeclass, istanza, funzioni (posso apparire in ogni ordine)
 - solo alcuni resi visibile all'esterno
 - altre definizioni ausiliare
 - il programma principale sceglie
 - da che moduli, file importare
 - seleziona cosa importare

Esempio

```
module BinarySearchTree (Bst (Null, Bst), insert, empty)
  where
data Bst = ...
insert = ...
empty  = ...
```

- la lista `(Bst (Null, Bst), insert, empty)` definisce in nomi esportati
- definizione che non appaiono nella lista non visibili all'esterno
- i costruttori di in tipo di dato messi vicino al tipo per leggibilità
- se ometto la lista dei nomi, tutto viene esportato
 - `Bst(Null, Bst)`
 - `Bst(..)` possibile sintassi
 - `Bst(Null)` costruttore `Bst` non visibile all'esterno.

- Posso utilizzare i moduli per implementare i tipi di dato astratti
 - nascondo l'effettiva implementazione del tipo di dato
 - fornisco solo un insieme di funzioni primitive per agire su questo
 - posso cambiare in modo trasparente l'implementazione

```
module BinarySearchTree (Bst, insert, null, empty, search)
```

```
module Main (main)
  where
import BinarySearchTree (Bst (Null), insert)
```

- le dichiarazioni di `import` vanno messe in testa
- posso selezionare i nomi da importare
- `import BinarySearchTree` senza lista, importata tutti i nomi esportati
- `import BinarySearchTree hiding (insert)` nasconde alcune `entity` che posso ridefinire

Qualified names

Attraverso le importazione da moduli

- possibile che lo stesso nome abbia più definizioni
- (ridefinizione dei nomi non possibile all'interno dello stesso modulo)

Tutte le definizioni restano valide

- devo eliminare l'ambiguità usando **qualified names**, nomi con prefissi
 - `BinarySearchTree.insert`
 - `Main.insert`
- per indicare `insert` nel caso
 - abbia importato il modulo `BinarySearchTree`
 - abbia definito nel codice una funzione `insert`.

- In un linguaggio funzionale un array `[] char A` diventa funzione `A :: Int -> Char`, o una lista `A :: [Char]`
- Implementazione inefficiente, per migliorare le prestazioni Haskell fornisce un tipo `Array`, definito nel modulo `Array`, da caricare

```
import Array
```

nel modulo troviamo le seguenti definizioni

Un type class per gli indici

```
class (Ord a) => Ix a where
  range    :: (a,a) -> [a]
  index    :: (a,a) -> a -> Int
  inRange  :: (a,a) -> a -> Bool
```

- posso costruire array con indici di a tipo qualsiasi, basta che a appartengano alla typeclass Ix
- sono dichiarati istanza di Ix gli interi, caratteri, enuple di istanze di Ix (Int, Int) usabili immediatamente come indici

Funzioni sugli array

Elenchiamo le più utili

Creazione:

```
array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

fornisco estremi degli indici, lista degli elementi, anche fuori ordine,
(indice, valore),

posso non inizializzare tutti gli elementi

```
squares = array (1,100) [(i, i*i) | i <- [1..100]]
```

possibili definizioni ricorsive

Accesso ai singoli elementi

```
(!) :: Array a b -> a -> b
```

```
squares!7 => 49
```

Funzioni sugli array

Accesso agli estremi del campo indici

```
bounds :: Array a b -> (a,a)
```

```
bounds squares => (1,100)
```

Modifica, multipla di alcuni elementi nell'array:

```
(//) :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

```
swapRows :: (Ix a, Ix b, Enum b) =>  
            a -> a -> Array (a,b) c -> Array (a,b) c
```

```
swapRows i i' a =  
  a // ([((i, j), a!(i',j)) | j <- [jLo..jHi]] ++  
        [(i',j), a!(i, j)] | j <- [jLo..jHi]))  
  where ((iLo,jLo),(iHi,jHi)) = bounds a
```

Formalmente si crea un nuovo array,
ma evitando di duplicare tutti gli elementi.

- Particolare idioma, stile di programmazione, facente uso di una typeclass di costrutti di tipo
- Nome preso dall'algebra, teoria delle categorie.
- Si replicano, in codice Haskell, alcuni modi di definire funzioni tipiche dell'algebra (superiore)
- Permettono definizioni molto compatte, ma piuttosto astratte e non di immediata comprensione.

Definizione formale

```
class Functor m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
  {-# MINIMAL (>>=) #-}
```

Un monade è:

- Un costruttore di tipi,
- Sub-Typeclass di Functor
- dalla definizione si deduce IO che sia una monade
- altri esempi: [], Maybe, Either a

- Inserisco il mio tipo di dato a dentro una struttura più ricca $m\ a$
- la funzione predefinita `return` $:: a \rightarrow m\ a$, opera l'inserzione
- definisco funzione sulle strutture ricche $m\ a \rightarrow m\ b$
 - a partire da funzioni sulle strutture semplici $a \rightarrow b$ o $a \rightarrow m\ b$
 - la funzione `(>>=)` $:: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ permette questo passaggio
 - $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ è isomorfo a $(a \rightarrow m\ b) \rightarrow (m\ a \rightarrow m\ b)$

Restano valide quelle definite per l'IO

- $c1 \gg c2 \gg c3$ associa a destra $c1 \gg (c2 \gg c3)$

lo posso scrivere come `do { c1; c2; c3 }`

```
do c1
   c2
   c3
```

- $c1 \gg= (\lambda x \rightarrow (c2 \gg= (\lambda y \rightarrow c3)))$

lo posso scrivere come

```
do x <- c1
   y <- c2
   c3
```

- []
- Maybe
- Either a

- Le definizioni `instance` ben fatte dovrebbero:
- soddisfare alcune eguaglianze
- si rispecchia la corrispondenza matematica

Per i funtori

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

Per le monadi

```
return a >>= k           = ka
m >>= return             = m
xs >>= return . f        = fmap f xs
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Esempio: Liste

```
[(x,y) | x <- [1,2,3] , y <- [3,2,1], x /= y]
```

```
[1,2,3] >>= (\ x -> [3,2,1] >>= (\y -> return (x/=y) >>=
  (\r -> case r of True -> return (x,y)
                _      -> fail "")))
```

```
do x <- [1,2,3]
   y <- [3,2,1]
   True <- return (x /= y)
   return (x,y)
```

definizione:

```
do p <- e1; e2 =
  e1 >>= (\v -> case v of p -> e2; _ -> fail "s")
```

Esempio: MyIO

- simulo, internamente ad Haskell, la Monade IO
- i comandi IO, sono funzioni
 - che modificano lo Stato
 - eventualmente restituiscono un valore
 - hanno tipo generico `Stato -> (a, Stato)`
- Stato coppia di stream: input, output
- definisco la monade

```
newtype MyIO a = MyIO (State -> (a, State))
```

- `>>=` deve gestire la composizione dei comandi
- trascrivo per `MyIO` le funzione base su IO

Esempio: linguaggi imperativo

- estendo l'esempio precedente
- simulo in Haskell l'esecuzione di comandi di ling. imperativo
- espressioni, comandi rappresentati come funzioni $\text{Stato} \rightarrow (a, \text{Stato})$
 - comandi pure non restituiscono valori in a
 - espressioni pure non modificano lo stato
 - ma posso mescolare le cose nessuna separazione netta
- Stato deve contenere anche la memoria
- Monade permette di gestire la composizione di comandi
- Liftare funzioni da valori a comandi
- problema complesso, legato semantica formale dei linguaggi,
- soluzione parziale