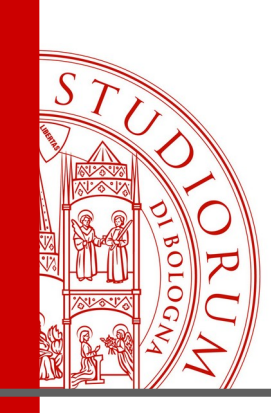
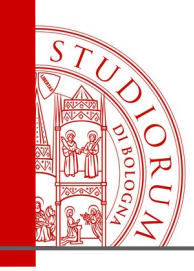
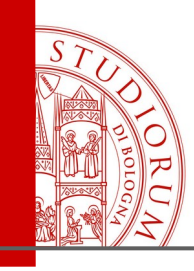


# 11 Monadi



# Haskell is pure!

- si può vivere senza side effects?
  - I/O (da file, da stream, a video, timers, …)
  - pseudo-random number generator
  - non determinismo
  - non local control (eccezioni, call-cc, eccezioni rientranti, handler algebrici, …)
  - concorrenza
  - accesso a stato condiviso (database)
  - ...



# Referential transparency

Purity => Referential transparency

Referential transparency:

**f i può essere sostituita con o dove o è il risultato della chiamata f i**

Proprietà che semplifica enormemente

- dimostrazione di correttezza di codice
- refactoring
- optimizations

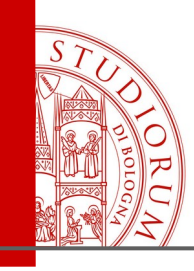
# Referential transparency

```
f () =
  let x = g 2
      y = h 2
  y + x + x
```

=

```
f () =
  h 2 + g 2
    + g 2
```

L’uguaglianza non vale in un linguaggio con side effects: l’ordine di esecuzione non può essere cambiato e nemmeno il numero di volte che un side effect viene eseguito

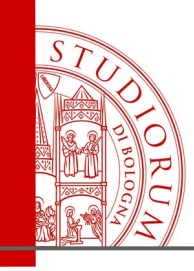


# Monads to the rescue!

Monadi:

- un meccanismo per gestire side-effect in un linguaggio funzionale SENZA perdere referential transparency
- MA NON SOLO!

Monade = un meccanismo per nascondere all'utente il threading di input/output aggiuntivi delle funzioni

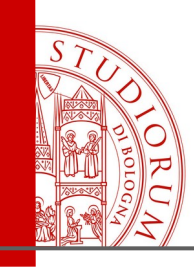


# Un esempio senza monadi

```
data Exp = Plus  Exp Exp
         | Minus Exp Exp
         | Times Exp Exp
         | Div   Exp Exp
         | Const Int

eval :: Exp -> Int
eval (Plus  e1 e2) = (eval e1) + (eval e2)
eval (Minus e1 e2) = (eval e1) - (eval e2)
eval (Times e1 e2) = (eval e1) * (eval e2)
eval (Div   e1 e2) = (eval e1) `div` (eval e2)
eval (Const i)     = i

answer = eval (Div (Plus (Const 4) (Const 2)) (Const 3))
```

# Gestire la divisione per zero

```
eval :: Exp -> Maybe Int
...
eval (Div  e1 e2) =
 case eval e1 of
    Nothing -> Nothing
    Just x1 ->
        case eval e2 of
          Nothing -> Nothing
          Just x2 ->
             if x2 == 0 then Nothing else Just (x1 `div` x2)
eval (Const i)     = Just i
```



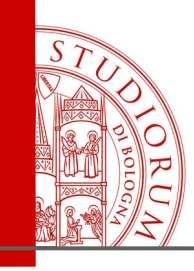
# Gestire la divisione per zero

```
-- >>= is spelled "bind"
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing >>= f   = Nothing
Just a >>= f    = f a

return :: a -> Maybe a
return x = Just x

eval :: Exp -> Maybe Int
eval (Div  e1 e2) =
   eval e1 >>= \x1 ->
   eval e2 >>= \x2 ->
   if x2 == 0 then Nothing else return (x1 `div` x2)
eval (Const i) = return i
```

# Do notation

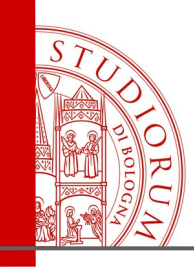
```
eval :: Exp -> Maybe Int
...
eval (Div  e1 e2) =
 do
   x1 <- eval e1
   x2 <- eval e2
   if x2 == 0 then Nothing else return (x1 `div` x2)
eval (Const i) = return i
```

The do notation (together with x ← t) introduces syntactic sugar for the >>= operator defined in the standard library of Haskell



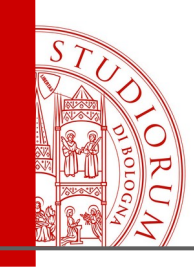
# Osservazioni

- grazie a return/>>= (e alla do notation) il codice esibisce esclusivamente la business logic
- la gestione funzionale della propagazione degli errori è completamente nascosta
- il Nothing del ramo else corrisponde a lanciare un’eccezione



# Osservazioni

- nel codice originale un elemento di tipo Int rappresenta il risultato di una computazione
- nel nuovo codice un elemento di tipo Maybe Int descrive una computazione che produrrà un intero
- l'operatore >>= permette di fare plumbing di tali descrizioni senza esporre come avviene la computazione
- Nothing descrive un nuovo modo di procedere nella computazione

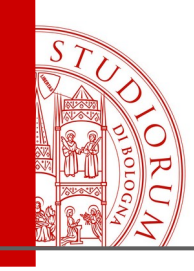


# Domanda

Quanto visto è replicabile per gli altri tipi di side effect?

# Sqrt e il non determinismo

```
data Exp = Plus  Exp Exp
         | Minus Exp Exp
         | Times Exp Exp
         | Div   Exp Exp
         | Sqrt  Exp
         | Const Int

isqrt :: Int -> Int
isqrt = floor . sqrt . fromIntegral

eval :: Exp -> [Int]
eval (Plus e1 e2) = [ x1 + x2 | x1 <- eval e1, x2 <- eval e2 ]
eval (Minus e1 e2) = [ x1 - x2 | x1 <- eval e1, x2 <- eval e2 ]
eval (Times e1 e2) = [ x1 * x2 | x1 <- eval e1, x2 <- eval e2 ]
eval (Div e1 e2) = [ x1 `div` x2 | x1 <- eval e1, x2 <- eval e2 ]
eval (Sqrt e1) = let l1 = isqrt (eval e1) in l1 ++ [ -x1 | x1 <- l1 ]
eval (Const i) = [i]

answer = eval (Div (Plus (Sqrt (Const 4)) (Sqrt (Const 4))) (Const 2))
```



# Sqrt e il non determinismo

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

return :: a -> [a]
return x = [x]

eval :: Exp -> [Int]
…
eval (Times e1 e2) = eval e1 >>= \x1 -> eval e2 >>= \x2 -> return (x1 * x2)
eval (Sqrt e1) = eval e1 >>= \x1 -> let x2 = isqrt x1 in [x2, -x2]
eval (Const i) = return i
```

# Do notation

```
eval :: Exp -> [Int]
…
eval (Times e1 e2) =
   do
      x1 <- eval e1
      x2 <- eval e2
      return (x1 * x2)
eval (Sqrt e1) =
   do
      x1 <- eval e1
      [x1, -x1]
eval (Const i) = return i
```

# Soluzioni a confronto

```
eval :: Exp -> Maybe Int
...
Eval (Times e1 e2) =
  do
       x1 <- eval e1
       x2 <- eval e2
       return (x1 * x2)
eval (Div  e1 e2) =
 do
   x1 <- eval e1
   x2 <- eval e2
   if x2 == 0 then
       Nothing
   else
       return (x1 `div` x2)
eval (Const i) = return i
```

```
eval :: Exp -> [Int]
...
eval (Times e1 e2) =
   do
       x1 <- eval e1
       x2 <- eval e2
       return (x1 * x2)
eval (Sqrt e1) =
   do
       x1 <- eval e1
       let x2 = isqrt x1 in
       [x2, -x2]


eval (Const i) = return i
```

# Osservazioni

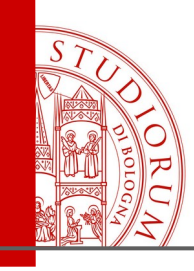
- nel codice originale un elemento di tipo Int rappresenta il risultato di una computazione
- nel nuovo codice un elemento di tipo [Int] descrive una computazione che produrrà un intero in maniera non deterministica
- l'operatore >>= permette di fare plumbing di tali descrizioni senza esporre come avviene la computazione
- [x1, -x1] descrive un nuovo modo di procedere nella computazione

# Le Monadi

```
class Applicative m => Monad m  where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  x >> y = x >>= \_ → y
  return :: a -> m a
  fail :: String -> m a
  fail s = error s
```

```
do                          e1 >>= \x ->
   p ← e1          =           case x of
   e2                             p → e2
                                  _ → fail “Pattern matching failure”
```

```
do
   e1              =        e1 >> e2
   e2
```



# Monadic Laws

1) return x >>= f = f x

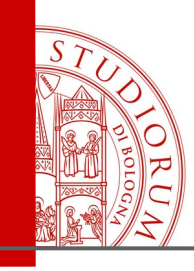
2) m >>= return = m

3) (m >>= f) >>= g = m >> \x → f x >>= g

4) return x >> m = m

5) non c'è una corrispondente della 2

6) (m >> n) >> l = m >> (n >> l)



# Monadic laws

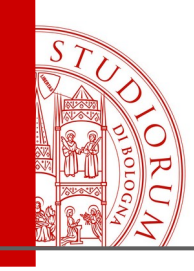
1) **do** { x' ← return x ; f x' } = f x

2) **do** { x ← m ; return x } = m

3) **do** { y ← **do** { x ← m ; f x } ; g y } = **do** { x ← m; **do** { y ← f x; g y }}

# Monads? They are everywhere!

- Writer:  scrive su stato write-only
  (es. logging)
- Reader: legge da stato read-only
  (es. Configurazione)
- State: legge e scrive da stato read/write
  - Caso particolare: pseudo generatore di numeri casuali

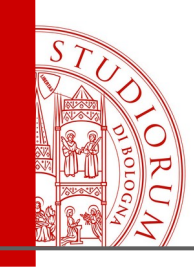


# Writer monad

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

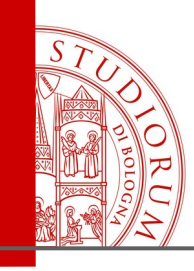
…. – (Writer w) must be shows to be an Applicative

```
instance (Monoid w) => Monad (Writer w) where
    return x = Writer (x, mempty)
    (Writer (x,v)) >>= f = let (Writer (y, v')) = f x in Writer (y, v `mappend` v')

tell :: Monoid w => w → Writer w ()
tell m = Writer ((), m)
```



# Writer monad

```
double :: Int -> Writer [Char] Int
double n =
  do
     tell "double called "
     return (n * n)

f :: Int -> Writer [Char] Int
f n =
  do
     tell "f called "
     x ← double n
     double x

*Main> f 2
Writer {runWriter = (16,"f called double called double called ")}
```

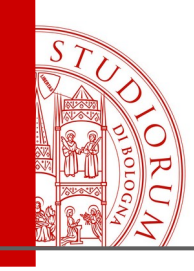


# Reader monad

```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w

read :: r -> r
read = id
```

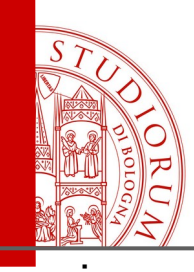
# Reader monad

```
age :: String -> [(String,Int)] -> Int
age x =
 do
   m <- read
   let a = case lookup x m of Just x -> x ; Nothing -> -1
   return a

family :: [(String,Int)] -> (Int,Int,Int)
family =
 do
   x <- age "Claudio"
   y <- age "Barbara"
   z <- age "Pietro"
   return (x,y,z)

answer :: (Int,Int,Int)
answer =
   let persons = [("Claudio",41),("Fabrizio",41),("Pietro",10),("Barbara",40)] in
   family persons
```



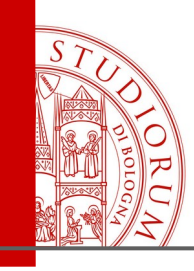
# State monad

```
newtype State s a = State { runState :: s -> (a,s) }

…

instance Monad (State s) where
    return x = State $ \s -> (x,s)
    (State h) >>= f = State $ \s -> let (a, newState) = h s
                                        (State g) = f a
                                    in  g newState

get :: State s s
get = State (\s -> (s,s))

put :: s -> State s ()
put s = State (\_ -> ((),s))
```



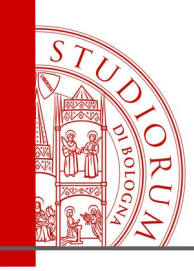
# State monad

```
incr :: State Int ()
incr =
 do
    n <- get
    put (n + 1)

f :: State Int (Int,Int)
f =
 do
   m <- get
   incr
   incr
   n <- get
   return (m,n)

answer :: (Int,Int)
answer = let { (State g) = f ; (r,_) = g 3 } in r
```



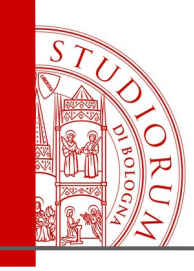
# Altre monadi

- distribuzioni probabilistiche (es. operatore di lancio di una moneta; descrive tutti i possibili output con le relative probabilità)
- eccezioni (tipo la Maybe monad, ma con valori trasportati)
- concorrenza e software transactional memory (= i thread comunicano con memoria condivisa + nozione di transazione atomica implementata in maniera ottimistica)
- futures
- parser che leggono stream e scrivono AST o falliscono, generando nomi freschi
- ……



# IO monad

- la IO monad gestisce l'interazione con il sistema (file, socket, timers, …)
- main deve avere tipo IO ()
- RICORDATE:
  - un valore di tipo "IO a" NON è il risultato di un'esecuzione che ha prodotto side-effec
  - è la DESCRIZIONE di tale esecuzione
- il run-time di Haskell "interpreta" il main di tipo IO () eseguendo la computazione, compresi i side-effect



# IO e referential transparency

```
f :: IO ()
f =  putStrLn "Hello" >> putStrLn "Hello" >> putStrLn "World"

g :: IO ()
G = let x = putStrLn "Hello" in x >> x >> putStrLn "World"

main :: IO ()
main = f >> g
```

# Un intero ecosistema categorico

- Functor m
  fmap :: (a → b) → m a → m b
- Applicative m
  pure :: a → m a
  <*> :: m (a → b) → m a → m b
  ….
- ……
- Monad transformer: combinano una monad m1 con una monade m2 per ottenere una che permetta le operazioni di entrambe
- ….

# Combinazione di monadi

```
eval :: Exp -> [Int + [Char]]
...
Eval (Times e1 e2) =
  do
      x1 <- eval e1
      x2 <- eval e2
      return (x1 * x2)

eval (Div  e1 e2) =
 do
   x1 <- eval e1
   x2 <- eval e2
   if x2 == 0 then
      error “division by zero”
   else
      return (x1 `div` x2)

eval (Const i) = return i
```

```
eval (Sqrt e1) =
   do
      x1 <- eval e1
      let x2 = isqrt x1
        [x2, -x2]
```

# Combinazione di monadi

[Int + [Char]]

E' una monade con le seguenti caratteristiche:
- non determinismo […]
- possibilità di errore … + [Char]

```
return x = [Left x]
bind (x::xs) f =
  (case x of
     Left a → f a
     Right msg → [Right msg]) ++ bind xs f
```

Possiamo evitare l'implementazione qua sopra e ottenere [Int + [Char]] per composizione?

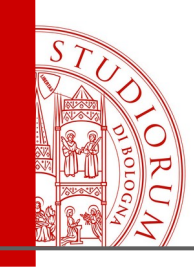
# Combinazione di monadi

Osservazione: la composizione, non è commutativa:

[Int + [Char]]  != [Int] + [Char]

Pertanto vorremmo ottenere

[.. + [Char]] := (.. + [Char]) o [..]



# Monad transformer

Un monad trasformer è

una monade di ordine superiore

ovvero

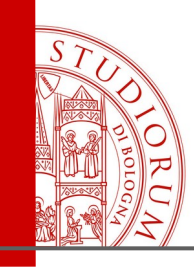
1) una monade parametrizzata su un'altra monade
2) che espone le operazioni della monade parametro

# Monad transformer: Maybe vs MaybeT

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >> = f = f x

type MaybeT m a = m (Maybe a)
```

Nota:

- m è un costruttore di tipo che è una monade
- a è un tipo



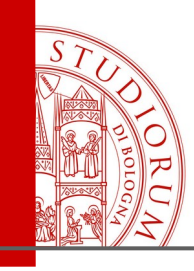
# MaybeT m è un Monad Transformer

```
type MaybeT m a = m (Maybe a)

instance Monad m => Monad (MaybeT m) where
   return :: a → m (Maybe a)
   return x = return (Just x)

   (>>=) :: m (Maybe a) → (a → m (Maybe b)) → m (Maybe b)
   x >>= f = x >>= \y →
      case y of
         Nothing → return Nothing
         Just z → f z
```

# Manca qualcosa!

Combiniamo nondeterminismo ([ ]) con errori (Maybe):

```
f :: Int → MaybeT [] Int
f 0 = return 3                          – ok return,
                                        – f 0 =  [ Just 3 ]
f 1 = [ 1, 2 ]                          – KO nondeterminism
                                        –  [1, 2] :: [ Int]
                                        – e non [ Maybe Int ]
f 2 = f 4 >>= \x → return (x + 1)       – ok bind,
                                        – se f4 = [ Just 1, Nothing] allora
                                        – f 2 = [ Just 2, Nothing ]
f 3 = Nothing                           – KO  errore
                                        – Nothing :: Maybe Int
                                        – e non  [ Maybe Int ]
```



# Recuperiamo le capabilities!

Errore:

```
error :: Monad m => m (Maybe a)
error = return Nothing
```

Ma come recuperiamo la capability della monade generica m?

```
class MonadTrans t where
    lift :: Monad m => m a → t m a
```

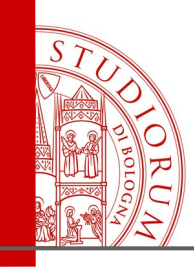
(Nota: t non è un tipo e nemmeno un costruttore di tipo; prende un costruttore di tipo e un tipo e restituisce un tipo!)

```
instance MonadTrans MaybeT where
    lift :: Monad m => m a → m (Maybe a)
    lift x = x >>= \y → return (Just y)
```

# Riproviamo!

Combiniamo nondeterminismo ([ ]) con errori (Maybe):

```
f :: Int → MaybeT [] Int
f 0 = return 3                          – ok return,
                                        – f 0 = [ Just 3 ]
f 1 = lift [ 1, 2 ]                     – ok nondeterminism
                                        – f 1 = [ Just 1, Just 2]
f 2 = f 4 >>= \x → return (x + 1)       – ok bind,
                                        – se f4 = [ Just 1, Nothing]
                                        – allora
                                        – f 2 = [ Just 2, Nothing ]
f 3 = error                             – ok errore
                                        – f 3 = [ Nothing ]
```



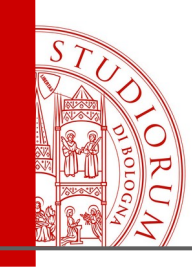
# Monad transformers: tutta la verità

t è un monad trasformer quando:

1) è una monade di ordine superiore
   **instance** Monad m => Monad (t m) **where** ...

2) lifta le operazioni della monade argomento
   **instance** MonadTrans t **where**
      lift :: Monad m => m a → t m a
      lift = ...

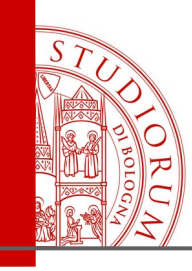
   soddisfando le equazioni:
      a) lift (return x) = return x
      b) lift (m >>= f) = lift m >>= \x → lift (f x)



# Evitare duplicazione Maybe/MaybeT

La monade assenza di side effect!

```
type Id x = x

instance Monad Id where
    return :: a → a
    return x = x

    (>>=) : a → (a → b) → b
    x >>= f = f x
```



# Evitare duplicazione Maybe/MaybeT

**type** Maybe x = MaybeT Id x

Quindi Maybe è un caso particolare di MaybeT ed è sufficiente dimostrare che MaybeT è un monad trasformer per avere che Maybe è una monade!

# Recapitoliamo sul nostro esempio

Vogliamo sia:
- nondeterminismo (per valutare radice quadrata)
- errori con messaggi di errore (per divisione per zero)

**type** Values a = EitherT [Char] [] a

Ovvero  Values a =  [ a + [Char] ]

eval :: Exp -> Values Int

# Combinazione di monadi

```
type Values a = EitherT [Char] [] a

eval :: Exp -> Values Int
...
Eval (Times e1 e2) =
  do
      x1 <- eval e1
      x2 <- eval e2
      return (x1 * x2)

eval (Div  e1 e2) =
 do
   x1 <- eval e1
   x2 <- eval e2
   if x2 == 0 then
      error “division by zero”
   else
      return (x1 `div` x2)
```

```
eval (Const i) = return i

eval (Sqrt e1) =
   do
      x1 <- eval e1
      let x2 = isqrt x1
        lift [x2, -x2]
```

# Successo? Quasi!

Nondeterminism**:**

lift [x2, -x2]

invece di

[ x2, -x2 ]

Se aggiungiamo il nuovo side-effect di logging:

**type** Values = EitherT [Char] (ListT (Writer [Char]))

possono aggiungersi altre lift, esempio:

lift (lift (tell “ciao”))  – 2 lift perché la Writer è trasformata due volte

# Soluzione: astriamo la monade

**class** Nondeterministic m **where**

returns :: [a] → m a

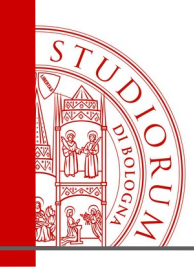
Returns è la capability del nondeterminismo:

es. returns [1, 1]

**instance** Nondeterministic [] **where**

returns :: [a] → [a]

returns l = l



# Soluzione: astraiamo la monade

```
instance
  (Nondeterministic m, MonadTrans t) =>
     Nondeterministic (t m)
where
   returns :: [a] → t m a
   returns l = lift (returns l)
```

Una trasformata qualunque di una monade nondeterministica lo è a sua volta!

Idem per ogni altra capability

# Codice finale: senza logging

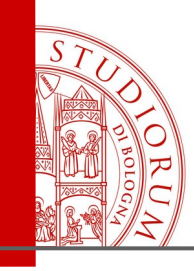
```
type Values =
  EitherT [Char] []

eval :: Exp -> Values Int
...
Eval (Times e1 e2) =
  do
      x1 <- eval e1
      x2 <- eval e2
      return (x1 * x2)

eval (Div  e1 e2) =
 do
   x1 <- eval e1
   x2 <- eval e2
   if x2 == 0 then
      error “division by zero”
   else
      return (x1 `div` x2)
```

```
eval (Const i) = return i

eval (Sqrt e1) =
   do
      x1 <- eval e1
      let x2 = isqrt x1
        returns [x2, -x2]
```

# Codice finale: con logging

```
type Values =
  EitherT [Char] (ListT (Writer [Char]))

eval :: Exp -> Values Int
...
Eval (Times e1 e2) =
  Do
      tell “times”
      x1 <- eval e1
      x2 <- eval e2
      return (x1 * x2)

eval (Div  e1 e2) =
 Do
   tell “div”
   x1 <- eval e1
   x2 <- eval e2
   if x2 == 0 then
      error “division by zero”
   else
      return (x1 `div` x2)
```

```
eval (Const i) = return i

eval (Sqrt e1) =
   do
      tell “sqrt”
      x1 <- eval e1
      let x2 = isqrt x1
        returns [x2, -x2]
```



# Monadi in Scala / Java

**No polimorfismo sui costruttori di tipo ==> solo istanze concrete di monadi che implementano map e flatMap + zucchero sintattico**

```
sealed trait Maybe[+A] {

  // >>=
  def flatMap[B](f: A => Maybe[B]): Maybe[B]
}

case class Just[+A](a: A) extends Maybe[A] {
  override def flatMap[B](f: A => Maybe[B]) = f(a)
}

case object Nothing extends Maybe[A] {
  override def flatMap[B](f: A => Maybe[B]) = Nothing
}
```

# Monadi in Scala / Java

```
def esempio(m1: Maybe[Int], m2: Maybe[Int]):
   Maybe[Int] =
{
 for {                          //  do
  a <- m1                       //   a ← m1
  b <- m2                       //   b ← m2
 } yield a + b                  //   return (a + b)
}

// is equivalent to this expression
m1.flatMap(a => m2.map(b => a + b)))
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA


Claudio Sacerdoti Coen

Dipartimento di Informatica: Scienza e Ingegneria (DISI)

claudio.sacerdoticoen@unibo.it


*www.unibo.it*