

# Haskell

- Haskell B. Curry = pioniere del  $\lambda$ -calcolo (teoria matematica delle funzioni) → FPL

- Discendente di ML {
  - Scope statico
  - Sintassi più simile a LP tradizionali che a Lisp
  - Dichiarazione di tipi
  - Inferenza di tipi
  - Tipizzazione forte (a differenza di Scheme, sostanzialmente typeless)
  - Costrutto di modularizzazione per ADT

- Differenza rispetto a ML {
  - FPL puro →  $\nexists$  variabili, assegnamento, effetti collaterali
  - Lazy evaluation**: valutazione della expr solo quando necessario
  - Possibile definire liste infinite

- Diverse implementazioni

Interpete (Hugs): <http://www.haskell.org/hugs/>

Compilatore + Interpretare (Glasgow): <http://www.haskell.org/ghc/>

# Valutazione di Espressioni

- Notazione naturale (infissa)

$2+3$

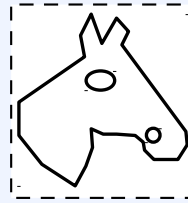
-- invece che  $(+ 2 3)$

$5*(4+6)-2$

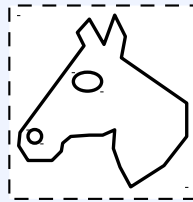
-- invece che  $(- (* 5 (+ 4 6)) 2)$

- Figura:

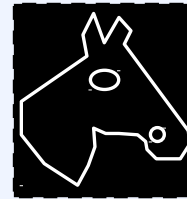
cavallo =



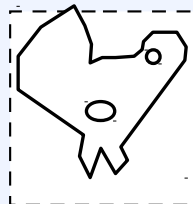
flipV cavallo  $\Rightarrow$



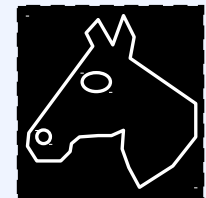
invertColour cavallo  $\Rightarrow$



flipH cavallo  $\Rightarrow$



invertColour (flipV cavallo)  $\Rightarrow$



# Definizioni

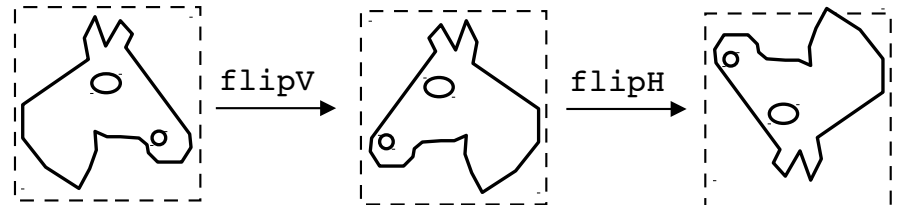
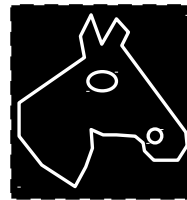
- Programma = { **definizioni** }
- Definizione = **identificatore** + **espressione** di un certo **tipo**
- Caso più semplice di definizione:

*id* :: *type*  
*id* = *expr*

```
size :: Int  
size = 12+13
```

```
cavalloNero :: Picture  
cavalloNero = invertColour cavallo
```

```
cavalloRuotato :: Picture  
cavalloRuotato = flipH (flipV cavallo)
```



# Regole di Layout

- Nello script, ogni definizione deve iniziare nella stessa colonna

```
a = b + c
  where
    b = 1
    c = 2
d = a * 2
```

- Possibile esplicitare il raggruppamento (sintassi libera dal formato)

```
a = b + c
  where
    {b = 1; c = 2};
d = a * 2
```

- Commenti: singola linea o blocco

```
-- Fattoriale di un intero positivo
fattoriale n = product [1..n]
```

```
{-
doppio x      = x + x
quadruplo x = doppio(doppio x)
-}
```

# Definizione di Funzioni

```
square :: Int -> Int
square n = n*n
```

- In generale:

tipi dei parametri formali      tipo del risultato

```
id :: type1 -> type2 -> ... -> typek -> type
```

parametri formali

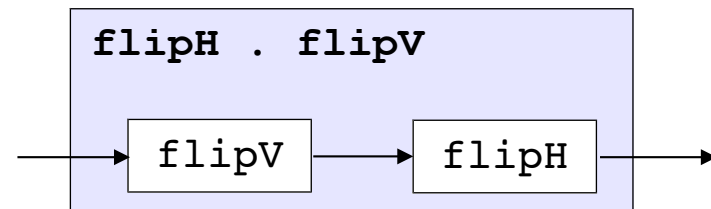
```
rotate :: Picture -> Picture
rotate figura = flipH (flipV figura)
```



```
cavalloRuotato :: Picture
cavalloRuotato = rotate cavallo
```

- Composizione di funzioni:

```
rotate :: Picture -> Picture
rotate = flipH . flipV
```



# Tipi Primitivi: Bool

- Valori = { **True**, **False** }
- Operatori = { **&&**, **||**, **not** }
- Definizione di *or* esclusivo:

x	y	exOr
True	True	False
True	False	True
False	True	True
False	False	False

```
exOr :: Bool -> Bool -> Bool
exOr x y = (x || y) && not (x && y)
```

- Definizione “artigianale” di not:

```
artNot :: Bool -> Bool
artNot True  = False
artNot False = True
```

- Definizione alternativa di exOr:

```
exOr :: Bool -> Bool -> Bool
exOr True  x = not x
exOr False x = x
```



definizione mediante **pattern-matching**

# Tipi Primitivi: **Int**

- Valori = insieme finito di interi (Integer: qualsiasi valore intero)
- Operatori = { **+**, **\***, **^**, **-**, **div**, **mod**, **abs**, **negate** }

```
2+3      -- 5
3^3      -- 27
div 10 5  -- 2
10 `div` 5 -- 2
10 `div` 3 -- 3
abs(-100) -- 100
negate(7) -- -7
```

- Operatori relazionali= { **>**, **>=**, **==**, **/=**, **<=**, **<** }

```
threeEqual :: Int -> Int -> Int -> Bool
threeEqual i j k = (i==j) && (i==k)
```

# Guardie

- Notazione alternativa per definire funzioni
- **Guardia** = expr booleana usata per esprimere un caso di definizione di funzione

```
max :: Int -> Int -> Int
max x y
  | x >= y      = x
  | otherwise   = y
```

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
  | x >= y && x >= z      = x
  | y >= z                = y
  | otherwise             = z
```

- In generale:

parametri formali

```
id p1 p2 ... pk
  | g1                = expr1
  | g2                = expr2
  | ...                = ...
  | otherwise          = expr ← ..... opzionale
```



# Espressioni Condizionali

obbligatorio



- In generale:

```
if cond then expr1 else expr2
```

```
max :: Int -> Int -> Int  
max x y =  
    if x >= y then x else y
```

```
max3 :: Int -> Int -> Int -> Int  
max3 x y z =  
    if x >= y && x >= z then x  
    else if y >= z then y  
    else z
```

# Tipi Primitivi: Char

- Valori = { 'a', 'b', ..., '\t', '\n', '\\', '\"', '\'' }

- Funzioni di conversione:

```
ord  :: Char -> Int  
chr  :: Int  -> Char
```

- Conversione di lettere:

```
offset :: Int  
offset = ord 'a' - ord 'A'  
  
toUpper :: Char -> Char  
toUpper c = chr(ord(c) - offset)  
  
toLower :: Char -> Char  
toLower c = chr(ord(c) + offset)
```

- Check di caratteri:

```
isDigit :: Char -> Bool  
isDigit c = ('0' <= c) && (c <= '9')  
  
isLetter :: Char -> Bool  
isLetter c = (('a' <= c) && (c <= 'z')) || (('A' <= c) && (c <= 'Z'))
```

# Tipi Primitivi: **Float**, **Double**

- Valori = { 0.31426, -23.12, 567.45, 13.0, ..., 231.61e7, 23.45e-2, 12.567e002, ... }

<i>Funzione</i>	<i>Protocollo</i>	<i>Significato</i>
<code>+ - * /</code>	<code>Float -&gt; Float -&gt; Float</code>	Operatori aritmetici
<code>**</code>	<code>Float -&gt; Float -&gt; Float</code>	Esponenziazione
<code>== /= &lt; &gt; &lt;= &gt;=</code>	<code>Float -&gt; Float -&gt; Bool</code>	Operatori relazionali
<code>abs</code>	<code>Float -&gt; Float</code>	Valore assoluto
<code>ceiling floor round</code>	<code>Float -&gt; Int</code>	Conversione in intero
<code>cos sin tan</code>	<code>Float -&gt; Float</code>	Funzioni trigonometriche
<code>exp</code>	<code>Float -&gt; Float</code>	Potenza di $e$
<code>log</code>	<code>Float -&gt; Float</code>	Logaritmo base $e$
<code>logBase</code>	<code>Float -&gt; Float -&gt; Float</code>	Logaritmo di base qualsiasi
<code>sqrt</code>	<code>Float -&gt; Float</code>	Radice quadrata
<code>pi</code>	<code>Float</code>	Costante $\pi$

# Operatori e Funzioni

- Operatori infissi esprimibili in forma prefissa:

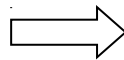
```
(+) :: Int -> Int -> Int  
(+) 2 3 ≡ 2 + 3
```

- Funzioni binarie esprimibili in forma infissa (come operatori):

```
2 `max` 3 ≡ max 2 3
```

- Definizione di nuovi operatori infissi:

```
exOr :: Bool -> Bool -> Bool  
exOr True x = not x  
exOr False x = x
```



```
(|||) :: Bool -> Bool -> Bool  
True  ||| x = not x  
False ||| x = x
```

# Ricorsione

- Fattoriale:

```
fac :: Int -> Int
fac n =
  if n == 0 then 1
  else n * fac(n-1)
```

```
fac :: Int -> Int
fac n
  | n == 0      = 1
  | otherwise   = n * fac(n-1)
```

```
fac :: Int -> Int
fac 0    = 1
fac n    = n * fac(n-1)
```

- Potenza  $2^n$ :

```
power2 :: Int -> Int
power2 n
  | n == 0      = 1
  | otherwise   = 2 * power2(n-1)
```

# Ricorsione (ii)

- Divisione fra interi positivi:

```
divide :: Int -> Int -> Int
divide n m
  | n < m      = 0
  | otherwise  = 1 + divide (n-m) m
```

- Resto di divisione fra interi positivi:

```
remainder :: Int -> Int -> Int
remainder n m
  | n < m      = n
  | otherwise  = remainder (n-m) m
```

# Ricorsione (iii)

- Somma dei fattoriali:  $0! + 1! + 2! + \dots + (n-1)! + n!$

```
sumFacs :: Int -> Int
sumFacs n
  | n == 0      = 1
  | otherwise   = sumFacs(n-1) + fac n
```

- Generalizzazione: somma delle applicazioni di f:  $f(0) + f(1) + f(2) + \dots + f(n-1) + f(n)$

```
sumF :: (Int -> Int) -> Int -> Int
sumF f n
  | n == 0      = f(0)
  | otherwise   = sumF f (n-1) + f n
```

- Specializzazione:

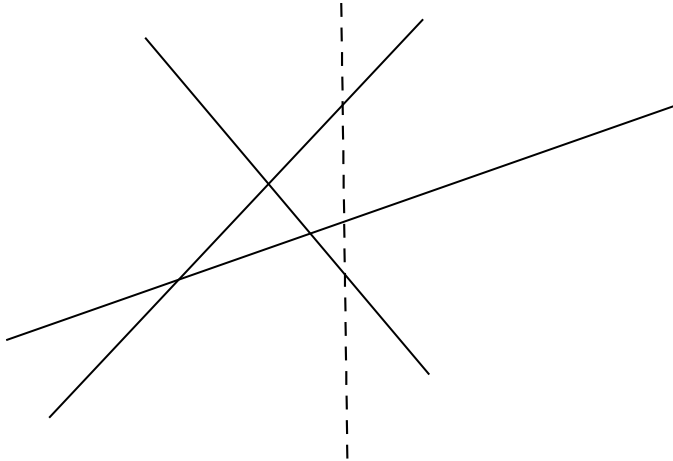
```
sumFacs :: Int -> Int
sumFacs n = sumF fac n
```

```
sumSquares :: Int -> Int
sumSquares n = sumF square n
```

```
sumFibs :: Int -> Int
sumFibs n = sumF fib n
```

# Ricorsione (iv)

- Problema geometrico: *Determinare il num max di regioni delimitate da n linee*



Linee	Regioni
0	1
1	2
2	4
3	7
4	11

- Oss: La linea **n**-esima aggiunge **n** nuove regioni alle precedenti

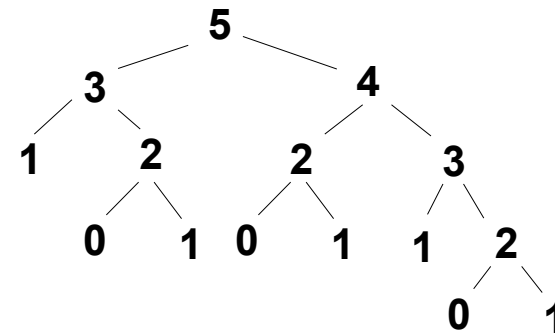
```
regioni :: Int -> Int
regioni n
  | n == 0  = 1
  | n > 0   = regioni(n-1) + n
```



# Ricorsione (v)

- Numeri di Fibonacci: *Partendo da 0, 1, i valori successivi sono la somma degli ultimi due numeri*

```
fib :: Int -> Int
fib n
  | n == 0    = 0
  | n == 1    = 1
  | n > 1     = fib(n-2) + fib(n-1)
```



- **Pb:** Ricomputazione delle stesse chiamate ( $\text{fib}(n-1) \rightarrow \text{fib}(n-2)$ )
- Calcolo del numero di chiamate per fib n:

```
fcont :: Integer -> Integer
fcont 0 = 1
fcont 1 = 1
fcont n = fcont(n-2) + fcont(n-1) + 1
```

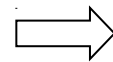
```
fconts :: Integer -> [Integer]
fconts 0 = [1]
fconts n = fconts (n-1) ++ [fcont n]
```

n	fcont n
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67
9	109
10	177
...	...
35	29860703

# Costruttori di Tipo: **Tupla**

- Tipo  $(t_1, t_2, \dots, t_n)$  ha valori  $(v_1, v_2, \dots, v_n)$  in cui  $v_i :: t_i$

```
type Articolo = (String, Int)
```



```
("Sale: 1Kg", 50)  
("Latte: 1Lt", 90)
```

- Esempi:

```
nuovoArt :: String -> Int -> Articolo  
nuovoArt nome costo = (nome, costo)
```

```
minAndMax :: Int -> Int -> (Int, Int)  
minAndMax x y  
| x <= y      = (x, y)  
| otherwise = (y, x)
```

```
ordinati :: Int -> Int -> Int -> (Int, Int, Int)  
ordinati x y z  
| x <= y && y <= z      = (x, y, z)  
| x <= z && z <= y      = (x, z, y)  
| y <= x && x <= z      = (y, x, z)  
| y <= z && z <= x      = (y, z, x)  
| z <= x && x <= y      = (z, x, y)  
| otherwise            = (z, y, x)
```

# Costruttori di Tipo: **Tupla** (ii)

- Definizione di funzioni su tuple: mediante **pattern matching**

- Esempi:

```
somma :: (Int, Int) -> Int
somma (x, y) = x + y
```

```
somma :: (Int, Int) -> Int
somma (0, y) = y
somma (x, 0) = x
somma (x, y) = x + y
```

```
shift :: ((Int, Int), Int) -> (Int, (Int, Int))
shift ((x, y), z) = (x, (y, z))
```

- Selezione di parti di tuple mediante pattern matching:

```
type Articolo = (String, Int)

nome :: Articolo -> String
prezzo :: Articolo -> Int

nome (n,p)    = n
prezzo (n,p)  = p
```

```
addPair :: (Int, Int) -> Int
addPair p = fst p + snd p
```

**selettori** per tuple di due elementi (coppie)

# Costruttori di Tipo: **Tupla** (iii)

- Numeri di Fibonacci: **0, 1, 1, 2, 3, 5, ..., u, v, (u+v), ...**

```
fib :: Int -> Int
fib n
  | n == 0    = 0
  | n == 1    = 1
  | n > 1     = fib(n-2) + fib(n-1)
```

⇒ Inefficienza: ricomputazione di `fib(n-2)` per  
computare `fib(n-1)`

↓ soluzione efficiente mediante l'uso di tuple

```
fibStep :: (Int, Int) -> (Int, Int)
fibStep (u, v) = (v, u+v)

fibPair :: Int -> (Int, Int)
fibPair n
  | n == 0      = (0, 1)
  | otherwise   = fibStep (fibPair (n-1))

fastFib :: Int -> Int
fastFib = fst . fibPair
```

<i>n</i>	<i>Coppia</i>
0	(0,1)
1	(1,1)
2	(1,2)
3	(2,3)
4	(3,5)
5	(5,8)
6	(8,13)
7	(13,21)

Idea: mantenere gli ultimi due numeri nella tupla `(u, v)`

# Costruttori di Tipo: **Lista**

- Lista = sequenza di elementi di un certo tipo:  $\forall$  tipo  $\tau \exists$  tipo  $[\tau]$  di liste di  $\tau$ , che ha valori  $[v_1, v_2, \dots, v_n]$ ,  $n \geq 0$ ,  $\forall i \in [1..n]$  ( $v_i :: \tau$ )

```
[1,2,3,4,2,3,4,8] :: [Int]
[True]           :: [Bool]
```

- **String** = sinonimo di **[Char]**

```
['c','i','a','o'] :: String
"ciao"           :: String
```

- Ortogonalità:

```
[[1,2,3],[45,53,12,68]] :: [[Int]]
[fac, square, cube]     :: [Int -> Int]
```

# Definizione di Liste mediante Range

- Requisito: lista con elementi **ordinabili**:

$$[n \dots m] = \begin{cases} [n, n+1, \dots, m] & \text{se } m \geq n \\ [] & \text{se } m < n \end{cases}$$

```
[2 .. 7]      -- [2,3,4,5,6,7]
[3.1 .. 6.1]  -- [3.1, 4.1, 5.1, 6.1]
['a' .. 'm']  -- "abcdefghijklm"
```

$[n,p \dots m] = [n, p, p+(p-n), p+2*(p-n), \dots, m]$

```
[7,6 .. 3]      -- [7,6,5,4,3]
[0.0,0.3 .. 1.0] -- [0.0,0.3,0.6,0.9]
['a','c' .. 'n'] -- "acegikm"
```


$[n \dots ] = [n, n+1, n+2, \dots]$  : lista possibilmente infinita

```
[0 ..]      -- [0,1,2 ...]
[0,2 ..]    -- [0,2,4 ...]
['\0' ..]   -- lista di tutti i caratteri
```

# Definizione Intensionale di Liste

- Mediante **generatore**:

```
[ 2*n | n <- [1 .. 10] ] -- [2,4,6,8,10,12,14,16,18,20]
```

 simbolo di appartenenza  $\in$

```
pari :: Int -> Bool
```

```
pari n = (n `mod` 2) == 0
```

```
[ pari n | n <- [1 .. 5] ] -- [False,True,False,True,False]
```

```
[ 2*n | n <- [1 .. 10], pari n, n > 3 ] -- [8,12,16,20]
```

- In generale, prima del simbolo di appartenenza ( $<-$ ): pattern invece di variabile

```
sommaCoppie :: [(Int,Int)] -> [Int]
```

```
sommaCoppie lista = [ m+n | (m,n) <- lista ]
```

```
sommaCoppie [(2,3),(2,1),(7,8)] -- [5,3,15]
```

```
sommaCoppieOrdinate :: [(Int,Int)] -> [Int]
```

```
sommaCoppieOrdinate lista = [ m+n | (m,n) <- lista, m<n ]
```

```
sommaCoppieOrdinate [(2,3),(2,1),(7,8)] -- [5,15]
```

# Definizione Intensionale di Liste (ii)

- Selezione di elementi:

```
cifre :: String -> String
cifre s = [ c | c<-s, c>='0', c<='9' ]

cifre "a2d343xy19" -- "234319"
```

- Incorporata in definizione di funzione:

```
tuttiPari :: [Int] -> Bool
tuttiDispari :: [Int] -> Bool
tuttiPari numeri = ([n | n<-numeri, pari n] == numeri)
tuttiDispari numeri = ([n | n<-numeri, pari n] == [])
```

```
esistePari :: [Int] -> Bool
esisteDispari :: [Int] -> Bool
esistePari numeri = ([n | n<-numeri, pari n] /= [])
esisteDispari numeri = ([n | n<-numeri, pari n] /= numeri)
```



# Definizione Intensionale di Liste (iii)

- Possibile dipendenza tra variabili di generatori:

```
-- Lista di tutte le coppie ordinate di elementi nella lista [1..3]
-- [(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)]
```

```
[(x,y) | x <- [1..3], y <- [x..3]]
```

```
-- Concatenazione di una lista di liste
```

```
concat :: [[a]] -> [a]
```

```
concat listaDiListe = [ x | lista <- listaDiListe, x <- lista ]
```

- Scarto di alcuni elementi dalla lista mediante pattern `_`:

```
-- Lista di tutti i primi elementi da una lista di coppie
```

```
primi :: [(a,b)] -> [a]
```

```
primi lista = [ x | (x,_) <- lista ]
```

```
-- Lunghezza di una lista
```

```
length :: [a] -> Int
```

```
length lista = sum [ 1 | _ <- lista ]
```

# Esempio: Biblioteca

- Strutture dati:

```
type Persona  = String
type Libro    = String
type Prestiti = [ (Persona, Libro) ]

prestiti :: Prestiti
prestiti = [("Alice", "TinTin"),
            ("Anna", "Piccole Donne"),
            ("Alice", "Asterix"),
            ("Rosy", "TinTin")]
```

Persona	Libro
Alice	TinTin
Anna	Piccole Donne
Alice	Asterix
Rosy	TinTin

- Requisiti funzionali:

- *Libri presi in prestito da una persona*
- *Persone che hanno preso in prestito un libro*
- *Stabilire se un libro è in prestito a qualcuno*
- *Numero di libri presi in prestito da una persona*

## Esempio: Biblioteca (ii)

- Libri presi in prestito da una persona*

```
libriInPrestito :: Persona -> [Libro]
libriInPrestito persona =
  [ l | (p, l) <- prestiti, p==persona ]
```

```
libriInPrestito "Alice" -- ["TinTin", "Asterix"]
libriInPrestito "Anna"  -- ["Piccole Donne"]
```

Persona	Libro
Alice	TinTin
Anna	Piccole Donne
Alice	Asterix
Rosy	TinTin

- Attenzione: variabili nella definizione intensionale sono sempre nuove!

```
libriInPrestito persona =
  [ l | (persona, l) <- prestiti ]
```

→ maschera il parametro omonimo della funzione → tutti i libri in prestito!

- Persone che hanno preso in prestito un libro*

```
personeConLibro :: Libro -> [Persona]
personeConLibro libro =
  [ p | (p, l) <- prestiti, l==libro ]
```

```
personeConLibro "TinTin" -- ["Alice", "Rosy"]
```

## Esempio: Biblioteca (iii)

- *Stabilire se un libro è in prestito a qualcuno*

```
inPrestito :: Libro -> Bool
inPrestito libro =
  [ l | (p, l) <- prestiti, l==libro ] /= []
```

```
inPrestito "Asterix" -- True
inPrestito "Obelix"  -- False
```

Persona	Libro
Alice	TinTin
Anna	Piccole Donne
Alice	Asterix
Rosy	TinTin

- *Numero di libri presi in prestito da una persona*

```
numeroLibri :: Persona -> Int
numeroLibri persona =
  length [ l | (p, l) <- prestiti, p==persona ]
```

```
numeroLibri "Alice" -- 2
numeroLibri "Rosy"  -- 1
numeroLibri "Zeno"  -- 0
```

# Esempio: Cifratura di Cesare

- Sostituzione di ogni lettera con la lettera tre posizioni avanti nell'alfabeto

"haskell is fun"



"kdvnhoo lv ixq"

- In generale: fattore di spostamento in [1 .. 25]
- Semplificazione: codifica solo di lettere minuscole
- Codifica di ogni lettera minuscola 'a' .. 'z' nel corrispondente numero 0 .. 25:

```
let2int :: Char -> Int  
let2int c = ord c - ord 'a'
```

```
> let2int 'a'  
0
```

- Decodifica di ogni numero 0 .. 25 nella corrispondente lettera minuscola 'a' .. 'z':

```
int2let :: Int -> Char  
int2let n = chr(ord 'a' + n)
```

```
> int2let 0  
'a'
```

## Esempio: Cifratura di Cesare (ii)

- Trasformazione di una lettera  $c$  sulla base di un fattore  $n$  di shift:

```
shift :: Int -> Char -> Char
shift n c
  | isLower c = int2let ((let2int c + n) `mod` 26)
  | otherwise = c
```

```
> shift 3 'a'
'd'
```

```
> shift 3 'z'
'c'
```

```
> shift (-3) 'c'
'z'
```

```
> shift 3 ' '
' '
```

- Cifratura di una stringa  $s$  sulla base di un fattore  $n$  di shift:

```
encode :: Int -> String -> String
encode n s = [shift n c | c <- s]
```

```
> encode 3 "haskell is fun"
"kdvnhoo lv ixq"
```

- Oss: Decodifica mediante shift negativo (non serve una nuova funzione)

```
> encode (-3) "kdvnhoo lv ixq"
"haskell is fun"
```

## Esempio: Cifratura di Cesare (iii)

- **Tabelle di frequenza:** statistiche sull'uso dei caratteri dell'alfabeto nel linguaggio:

```
table :: [Float]
table = [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0, 6.1, 7.0, 0.2, 0.8, 4.0, 2.4,
        6.7, 7.5, 1.9, 0.1, 6.0, 6.3, 9.1, 2.8, 1.0, 2.4, 0.2, 2.0, 0.1]
```

- Tabelle di frequenza per singole stringhe:

→ converte da intero a reale

```
percent :: Int -> Int -> Float
percent n m = (fromIntegral n / fromIntegral m) * 100
```

```
lowers :: String -> Int
lowers s = length [c | c <- s, isLower c]
```

```
count :: Char -> String -> Int
count c s = length [c' | c' <- s, c == c']
```

```
freqs :: String -> [Float]
freqs s = [percent(count c s) n | c <- ['a'..'z']]
          where n = lowers s
```

```
> freqs "abbccdddeeeee"
[6.7, 13.3, 20.0, 26.7, 33.3, 0.0, 0.0, ..., 0.0]
```

## Esempio: Cifratura di Cesare (iv)

- Confronto fra due liste di frequenze  $\begin{cases} \text{osservate (o)} \\ \text{attese (a)} \end{cases} \xrightarrow{\text{yellow arrow}} \text{distribuzione } \chi^2:$

$$\sum_{i=0}^{n-1} \frac{(o_i - a_i)^2}{a_i} \quad \text{matching migliore per valore minimo}$$



```
chisqr :: [Float] -> [Float] -> Float
chisqr oss att = sum [((o - a)^2)/a | (o,a) <- zip oss att]
```

- Rotazione di una lista di  $n$  posizioni:

```
rotate :: Int -> [a] -> [a]
rotate n lista = drop n lista ++ take n lista
```

```
> rotate 3 [1,2,3,4,5]
[4,5,1,2,3]
```



# Esempio: Cifratura di Cesare (v)

- Problema: Data una stringa cifrata, determinare il fattore di shift.
- Soluzione:
  1. Generazione della tabella delle frequenze (della stringa cifrata);
  2. Calcolo della distribuzione  $\chi^2$  per ogni possibile rotazione della tabella rispetto alla tabella delle frequenze attese;
  3. Fattore di shift = posizione del valore minimo della distribuzione  $\chi^2$ .

```
table' = freqs "kdvnhoo lv ixq"  
[chisqr (rotate n table') table | n <- [0..25]]
```



```
[1408.8, 640.3, 612.4, 202.6, 1439.8, 4247.2, 651.3, ..., 626.7]
```

3

# Esempio: Cifratura di Cesare (vi)

- Lista delle posizioni delle occorrenze di `x` in una `lista`:

```
positions :: Eq a => a -> [a] -> [Int]
positions x lista = [i | (x',i) <- zip lista [0..n], x == x']
                    where n = length lista - 1
```

```
> positions False [True, False, True, False]
[1,3]
```

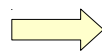
- Decodifica della stringa cifrata `s`:

```
crack :: String -> String
crack s = encode (-factor) s
  where
    factor = head (positions (minimum chitab) chitab)
    chitab = [chisqr (rotate n table') table | n <- [0..25]]
    table' = freqs s
```

```
> crack "kdvnhoo lv ixq"
"haskell is fun"
```

```
> crack (encode 3 "haskell is fun")
"haskell is fun"
```

```
> crack (encode 3 "haskell")
"piasmтт"
```

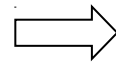


inefficace quando distribuzione  
di lettere anomala

# Funzioni Generiche: Polimorfismo

- Funzione polimorfa: possiede svariati (anche infiniti) tipi:

```
length :: [Bool] -> Int
length :: [[Char]] -> Int
length :: [(Int,String)] -> Int
length :: [[[[Float]]]] -> Int
...
```



```
length :: [a] -> Int
```

- Uso di **variabili di tipo**:  $a, b, c, \dots$  ; denotano un qualsiasi tipo

- Istanziamento:

```
a ⇒ Int
a ⇒ Char
a ⇒ [Float]
a ⇒ [(Int,String)]
[a] -> Int ⇒ [Bool] -> Int
[a] -> Int ⇒ [(Int,String)] -> Int
```

- Tipo più generale di una funzione polimorfa (length): `[a] -> Int`

# Funzioni Generiche: Polimorfismo (ii)

- Concatenazione di liste: `(++) :: [a] -> [a] -> [a]`



```
[a] -> [a] -> [a] ⇒ [Int] -> [Int] -> [Int]
                    ⇒ [(Int,String)] -> [(Int,String)] -> [(Int,String)]
                    ⇒ [[Float]] -> [[Float]] -> [[Float]]
                    ⇒ [Int] -> [Int] -> [Char]
```

- Creazione di lista di coppie:

```
zip :: [a] -> [b] -> [(a,b)]
```

→

```
[Int] -> [Bool] -> [(Int,Bool)]
[Int] -> [Int] -> [(Int,Int)]
[a] -> [a] -> [(a,a)]
```

- Creazione di coppia di liste:

```
unzip :: [(a,b)] -> ([a],[b])
```

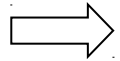
→

```
[(Int,Bool)] -> ([Int],[Bool])
[(Int,Int)] -> ([Int],[Int])
[(a,a)] -> ([a],[a])
```

# Funzioni Generiche: Polimorfismo (iii)

- *Come si definiscono le funzioni polimorfe?*

```
identita :: a -> a  
identita x = x
```



Unico vincolo: tipo input = tipo output

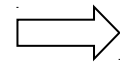
```
fst :: (a,b) -> a  
fst(x,y) = x
```



~~∃~~ correlazione tra a e b

- Inferenza dei tipi:

```
mistero (x,y) = if x then 'c' else 'd'
```



```
(Bool,a) -> Char
```

- Polimorfismo e overloading: stesso nome di funzione applicato a tipi diversi, però:
  - Funzione polimorfa (`fst`) → unica definizione
  - Funzione overloaded (`==`) → diverse definizioni

# Funzioni di Libreria per Liste (Prelude.hs)

Funzione	Protocollo	Significato
:	<code>a -&gt; [a] -&gt; [a]</code>	Aggiunta di un elemento in testa alla lista
++	<code>[a] -&gt; [a] -&gt; [a]</code>	Concatenazione di due liste
!!	<code>[a] -&gt; Int -&gt; a</code>	<code>x !! n</code> indica l'n-esimo elemento della lista <code>x</code>
concat	<code>[[a]] -&gt; [a]</code>	Concatenazione di una sequenza di liste
length	<code>[a] -&gt; Int</code>	Lunghezza della lista
head, last	<code>[a] -&gt; a</code>	Primo/ultimo elemento della lista
tail, init	<code>[a] -&gt; [a]</code>	Lista privata del primo/ultimo elemento
replicate	<code>Int -&gt; a -&gt; [a]</code>	Creazione di una lista composta da <i>n</i> occorrenze di elem
take	<code>Int -&gt; [a] -&gt; [a]</code>	Preleva i primi <i>n</i> elementi della lista
drop	<code>Int -&gt; [a] -&gt; [a]</code>	Cancella i primi <i>n</i> elementi della lista
splitAt	<code>Int -&gt; [a] -&gt; ([a],[a])</code>	Divisione della lista in una certa posizione
reverse	<code>[a] -&gt; [a]</code>	Inversione della lista
zip	<code>[a] -&gt; [b] -&gt; [(a,b)]</code>	Creazione di lista di coppie
unzip	<code>[(a,b)] -&gt; ([a],[b])</code>	Creazione di coppia di liste
sum	<code>[Int] -&gt; Int</code> <code>[Float] -&gt; Float</code>	Somma gli elementi della lista
product	<code>[Int] -&gt; Int</code> <code>[Float] -&gt; Float</code>	Moltiplica gli elementi della lista

# Funzioni di Libreria per Liste (Prelude.hs) (ii)

```
pari      = [0,2,4,6,8]
dispari   = [1,3,5,7,9]
```

```
5:pari    -- [5,0,2,4,6,8]
5:[]      -- [5]
pari ++ dispari -- [0,2,4,6,8,1,3,5,7,9]
pari !! 3  -- 6
concat [pari,dispari,[10,20,30]] -- [0,2,4,6,8,1,3,5,7,9,10,20,30]
length pari -- 5
head pari   -- 0
last pari   -- 8
tail pari   -- [2,4,6,8]
init pari   -- [0,2,4,6]
replicate 10 True -- [True,True,True,True,True,True,True,True,True,True]
take 3 pari  -- [0,2,4]
drop 3 pari  -- [6,8]
splitAt 3 pari -- ([0,2,4],[6,8])
reverse pari  -- [8,6,4,2,0]
zip pari dispari -- [(0,1),(2,3),(4,5),(6,7),(8,9)]
unzip [(0,1),(2,3),(4,5),(6,7),(8,9)] -- ([0,2,4,6,8],[1,3,5,7,9])
sum pari     -- 20
product dispari -- 945
```

# Tipo String

- Caso speciale di lista: `type String = [Char]`

`"ciao"`  $\Leftrightarrow$  `['c','i','a','o']`

`"alfa" ++ "beto"`  $\Leftrightarrow$  `['a','l','f','a',]++['b','e','t','o']`  $\Leftrightarrow$  `"alfabeto"`

- Conversione stringa  $\Leftrightarrow$  valore: `show` e `read`

```
show (2+3) -- "5"
show (True || False) -- "True"
read "True" -- True
read "3" -- 3
(read "[1,2]") :: [Int] -- [1,2]
```



# Programmazione con Liste: Picture

```
type Picture = [String]

flipH :: Picture -> Picture
flipH = reverse

-- sovrapposizione di figure
above :: Picture -> Picture -> Picture
above = (++)

flipV :: Picture -> Picture
flipV pic = [ reverse linea | linea <- pic ]

-- accostamento di figure
sideByside :: Picture -> Picture -> Picture
sideByside pic1 pic2 = [ line1 ++ line2 | (line1,line2) <- zip pic1 pic2 ]
```

```
.....
.....#####
.....#.....#
.....#.....#
.....#.....#
.....#####
.....
```

# Programmazione con Liste: Picture (ii)

```
-- Inversione di colore
```

```
invertChar :: Char -> Char
```

```
invertChar '.' = '#'
```

```
invertChar '#' = '.'
```

```
invertLine :: [Char] -> [Char]
```

```
invertLine line = [ invertChar c | c <- line ]
```

```
invertColour :: Picture -> Picture
```

```
invertColour pic = [ invertLine line | line <- pic ]
```

```
.....#####  
.....####..  
.....#...#..  
.....#...#..  
.....#...#..  
.....#...#..  
...#####..  
.....
```



```
-- Inversione di colore
```

```
invertColour :: Picture -> Picture
```

```
invertColour pic =
```

```
  [ [ if c == '.' then '#' else '.' | c <- line ] | line <- pic ]
```

# Definizioni Locali nella Specifica di Funzioni

- Somma del quadrato di due numeri:

```
sumSquares :: Int -> Int -> Int
sumSquares n m = n2 + m2
    where
        n2 = n*n
        m2 = m*m
```

- Somma di elementi corrispondenti: `addCor [1,7] [8,4,2,7] -- [9,11]`

```
addCor :: [Int] -> [Int] -> [Int]
addCor list1 list2 =
    [ m+n | (m,n) <- zip list1 list2 ]
```

- Elementi non sommati → in coda: `addCor' [1,7] [8,4,2,7] -- [9,11,2,7]`

```
addCor' :: [Int] -> [Int] -> [Int]
addCor' list1 list2 =
    front ++ back
    where
        lungMin = min (length list1) (length list2)
        front   = addCor list1 list2
        back    = drop lungMin list1 ++ drop lungMin list2
```

# Pattern Matching

- Distinzione fra diversi casi nella definizione di funzione:

```
funzione x y
  | x == 0      = y
  | otherwise   = x
```

```
funzione 0 y = y
funzione x y = x
```

```
funzione 0 y = y
funzione x _ = x
```

- Identificazione di componenti di tupla: 

```
joinStr :: (String, String) -> String
joinStr (s, s') = s ++ "\t" ++ s'
```

- Identificazione di parti di lista: 

```
head :: [a] -> a
head (testa:_) = testa
```

```
tail :: [a] -> [a]
tail (_:coda) = coda
```

- Possibili pattern:

- Valore **letterale**: 24, 'a', 0.15
- **Variabile**: x, lista, cavalloNero
- **Underscore**:
- Pattern **tupla**: (p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>) in cui i p<sub>i</sub> sono pattern
- Pattern **lista**: [p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>] in cui i p<sub>i</sub> sono pattern
- Pattern **costruttore lista**: testa:coda in cui testa e coda sono pattern

# Pattern per Liste

- Lista vuota: `[]`

- Lista non vuota:

```
(:) :: a -> [a] -> [a]    -- costruttore di liste  
[2,3,4] = 2:[3,4] = 2:3:[4] = 2:3:4:[]
```

- Pattern per liste:

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

```
sum :: [Int] -> Int  
sum [] = 0  
sum(testa:coda) = testa + sum coda
```

```
(++) :: [a] -> [a] -> [a]  
[] ++ lista      = lista  
(testa:coda) ++ lista = testa:(coda ++ lista)
```

```
concat :: [[a]] -> [a]  
concat [] = []  
concat(testa:coda) = testa ++ concat coda
```

```
elem :: Int -> [Int] -> Bool  
elem n []      = False  
elem n (testa:coda) = (n == testa) || (elem n coda)
```

- Unica occorrenza di ogni variabile in pattern:

```
elem n (n:_)      = True  
elem n (_:coda) = elem n coda
```

# Pattern per Liste (ii)

- Raddoppio di ogni elemento di una lista: `raddoppio :: [Int] -> [Int]`

```
raddoppio lista = [ 2*n | n <- lista ]
```

```
raddoppio []      = []  
raddoppio (n:coda) = (2*n) : raddoppio coda
```

- Selezione dei numeri pari: `selPari :: [Int] -> [Int]`

```
selPari lista = [ n | n <- lista, isEven n ]
```

```
selPari []      = []  
selPari (n:coda)  
  | isEven n    = n : selPari coda  
  | otherwise   = selPari coda
```

- Ordinamento di lista di numeri per inserzione: `iSort :: [Int] -> [Int]`

```
iSort []      = []  
iSort (n:coda) = ins n (iSort coda)
```

```
ins :: Int -> [Int] -> [Int]  
ins n [] = [n]  
ins n (testa:coda)  
  | n <= testa = n:(testa:coda)  
  | otherwise  = testa : ins n coda
```

## Pattern per Liste (iii)

- Creazione di liste di coppie: `zip :: [a] -> [b] -> [(a,b)]`

```
zip [1,3] ['a','b','c'] ; [(1,'a'),(3,'b')]
```

```
zip (x:coda) (x':coda') = (x,x') : zip coda coda'  
zip _ _ = []
```

- Prelievo di un prefisso della lista: `take :: Int -> [a] -> [a]`

```
take 4 "alfabeto" -- "alfa"
```

```
take 0 _ = []  
take _ [] = []  
take n (testa:coda)  
  | n>0 = testa : take (n-1) coda
```

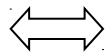
- Quicksort di una lista di numeri: `qSort :: [Int] -> [Int]`

```
qSort [] = []  
qSort (n:coda) =  
  qSort [n' | n' <- coda, n'<=n] ++ [n] ++ qSort [n' | n' <- coda, n'>n]
```

# Forme Funzionali

- Protocollo delle funzioni definite mediante la notazione **currying**
- Formalmente: funzione applicabile ad un solo argomento

```
add :: Int -> Int -> Int
add x y = x + y
```



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

- Formalmente: add è una funzione applicata ad un intero x, che restituisce una funzione applicata ad un intero y che restituisce x+y

```
add 3 4 ≡ (add 3) 4
```

- Moltiplicazione di tre numeri:

```
mult :: Int -> Int -> Int -> Int
mult x y z = x * y * z
```

```
mult :: Int -> (Int -> (Int -> Int))
mult = \x -> (\y -> (\z -> x * y * z))
```

```
mult 3 4 5 ≡ ((mult 3) 4) 5
```



## Forme Funzionali (ii)

- Funzione come argomento di un'altra funzione

```
twice :: (a -> a) -> a -> a
twice f x = f(f x)
```



```
twice (*2) 3      -- 12
twice reverse [1,2,3] -- [1,2,3]
```

- Applicazione parziale di twice → nuova funzione

```
twice (*2) -- funzione che quadruplica il suo argomento
```

```
quadruplo :: Int -> Int
quadruplo = twice (*2)
```

# Forme Funzionali (iii)

- Applicazione universale: `map :: (a -> b) -> [a] -> [b]`

```
map f lista = [ f x | x <- lista ]
```

```
map f [] = []  
map f (testa:coda) = f testa : map f coda
```



```
convertChars :: [Char] -> [Int]  
convertChars list = map ord list
```

```
flipV :: Picture -> Picture  
flipV pic = map reverse pic
```

```
map (+1) [1,3,5,7] -- [2,4,6,8]
```

```
.....  
.....#####  
.....#....#  
.....#....#  
.....#....#  
.....#....#  
.....#####  
.....  
.....  
...#####  
...#....#  
...#....#  
...#....#  
...#....#  
...#####  
.....
```

- Gestione di liste annidate:

```
map(map(+1))[[1,2,3],[4,5]] -- [[2,3,4],[5,6]]
```

# Forme Funzionali (iv)

- Selezione: 

```
select :: (a -> Bool) -> [a] -> [a]
select p lista = [ x | x <- lista, p x ]
```

```
select isEven [1,2,3,4,5,6,7,8,9,10]      -- [2,4,6,8,10]
select isDigit "a2beta34gamma5"          -- "2345"
select isSorted [[2,3,4,5],[3,1,6],[2],[ ]] -- [[2,3,4,5],[2],[ ]]
```

```
isSorted :: [Int] -> Bool
isSorted [] = True
isSorted [_] = True
isSorted (n:(n2:coda2)) = n <= n2 && isSorted (n2:coda2)
```

- Applicazione di funzione ad elementi corrispondenti in liste:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:cx) (y:cy) = f x y : zipWith f cx cy
zipWith f _ _ = []
```

```
zipWith (+) [1,2,3,4] [5,6,7,8,9,10]      -- [6,8,10,12]
zipWith (++) ["alfa", "beta"] ["delta", "zeta"] -- ["alfadelta", "betazeta"]
```

- Combinazione di map e select: somma dei quadrati dei numeri pari di una lista

```
sommaQuadPari :: [Int] -> Int
sommaQuadPari lista = sum(map(^2)(select isEven lista))
```

# Forme Funzionali: **foldr**

- Pattern ricorrente per definire funzioni di liste (in cui  $\oplus$  è un operatore binario):

```
f [] = v  
f (testa:coda) = testa  $\oplus$  (f coda)
```

```
sum [] = 0  
sum (testa:coda) = testa + (sum coda)
```

```
product [] = 1  
product (testa:coda) = testa * (product coda)
```

```
or [] = False  
or (testa:coda) = testa || (or coda)
```

```
and [] = True  
and (testa:coda) = testa && (and coda)
```

# Forme Funzionali: **foldr** (ii)

- Incapsulamento del pattern nella funzione **foldr**  $\oplus$  **v**

```
sum      = foldr (+) 0
product  = foldr (*) 1
or       = foldr (||) False
and      = foldr (&&) True
```

- Definizione di **foldr**:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op v []           = v
foldr op v (testa:coda) = op testa (foldr op v coda)
```

- Intuitivamente: nella lista, sostituzione di ':' con **op** e [] con **v**

```
foldr (+) 0 [1,2,3]  ⇔  foldr (+) 0 1:(2:(3:[]))  ⇒  1+(2+(3+0))
```

- In generale:

$$\text{foldr } (\oplus) \text{ v } [x_0, x_1, \dots, x_n] = x_0 \oplus (x_1 \oplus (\dots (x_n \oplus v) \dots))$$

# Forme Funzionali: **foldl**

- Operatore associato a sinistra

```
sum  =  sum' 0
      where
          sum' v [] = v
          sum' v (testa:coda) = sum' (v + testa) coda
```

```
sum [1,2,3] = sum' 0 [1,2,3]
            = sum' (0+1) [2,3]
            = sum' ((0+1)+2) [3]
            = sum' (((0+1)+2)+3) []
            = ((0+1)+2)+3
            = 6
```

## Forme Funzionali: **foldl** (ii)

- Pattern ricorrente per definire funzioni di liste (in cui  $\oplus$  è un operatore binario):

```
f v [] = v
f v (testa:coda) = f (v  $\oplus$  testa) coda
```

- Incapsulamento del pattern nella funzione **foldl**  $\oplus$  v

```
sum      = foldl (+) 0
product  = foldl (*) 1
or        = foldl (||) False
and       = foldl (&&) True
```

- Definizione di **foldl**:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl op v [] = v
foldl op v (testa:coda) = foldl op (op v testa) coda
```

- In generale:  $\text{foldl } (\oplus) \ v \ [x_0, x_1, \dots, x_n] = (\dots((v \oplus x_0) \oplus x_1) \dots) \oplus x_n$

# Forme Funzionali: Composizione

- Definizione di composizione di funzioni:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

- Definizione di funzioni mediante composizione:

```
isOdd n = not (isEven n)
twice f x = f (f x)
sommaQuadPari lista = sum (map (^2) (select isEven lista))
```



```
isOdd      = not . isEven
twice f    = f . f
sommaQuadPari = sum . map (^2) . select isEven
```

- Definizione di `sommaQuadPari` sfrutta la proprietà associativa della composizione:

```
f . (g . h) = (f . g) . h
```



# Forme Funzionali: Composizione (ii)

- Identità della composizione: funzione `id`

```
id :: a -> a
id = \x -> x
```

- Proprietà:

```
id . f = f
f . id = f
```

- Definizione della composizione di una lista di funzioni:

```
compose :: [a->a] -> (a->a)
compose = foldr (.) id
```

oppure

```
compose :: [a->a] -> (a->a)
compose = foldl (.) id
```



```
foldr (⊕) v [x0, x1, ..., xn] = x0 ⊕ (x1 ⊕ (... (xn ⊕ v) ...))
```



```
foldr (.) id [f0, f1, ..., fn] = f0 . (f1 . (... (fn . id) ...))
```

# Forme Funzionali: Trasmissione di Stringhe

- Problema: trasmissione di una stringa mediante sequenza di cifre binarie
- Definizione di due funzioni: codifica + decodifica
- Assunzione semplificativa: numeri binari scritti in ordine inverso:

$$1101 \rightarrow 1011 = (1*1)+(2*0)+(4*1)+(8*1) = 13$$

- Conversione numero binario  $\rightarrow$  numero intero

```
type Bit = Int

bin2int :: [Bit] -> Int
bin2int bits = sum [ p*b | (p, b) <- zip pesi bits ]
    where
        pesi = iterate (*2) 1
```

dove `iterate f x = [ x, f x, f(f x), f(f(f x)), ... ]`

```
iterate (*2) 1      -- [1, 2, 4, 8, ...]
bin2int [1,0,1,1]  -- 13
```

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

# Forme Funzionali: Trasmissione di Stringhe (ii)

- Definizione più semplice di `bin2int` sulla base di proprietà algebriche:

```
bin2int [a,b,c,d]  →  (1 * a) + (2 * b) + (4 * c) + (8 * d)
                   =  a + (2 * b) + (4 * c) + (8 * d)
                   =  a + 2 * (b + (2 * c) + (4 * d))
                   =  a + 2 * (b + 2 * (c + (2 * d)))
                   =  a + 2 * (b + 2 * (c + (2 * (d + (2 * 0)))))
```



```

$$x_0 \oplus (x_1 \oplus (\dots (x_n \oplus v) \dots))$$

```



```
bin2int :: [Bit] -> Int
bin2int = foldr (\x y -> x + 2 * y) 0
```

# Forme Funzionali: Trasmissione di Stringhe (iii)

- Conversione numero intero  $\rightarrow$  numero binario: ripetizione della divisione dell'intero per 2  $\rightarrow$  resto = cifra binaria, finché l'intero diventa 0

```
13 div 2 = 6  $\rightarrow$  1
 6 div 2 = 3  $\rightarrow$  0
 3 div 2 = 1  $\rightarrow$  1
 1 div 2 = 0  $\rightarrow$  1
```

```
int2bin :: Int -> [Bit]
int2bin 0 = []
int2bin n = n `mod` 2 : int2bin (n `div` 2)
```

- Vincolo: numeri binari con stessa lunghezza = 8 bit  $\rightarrow$  funzione di formattazione

```
make8 :: [Bit] -> [Bit]
make8 bits = take 8 (bits ++ repeat 0)
```

dove `repeat 0 = [0,0,0, ...]`

```
make8 [1,0,1,1] -- [1,0,1,1,0,0,0,0]
```

```
repeat :: a -> [a]
repeat x = lista
          where lista = x:lista
```

oppure

```
repeat x = x:(repeat x)
```

# Forme Funzionali: Trasmissione di Stringhe (iv)

- **Codifica** di stringa di caratteri in lista di bit mediante funzione encode:
  1. Carattere → numero Unicode
  2. Numero → numero binario di 8 bit (byte)
  3. Concatenazione di tutti i byte in una lista di bit

```
encode :: String -> [Bit]
encode = concat . map (make8 . int2bin . ord)
```

```
encode "abc"  -- [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

- **Decodifica** → taglio della lista di bit in byte mediante la funzione chop8:

```
chop8 :: [Bit] -> [[Bit]]
chop8 []      = []
chop8 bits    = (take 8 bits):chop8(drop 8 bits)
```

```
chop8 [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
-- [[1,0,0,0,0,1,1,0],[0,1,0,0,0,1,1,0],[1,1,0,0,0,1,1,0]]
```

# Forme Funzionali: Trasmissione di Stringhe (v)

- Decodifica di una lista di bit in una stringa di caratteri mediante decode:

```
decode :: [Bit] -> String
decode = map (chr . bin2int) . chop8
```

```
decode [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
-- "abc"
```

- Simulazione della trasmissione di una lista di caratteri come lista di bit, attraverso un canale perfetto, mediante la funzione transmit

```
transmit :: String -> String
transmit = decode . channel . encode
```

```
channel :: [Bit] -> [Bit]
channel = id
```

```
transmit "le forme funzionali sono sorprendenti"
-- "le forme funzionali sono sorprendenti"
```

# Dichiarazione di Tipi

- Dichiarazione di tipi sinonimi: **type**

```
type String = [Char]
```

```
type Board = [Pos]  
type Pos = (Int,Int)
```

- Dichiarazioni **type**: non possono essere ricorsive

```
type Tree = (Int,[Tree])
```

- Dichiarazioni **type**: possono essere parametrizzate

```
type Assoc k v = [(k,v)]
```

```
find :: Eq k => k -> Assoc k v -> v  
find key tab = head [v | (k,v) <- tab, k == key]
```

# Dichiarazione di Tipi (ii)

- Dichiarazione di nuovi tipi: **data**

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

- Valori del tipo: **costruttori** (non riusabili in altri tipi)
- Uso dei costruttori come normali valori:

```
data Move = Left | Right | Up | Down
```

```
move :: Move -> Pos -> Pos
```

```
move Left (x,y)  = (x-1, y)
```

```
move Right (x,y) = (x+1, y)
```

```
move Up (x,y)    = (x, y-1)
```

```
move Down (x,y)  = (x, y+1)
```

```
moves :: [Move] -> Pos -> Pos
```

```
moves [] p      = p
```

```
moves (m:coda) p = moves coda (move m p)
```

```
flip :: Move -> Move
```

```
flip Left  = Right
```

```
flip Right = Left
```

```
flip Up    = Down
```

```
flip Down  = Up
```



# Dichiarazione di Tipi (iii)

- Dichiarazioni **data**: possono essere parametrizzate

```
data Shape = Circle Float | Rect Float Float

square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

- Oss: Circle e Rect sono effettivamente funzioni costruttore:

```
> :type Circle
Circle :: Float -> Shape
```

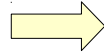
```
> :type Rect
Rect :: Float -> Float -> Shape
```

- Differenza rispetto alle normali funzioni: costruttori senza corpo (equazioni)  
(esistono solo per costruire dati)

# Dichiarazione di Tipi (iv)

- Dichiarazioni **data**: possono essere **ricorsive** → infiniti valori!

```
data Nat = Zero | Succ Nat
```



```
Zero  
Succ Zero  
Succ (Succ Zero)  
Succ (Succ (Succ Zero))  
...
```

- Funzioni di conversione:

```
nat2int :: Nat -> Int  
nat2int Zero = 0  
nat2int (Succ n) = 1 + nat2int n
```

```
int2nat :: Int -> Nat  
int2nat 0 = Zero  
int2nat n = Succ (int2nat (n-1))
```

- Somma di Nat:

```
add :: Nat -> Nat -> Nat  
add m n = int2nat (nat2int m + nat2int n)
```

oppure (senza conversioni):

```
add Zero n = n  
add (Succ m) n = Succ (add m n)
```

# Dichiarazione di Tipi (v)

- Definizione artigianale del tipo lista di elementi di tipo `a`:

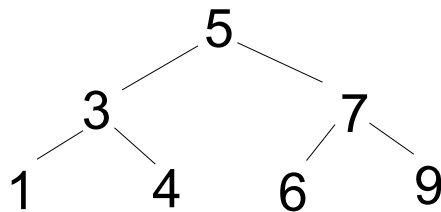
```
data List a = Nil | Cons a (List a)
```

- Definizione artigianale della funzione lunghezza della lista:

```
len :: List a -> Int  
len Nil = 0  
len (Cons _ coda) = 1 + len coda
```

- Albero binario

```
data Tree = Leaf Int | Node Tree Int Tree
```



```
albero :: Tree  
albero = Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9))
```

# Dichiarazione di Tipi (vi)

- Esistenza di un numero nell'albero:

```
occurs :: Int -> Tree -> Bool
occurs m (Leaf n)    = m == n
occurs m (Node left n right) = m == n || occurs m left || occurs m right
```

Oss: se non esiste → attraversamento di tutto l'albero

- Appiattimento di un albero in una lista:

```
flatten :: Tree -> [Int]
flatten (Leaf n)          = [n]
flatten (Node left n right) = flatten left ++ [n] ++ flatten right
```

```
> flatten albero
[1,3,4,5,6,7,9]
```

Oss: quando la lista ottenuta è ordinata → **search tree**

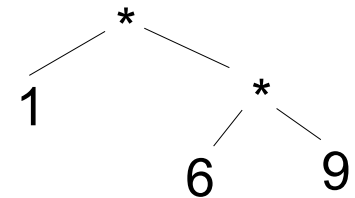
- Esistenza di un numero nel search tree (più efficiente):

```
occurs' m (Leaf n)    = m == n
occurs' m (Node left n right)
  | m == n    = True
  | m < n     = occurs' m left
  | otherwise = occurs' m right
```

# Dichiarazione di Tipi (vii)

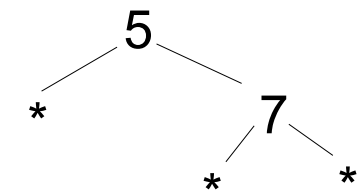
- Alberi binari con dati solo nelle foglie:

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



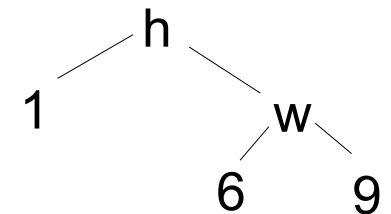
- Alberi binari con dati solo nei nodi:

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



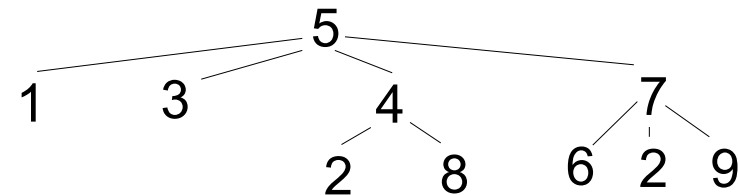
- Alberi binari con dati sia in nodi che in foglie (eterogenei):

```
data Tree a b = Leaf a | Node (Tree a b) b (Tree a b)
```



- Alberi senza limiti di prole:

```
data Tree a = Node a [Tree a]
```



# Verifica di Tautologie

- **Tautologia** = Proposizione logica sempre vera ( $\forall$  assegnamento delle var logiche)

$$A \wedge \neg A$$

$$(A \wedge B) \Rightarrow A$$

$$A \Rightarrow (A \wedge B)$$

$$(A \wedge (A \Rightarrow B)) \Rightarrow B$$

- Tabelle di verità degli operatori logici:

A	$\neg A$
F	T
T	F

A	B	$A \wedge B$
F	F	F
F	T	F
T	F	F
T	T	T

A	B	$A \Rightarrow B$
F	F	T
F	T	T
T	F	F
T	T	T

- Tabelle di verità delle proposizioni:

A	$A \wedge \neg A$
F	F
T	F

A	B	$(A \wedge B) \Rightarrow A$
F	F	T
F	T	T
T	F	T
T	T	T

A	B	$A \Rightarrow (A \wedge B)$
F	F	T
F	T	T
T	F	F
T	T	T

A	B	$(A \wedge (A \Rightarrow B)) \Rightarrow B$
F	F	T
F	T	T
T	F	T
T	T	T

# Verifica di Tautologie (ii)

- Definizione del tipo *proposizione*:

```
data Prop = Const Bool
          | Var Char
          | Not Prop
          | And Prop Prop
          | Imply Prop Prop
```

$A \wedge \neg A$

```
p1 :: Prop
p1 = And(Var 'A')(Not (Var 'A'))
```

$(A \wedge B) \Rightarrow A$

```
p2 :: Prop
p2 = Imply(And(Var 'A')(Var 'B'))(Var 'A')
```

$A \Rightarrow (A \wedge B)$

```
p3 :: Prop
p3 = Imply(Var 'A')(And(Var 'A')(Var 'B'))
```

$(A \wedge (A \Rightarrow B)) \Rightarrow B$

```
p4 :: Prop
p4 = Imply(And(Var 'A')(Imply(Var 'A')(Var 'B')))(Var 'B')
```

# Verifica di Tautologie (iii)

- *Sostituzione* = tabella associativa tra variabili e valori

```
type Assoc k v = [(k,v)]
```

```
find :: Eq k => k -> Assoc k v -> v  
find key tab = head [v | (k,v) <- tab, k == key]
```

```
type Subst = Assoc Char Bool
```

- Valutazione della proposizione in base alla tabella di sostituzione:

```
eval :: Subst -> Prop -> Bool  
eval _ (Const b)      = b  
eval s (Var x)         = find x s  
eval s (Not p)          = not(eval s p)  
eval s (And p q)        = eval s p && eval s q  
eval s (Imply p q)      = eval s p <= eval s q
```



# Verifica di Tautologie (iv)

- Lista (con duplicati) delle variabili coinvolte in una proposizione:

```
vars :: Prop -> [Char]
vars (Const _)    = []
vars (Var x)       = [x]
vars (Not p)       = vars p
vars (And p q)     = vars p ++ vars q
vars (Imply p q)   = vars p ++ vars q
```

$(A \wedge B) \Rightarrow A$

```
p2 :: Prop
p2 = Imply(And(Var 'A')(Var 'B'))(Var 'A')
```

```
> vars p2
['A', 'B', 'A']
```

# Verifica di Tautologie (v)

- Necessario poter generare la lista di tutte le possibili sostituzioni per  $n$  variabili:

```
bools :: Int -> [[Bool]]
```

```
> bools 3
[[False, False, False], 0
 [False, False, True],   1
 [False, True, False],   2
 [False, True, True],    3
 [True, False, False],   4
 [True, False, True],    5
 [True, True, False],    6
 [True, True, True]]     7
```

```
bools n = map (map conv . make n . int2bin) [0..limit]
  where
    limit = (2 ^ n) - 1;
    make n bs = take n (bs ++ repeat 0)
    conv 0 = False
    conv 1 = True
```

# Verifica di Tautologie (vi)

- Alternativa più semplice (ricorsiva) alla definizione di `bools`:

False	False	False
False	False	True
False	True	False
False	True	True
True	False	False
True	False	True
True	True	False
True	True	True

Uguali

```
bools :: Int -> [[Bool]]
bools 0      = [[]]
bools n = map (False:) bn ++ map (True:) bn
      where
        bn = bools (n-1)
```

# Verifica di Tautologie (vii)

- Rimozione duplicati:

```
remdup :: Eq a => [a] -> [a]
remdup [] = []
remdup (testa:coda) = testa:remdup(select (/= testa) coda)
```

- Generazione di tutte le sostituzioni in una proposizione:

```
substitute :: Prop -> [Subst]
substitute p = map (zip vs) (bools (length vs))
  where
    vs = remdup (vars p)
```

$(A \wedge B) \Rightarrow A$

```
> substitute p2
[[('A',False), ('B',False)],
 [('A',False), ('B',True)],
 [('A',True), ('B',False)],
 [('A',True), ('B',True)]]
```

- Check di tautologia:

```
isTaut :: Prop -> Bool
isTaut p = and [eval s p | s <- substitute p]
```


```
> isTaut p1
False
```

```
> isTaut p2
True
```

```
> isTaut p3
False
```

```
> isTaut p4
True
```

# Overloading di Funzioni

- Funzioni che operano su più tipi:  **Polimorfe**  
**Overloaded**

```
length :: [a] -> Int
```

```
(>) :: Int -> Int -> Bool
```

```
(>) :: String -> String -> Bool
```

- Perché overloading? Se non ci fosse:

```
elemBool :: Bool -> [Bool] -> Bool
elemBool x [] = False
elemBool x (testa:coda) = (x ==Bool testa) || elemBool x coda
```

```
elemInt :: Int -> [Int] -> Bool
elemInt x [] = False
elemInt x (testa:coda) = (x ==Int testa) || elemInt x coda
```

- Possibili soluzioni:

- Funzione di uguaglianza come parametro di una funzione generale:

```
elemGen :: (a -> a -> Bool) -> a -> [a] -> Bool     $\Rightarrow$     elemGen (==Int) 3 lista
```

- Definizione di una funzione (polimorfa) che usa la funzione di uguaglianza overloaded:

```
elem :: a -> [a] -> Bool
```

Riuso: elem può essere usato su tutti i tipi con uguaglianza

Leggibilità: meglio unico simbolo ==

# Classi di Tipi

- **Classe di tipi** = { tipi sui quali è definita una certa funzione }

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

- **Istanza** = membro della classe
- Esempi di istanze di Eq:
  - Built-in: **Int**, **Float**, **Bool**, **Char**
  - Mediante costruttori tupla e lista applicati a tipi che sono istanze di Eq **(Int, Bool)**
- Esempio di non istanze di Eq:
  - Tipi di funzioni **Int -> Int**

# Esempi di Funzioni che Utilizzano l'Uguaglianza

```
allEqual :: Int -> Int -> Int -> Bool  
allEqual m n p = (m==n) && (n==p)
```



```
allEqual :: Eq a => a -> a -> a -> Bool  
allEqual m n p = (m==n) && (n==p)
```

- **Contesto** = parte che precede ' $\Rightarrow$ ' nel protocollo

*Se il tipo  $a$  appartiene alla classe `Eq` (cioè, se `==` è definita sul tipo  $a$ ) allora `allEqual` ha tipo  $a \rightarrow a \rightarrow a \rightarrow \text{Bool}$*

```
allEqual :: Char -> Char -> Char -> Bool  
allEqual :: (Int, Bool) -> (Int, Bool) -> (Int, Bool) -> Bool
```

```
next :: Int -> Int  
next n = n+1
```



```
> allEqual next next next  
Error: Int -> Int is not an instance of class "Eq"
```

# Esempi di Funzioni che Utilizzano l'Uguaglianza (ii)

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



```
Bool -> [Bool] -> Bool  
Int  -> [Int]  -> Bool
```

- Generalizzazione delle funzioni della biblioteca:

```
libriInPrestito :: [(Persona,Libro)] -> Persona -> [Libro]  
libriInPrestito database persona =  
  [ l | (p, l) <- database, p==persona ]
```



```
lookupFirst :: Eq a => [(a,b)] -> a -> [b]  
lookupFirst coppie elemento =  
  [ secondo | (primo, secondo) <- coppie, primo==elemento ]
```



# Dichiarazione di una Classe di Tipi

```
class Visible a where
  toString :: a -> String
  size     :: a -> Int
```

} signature

- `toString`: converte un oggetto del tipo `a` in `String`
- `size`: genera la dimensione dell'oggetto come valore intero

- In generale:

```
class nome vtipo where
  ... signature di funzione definita su vtipo
```

# Istanziamento di una Classe di Tipi

- Istanziamento della classe `Eq` mediante il tipo `Bool`:

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _      == _     = False
```

- Istanziamento della classe `Visible` mediante il tipo `Bool`:

```
instance Visible Bool where
  toString True  = "True"
  toString False = "False"
  size _         = 1
```

- Istanziamento della classe `Visible` mediante lista di tipo generico (ma visibile):

```
instance Visible a => Visible [a] where
  toString = concat . map toString
  size lista = sum (map size lista) + 1
```

# Definizioni di Default

- Effettiva definizione della classe `Eq` in Haskell:

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

$\Rightarrow$  Definizioni di **default** di  $\begin{cases} /= (da ==) \\ == (da /=) \end{cases}$

- Definizioni di default: **sovrascritte** da istanziazioni
- Per ogni istanziazione  $\rightarrow$  necessario definire almeno una delle due operazioni (l'altra è definita per default)
- Se definite entrambe le operazioni  $\rightarrow$  non usato il default
- Inibizione della sovrascrittura di un default: mediante definizione globale:

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

# Classi Derivate

- Definizione della classe di tipi Ord derivata dalla classe di tipi Eq:

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min           :: a -> a -> a
  compare            :: a -> a -> Ordering.....>{ EQ, LT, GT }
```

- Per istanziare la classe Ord mediante un tipo → necessario definire per il tipo le operazioni relative alle signature di Eq ed Ord (separatamente)
- Possiamo dire che “la classe Ord *eredita* le operazioni della classe Eq”
- Se fornita la definizione di <, possibile derivare le altre mediante i default:

```
x <= y  = (x < y || x == y)
x > y   = y < x
x >= y  = (y < x || x == y)
```

- Esempio di funzioni definite su tipi della classe Ord:

```
iSort :: Ord a => [a] -> [a]
ins  :: Ord a => a -> [a] -> [a]
```

# Vincoli Multipli

- Esempio: *Ordinamento di una lista e visualizzazione del risultato come stringa*

```
vSort :: (Ord a, Visible a) => [a] -> String
vSort = toString . iSort
```

- Esempio: *Visualizzazione come stringa del risultato di lookupFirst*

```
vLookupFirst :: (Eq a, Visible b) => [(a,b)] -> a -> String
vLookupFirst coppie elemento = toString (lookupFirst coppie elemento)
```

- Vincoli multipli nella istanziiazione:

```
instance (Eq a, Eq b) => Eq (a,b) where
  (x,y) == (z,w) = x==z && y==w
```

- Vincoli multipli nella definizione di classe (**ereditarietà multipla**):

```
class (Ord a, Visible a) => OrdVis a
```



```
vSort :: OrdVis a => [a] -> String
```

⇒ signature vuota: per essere in OrdVis, al tipo basta essere sia in Ord che in Visible