Funções como argumento
Padrões de computo

Padrões de computo

- Reuso é uma meta principal na indústria de software
- Haskell permite definir funções gerais
  - Polimorfismo paramétrico
  - Funções como argumentos
- Funções como argumento permitem escrever funções que representam padrões de computo
  - Transformar todos os elementos de uma lista
  - Combinar os elementos de uma lista usando um operador
- Chamamos estas funções de combinadores

Padrões de Computo sobre listas

Aplicando a todos

```
doubleAll :: [Int] -> [Int]
doubleAll [ ]  = [ ]
doubleAll (x : xs)    = 2*x : doubleAll xs

roundAll :: [Float] -> [Int]
roundAll [ ]           = [ ]
roundAll (f : fs)      = round f :  roundAll fs

capitalize :: String -> String
capitalize [ ]   = [ ]
capitalize (c : cs)   = toUpper c : capitalize cs
```

Padrão: transformar (mapear) todos os elementos de uma lista

Podemos definir uma única função passando a transformação como argumento

A função map

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

```
roundAll :: [Float] -> [Int]
roundAll fs = map round fs

capitalize :: String -> String
capitalize cs = map toUpper cs

doubleAll :: [Int] -> [Int]
doubleAll xs = map times2 xs
    where times2 x = 2*x
```

Escolhendo elementos – Filtrando

- Outro padrão comum:

  Escolher os elementos de uma lista que possuem uma dada propriedade

- Exemplos:
  - Escolher de uma lista de inteiros os números impares
  - Escolher dígitos de uma string

- Modelamos as propriedades com funções que retornam `Bool`

```
odd :: Int -> Bool
isDigit :: Int -> Bool
```

Um elemento x tem a propriedade f quando `f x == True`

A função filter

```
filter p [ ] = [ ]
filter p (x:xs)
    | p x  = x : filter p xs
    | otherwise = filter p xs
```

Qual é o tipo mais geral de filter?

```
impares :: [Int]  -> [Int]
impares ns = filter odd ns

digitos :: String -> String
digitos cs = filter isDigit cs

maiusculas :: String -> String
maiusculas cs = filter isUpper cs
```

Exercícios

- Resolva os problemas 1 e 4 da primeira prova sem usar nem compreensões e nem recursão

`zipWith`: Combinando `zip` com `map`

- A função `zip` permite agrupar duas listas numa só onde c/elemento é um par

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

- Duas visões de `zipWith`
  - generalização de `map` para funções binárias, ou
  - generalização de `zip` onde o “critério de agrupamento” é dado como argumento

- Exemplo:

```
zipWith (+) [1,2,3] [10,20,30,40] = [1+10,2+20,3+30]
```

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _   = [ ]
```

Qual é o tipo mais geral de `zipWith`?

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Exercícios

- Defina `length` usando `map` e `sum`
- Considere a função

```
addUp ns = filter greaterOne (map addOne ns)
    where greaterOne n = n>1
          addOne n = n+1
```

Como pode redefinir `addUp` de tal forma que seja feito o `filter` antes do `map`, como em

```
addUp ns = map fun1 (filter fun2 ns)
```

- Qual é o o efeito de

```
map addOne (map addOne ns)
```

Pode concluir algo geral sobre propriedades de `map f (map g ns)` onde `f` e `g` são funções arbitrárias?

- Defina funções que tomem uma lista, `ns`, e
  - retorne a lista consistindo dos quadrados dos inteiros em `ns`
  - retorne a soma dos quadrados dos itens em `ns`
  - Verifique se todos os itens da lista são maiores que zero

- Defina funções para
  - calcular o menor valor de uma função `f` aplicada de `0` até `n`
  - verificar se os valores de `f` aplicados de `0` até `n` são todos iguais
  - verificar se todos os valores de `f` aplicados de `0` até `n` são maiores que `0`
  - verificar se os valores de `f` aplicados de `0` até `n` estão em ordem crescente

- Defina uma função `p` que receba uma lista de strings `strs` e uma lista de caracteres `cs` e retorne um string contendo cada string em `str` acrescido, ao fim dele, o correspondente caractere da lista cs.
  Por exemplo

```
p ["Penso", "logo", "existo"] [',', ' ', '.'] =

                "Penso,logo existo."
```

- Estabeleça o tipo e defina uma função `twice` que aceita uma função e um valor e aplica esta função duas vezes. Por exemplo, a função `twice` aplicada as entradas `double` e a `7` produzirá `28` como resultado
- Defina o tipo e defina a função iter tal que
  `iter n f x = f (f (f … (f x)…))`
  onde f ocorre n vezes no lado direito da equação. Por exemplo, deveríamos ter que
  `iter 3 f x = f (f (f x)))`
- Usando `iter` e double defina uma função a qual para a entrada `n` retorna 2^n

Combinando todos os elementos de uma lista – *folding*

- Mais um padrão: aplicar uma função binária sobre todos os elementos de uma lista

```
sum [2,3,71] = 2 + 3 + 71

concat [ [1..5], [3..10], [15..16] ] =
     [1..5] ++ [3..10] ++ [15..16]

concat ["casa ", "de Tonho", "\n"] =
     "casa " ++  "de Tonho" ++  "\n"

and [True, True, False] = True && True && False

or [True, True, False] = True || True || False

maximum [2, 71, 40] = 2 `max` 71 `max` 40
```

foldr1

```
foldr1 g [e1,e2, ... ,ek]
    = e1 `g` (e2 `g` ... (... `g` ek) ... )
    = e1 `g` (foldr1 g [e2, ..., ek])
    = g e1 (foldr1 g [e2, ..., ek])
```

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 g [x]    = x
foldr1 g (x:xs)   = g x  (foldr1 g xs)
```

```
foldr1 (+) [3,98,1] ↝ 3 + 98 +1 ↝ 102
```

```
foldr1 (||) [False,True,False]  ↝ False || True || False
     ↝ True
```

```
foldr1 (++) ["casa ", "de Tonho", "\n"]
   ↝ "casa " ++ "de Tonho" ++ "\n"
   ↝ "casa de Tonho\n“
```

```
foldr1 max [8,10,4]
   ↝ 8 `max` (10 `max` 4)
   ↝ 10
```

foldr

- `foldr1` está indefinido para `[]`
- `foldr` recebe um argumento extra para devolver quando a lista for `[]`

```
foldr g s []     = s
foldr g s (x:xs) = g x (foldr g s xs)
```

Definições com `foldr` e `foldr1`

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

product :: Num a => [a] -> a
product xs = foldr (*) 1 xs

concat :: [[t]] -> [t]
concat xss = foldr (++) [] xss

and :: [Bool] -> Bool
and bs = foldr (&&) True bs

or :: [Bool] -> Bool
or bs = foldr (||) False bs

maximum :: Num a => [a] -> a
maximum ns = foldr1 max ns

fact :: Integral a => a -> a
fact n = foldr (*) 1 [1..n]
```

Combinando `and`/`or` com `map`

```
all :: (t -> Bool) -> [t] -> Bool
all p xs = and (map p xs)

any :: (t -> Bool) -> [t] -> Bool
any p xs = or (map p xs)
```

```
all odd [1,3,5,7] ↝ True

any even [1,3,5,7] ↝ False
```

- Qual é o tipo mais geral de `foldr`?

```
foldr :: (a -> b -> b) -> b -> [a] ->  b
foldr g s []     = s
foldr g s (x:xs) = g x (foldr g s xs)
```

`foldr` e recursão primitiva

- Definições com recursão primitiva sobre listas podem ser expressas sem recursão usando `foldr`
- `length` com `foldr`
  precisamos encontrar uma função g tal que, por exemplo

```
foldr g 0 [x, y, z] ↝
    x `g` (y `g` ( z `g` 0)) ↝ ... ↝ 3
```

```
length :: [t] -> Int
length xs = foldr g 0 xs
    where g :: t -> Int -> Int
          g _ n = n + 1
```

- `reverse` com `foldr`

```
foldr g [] [x, y, z] ↝
    x `g` (y `g` ( z `g` [])) ↝
    ... ↝ [z, y, x]
```

```
reverse :: [t] -> [t]
reverse xs = foldr snoc []  xs
   where snoc x xs =  xs ++ [x]
```

```
iSort :: Ord a => [a] -> [a]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)

iSort :: Ord a => a -> [a] -> [a]
ins x [] = [x]
ins x (y:ys)
   | x < y = x : y : ys
   | otherwise = y : ins x ys
```

- `iSort` com `foldr`

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

Sem recursão primitiva, mas com `foldr`

- Estratégias
  - Adivinhar a função de *folding*, ou
  - Definir usando recursão primitiva e depois transformar numa definição com `foldr`
- Vantagens
  - maior clareza, sabemos que há um *fold*
  - podemos usar propriedades do `foldr`

Exercícios

- Calcule a soma dos quadrados dos números naturais `1` até `n` usando `map` e `foldr`
- Defina uma função que dê a soma dos quadrados dos inteiros positivos de uma lista de inteiros
- Usando `foldr` defina as funções `unzip`, `last` e `init`
- O que calcula a seguinte função

```
misterio xs = foldr (++) [] (map sing xs)
    where sing x = [x]
```

- Defina uma função

```
filterFirst :: (a -> Bool) -> [a] -> [a]
```

tal que `filterFirst p xs` remova o primeiro elemento de `xs` que não satisfaz a propriedade `p`.

- Defina

```
filterLast :: (a -> Bool) -> [a] -> [a]
```

que remove a última ocorrência de um elemento de uma lista que não satisfaz a propriedade.

- Defina a função `switchMap` que aplica de forma alternada duas funções aos elementos de uma lista. Por exemplo

```
switchMap addOne addTen [1,2,3,4] ↝ [2,12,4,14]
```

- Defina funções

```
split :: [a] -> ([a], [a])
merge :: ([a], [a]) -> [a]
```

tal que `split` divide em duas listas, pegando alternadamente, enquanto `merge` intercala as duas listas. Por exemplo

```
split [1,2,3,4,5] ↝ ([1,3,5], [2,4])
merge ([1,3,5], [2,4]) ↝ [1,2,3,4,5]
```

- Formule propriedades que relacionem `split` e `merge` e teste-as com QuickCheck.

Generalizando divisão de listas

- Funções para quebrar listas já vistas
  - `take,drop :: Int -> [a] -> [a]`
  - `splitAt :: Int -> [a] -> ([a],[b])`
  - `takeWhile, dropWhile ::`
    `(a -> Bool) -> [a] -> [a]`

- `takeWhile` (`dropWhile`) captura um padrão comum:
  - Criar uma sublista com (eliminando) os primeiros elementos que satisfazem alguma propriedade

```
takeWhile men8 [2,5,7,8,10,11,12] ↝ [2,5,7]
takeWhile isDigit "12bcd34gh" ↝ "12"
takeWhile isLetter "bcd 34 gh" ↝ "bcd"
```

```
ins :: Int -> [Int] -> [Int]
ins x xs = takeWhile mx xs ++ [x] ++  dropWhile mx xs
    where mx y = y <= x
```

```
type Word = String

splitWords :: String -> [Word]
splitWords st = split (dropSpace st)
  where
    split [] = []
    split ss = takeWord ss :
                    split (dropSpace (dropWord ss))
```

```
takeWord :: String -> String
takeWord xs = takeWhile isLetter xs

dropWord :: String -> String
dropWord xs = dropWhile isLetter xs

dropSpace :: String -> String
dropSpace xs = dropWhile notLetter xs
```

Exercícios

- Defina as funções `takeWhile` e `dropWhile`
  Quais são seus tipos mais gerais?
- Usando `takeWhile` e `dropWhile` defina a função
  ```
  splitLines :: String -> [String]
  ```
  tal que `splitLines txt` retorna uma lista com as linhas em `txt`, assim por exemplo:

  ```
  splitLines "Este texto\ntêm\n\nquatro linhas" ↝
      ["Este texto","têm", "","quatro linhas"]
  ```