# 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
- 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
```