```
-- ************Funções de alta ordem***************

-- Funções são dados e tem um tipo (assim como Int, Bool,
-- [Int], [(a,b)], etc)

--   + podem ser combinadas usando operadores (assim como números
....)
--   + abstrações lambda (descrever uma função através de uma
expressão)
--   + podem ser argumentos e *resultados* de outras funções (alta
ordem)
--   + aplicação parcial

-- Definições ao nível de funções - point free definitions
--   + mais legíveis (??) (maior abstração)
--   + mais adequadas para verificação e transformação de programas

-- Composição de funções
------------------------
-- Definição

(f . g) x = f (g x)

--          ---------------------
--         | f.g                 |
--         |                     |
--       x|   ----  g x   ---    | f (g x)
--      --|->|  g | ---> | f |--|-->
--         |   ----        ---   |
--          ---------------------

-- Qual é o tipo mais geral?

(.) :: ...


-- O resultado da (.) é uma função!

--           ---
--      f-->| . |-->f.g
--      g-->|   |
--           ---

-- O gráfico ilustra uma forma de pensamento de alto nível.

-- Exemplo: somar as posições das letras maiúsculas

h str = sum ( segundos ( filtrarMaiusculas (zip str [0..]) )

filtrarMaiusculas ps = filter maius ps
   where maius (c, p) = isUpper c

segundos ps = map snd ps

-- Alternativamente, usando composição
```

```
h str = ( sum . segundos . filtrarMaiusculas ) (zip str [0..])

filtrarMaiusculas ps = filter maius ps
   where maius = isUpper . fst

-- Observe a última definição, maius é *point free*, definida
-- sem usar argumentos. Ela é *de alto nível*, estilo comum em
-- programação funcional.
-- Este estilo exige um pensamento diferente, de alto nível
-- Poderiamos ter definido assim:

...
  where maius p = (isUpper . fst) p

-- Isto, além de não ser elegante, nos faz pensar ao nível dos
-- elementos que são aplicados às funções.

-- O princípio da extensionalidade
-- para todo x, f x = g x sse f = g

-- Atenção a precedência da composição. Se escrevemos
--    f . g x
-- significa que queremos calcular
--    f . (g x)
-- porque a aplicação tem maior precedência sobre qualquer operador


-- Mais exemplos

dropSpace xs = dropWhile notLetter xs
  where notLetter c = not (isLetter c)

-- Uma versão sem definição auxiliar:
dropSpace xs = dropWhile (not . isLetter) xs

-- twice aceita uma função e a aplica duas vezes
twice f = f . f

-- Por exemplo,
(twice succ) 5 --> (succ . succ) 5 --> succ (succ 5) --> succ 6 --> 7

-- Qual é o tipo mais geral de twice?
twice :: ?


-- O operador de aplicação $
-----------------------------

-- A notação para a aplicação é por justaposição. Por ex:
-- f x
-- Mas podemos definir uma notação explícita:
-- f $ x

-- Para que isto?
--   + Uma alternativa para evitar parênteses
--     Por exemplo, ao invés de
--     sum ( segundos ( filtrarMaiusculas (zip str [0..]) )
```

```
--     podemos escrever
--     sum $ segundos $ filtrarMaiusculas $ zip str [0..]
--
--     Observe que $ é associativa à direita
--
--   + Podemos precisar usar a aplicação como uma função.
--     Por exemplo:
--     zipWith ($) [sum, produtc] [[1, 2], [3, 4]]


-- ***** Exercícios ******

--   + 11.3 do livro Haskell: the craft
--   + Qual é o tipo do operador de aplicação $?
--   + Considerando que id é a função identidade, explique qual é
--     o comportamento de cada expressão
--         + id $ f
--         + f $ id
--         + id ($)
--   + Defina a generalização de twice
--     iter :: Int -> (a -> a) -> (a -> a)
--     tal que
--     iter n f
--     é a composição de f com f, n vezes.
--   + Usando iter, defina a função
--     pot2 :: Int -> Int
--     tal que  pot2 n = 2^n


-- Expressões Lambda
----------------------

-- funções sem nome
-- funções escritas "on the fly"
-- Expressões lambda são expressões ;)

-- Exemplo

addOneAll xs = map addOne xs
  where addOne n = n + 1

addOneAll xs = map (\n -> n+1) xs

-- Aqui
-- \n -> n+1
-- é uma expressão lambda. Representa a uma função com argumento n
-- que retorna n+1

-- O símbolo \ é lido como "lambda"

-- Expressões lambda vêm do cálculo lambda, uma teoria de funções
-- inventada por Haskell B. Curry

-- Mais exemplos

--
doubleAll xs = map times2 xs
```

```
    where times2 x = 2*x

doubleAll xs = map (\x -> 2*x) xs
--

-- uma expressão lambda pode ter vários argumentos:

length xs = foldr g 0 xs
    where g :: t -> Int -> Int
          g m n = n + 1

length xs = foldr (\m n -> n+1) 0 xs

-- podemos usar padrões como argumentos de uma exp. lambda:

length xs = foldr (\_ n -> n+1) 0 xs

segundos ps = map (\(_,y) -> y) ps

--
mapFuns fs x = map applyToX fs
  where applyToX f = f x

mapFuns fs x = map (\f -> f x) fs

-- A última definição é mais direta e clara.

-- Expressões lambda podem ser úteis numa definição de função
-- que retorna uma função
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)


-- Uma definição, como
addOne :: Int -> Int
addOne n = n+1

-- Pode, alternativamente, ser dada com expressões lambda, assim
addOne :: Int -> Int
addOne = \n -> n+1

-- Como exemplo mais geral, as seguintes definições são equivalentes
f x y z = resultado

f = \x y z -> resultado


-- ***** Exercícios ******

--   + 11.7, 11.8, 11.9 e 11.10 do livro Haskell: the craft 2ed
--   + Usando ranges, map e expressões lambda, defina replicate
--     replicate :: int -> a -> [a]
--     tal que replicate n x devolva uma lista formada por n x's.
--   + Usando replicate e foldr, defina iter
```

```
-- Aplicação parcial
--------------------

multiply x y = x*y

-- Graficamente

--         ----------
--     -->|          |
--        | multiply |-->
--     -->|          |
--         ----------

-- mult têm dois argumentos, no entanto Haskell permite aplicação
-- parcial, por exemplo

multiply 2

--          ----------
--     2-->|          |
--         | multiply |-->
--      -->|          |
--          ----------

multiply 2 é uma função que ao receber n, retorna 2*n

-- Exemplos:

doubleAll xs = map (multiply 2) xs

double = multiply 2

triple = multiply 3

--
subtract x y = x - y

minus = subtract 0

-- Só é possível aplicação parcial no primeiro argumento. Não
-- podemos aplicar parcialmente só no segundo argumento.

-- De fato, se a função tem vários argumentos, podemos aplicar
-- parcialmente só com o primeiro argumento, ou só com o primeiro
-- e o segundo, ou ...
-- Em outras palavras, para aplicar ao segundo temos
-- que ter aplicado ao primeiro ...

maxThree :: Int -> Int -> Int -> Int
maxThree m n p = max m (max n p)

-- são expressões válidas
maxThree
maxThree 5
maxThree 5 12
maxThree 5 12 8
```

```
-- Consegue explicar qual é o comportamento de cada expressão?

-- Aplicação parcial a operadores (seções)
-----------------------------------------

(+ 2) -- função que soma 2
(2 +) -- idem
(> 2) -- função que verifica se um número é maior que 2
(2 >) -- função que verifica se um número é menor que 2
(3:) -- função que acrescenta 3 na cabeça da lista
(++ "\n") -- função que adiciona uma nova linha ao final de um string

-- O que representa a seguinte expressão?

filter (> 0) . map (+ 1)

--Exemplos
doubleAll :: Num a => [a] -> [a]
doubleAll = map (* 2)

even :: Integral a => a -> Bool
even = (== 0) . (`mod` 2)

odd :: Integral a => a -> Bool
odd = not . even

-- Funções curried
------------------

-- Quantos argumentos tem a seguinte função?
maior :: Int -> Int -> Int
maior x y
   | x <= y    = y
   | otherwise = x

-- Só um!
-- A seta -> associa à direita, logo a definição de tipo tem que ser
-- entendida como

maior :: Int -> (Int -> Int)

-- Assim, maior pega um argumento Int e retorna uma função de tipo
-- Int -> Int

-- Por outro lado, a aplicação (por justaposição) é associativa à
-- esquerda, assim

maior 3 4

-- deve ser entendida como

(maior 3) 4

-- De fato, rigorosamente falando, podemos dizer que Haskell só tem
-- funções com um argumento. Mas evitamos dizer isto, por
conveniência.
-- Assim, falamos que maior aceita dois argumentos do tipo Int.
```

```
-- Particularmente a expressão

maior 3

-- representa a aplicação parcial de maior a um argumento, neste caso
3,
-- retornando uma função que ao receber um n retornará o maior entre
-- 3 e n.

-- funções uncurried
maiorUC :: (Int, Int) -> Int
maiorUC (x, y)
   | x <= y    = y
   | otherwise = x

-- curried ou uncurried? Prefira curried
--    + notação leve, aplicação por justaposição
--    + permite aplicação parcial

-- transformando curried para uncurried e viceversa
-- Gráficos
curry ::  ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)
-- ou
curry f = \x y -> f (x,y)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (x, y) = f x y
-- ou
uncurry f  = \(x, y) -> f x y


-- ***** Exercícios ******

-- Redefina mapFuns usando aplicação parcial sobre $

-- Usando combinadores estudados(map, foldr, filter, etc), aplicação
parcial,
-- composição e expressões lambda, escreva definições para as funções
do exemplo do
-- banco de dados de uma biblioteca (Seção 5.7 do livro)

type Person = String
type Book = String

type Database = [ (Person, Book) ]

exampleBase :: Database
exampleBase = [ ("Alice", "Tintin"), ("Anna", "Little women"),
                ("Alice", "Asterix"), ("Rory", "Tintin") ]

books :: Database -> Person -> [Book]
borrowers :: Database -> Book -> [Person]
borrowed :: Database -> Book -> Bool
numBorrowed :: Database -> Person -> Int
makeLoan :: Database -> Person -> Book -> Database
returnLoan :: Database -> Person -> Book -> Database
```

```
-- Reimplemente as funções do exercício anterior, também usando
combinadores,
-- aplicação parcial, composição e expressões lambda, usando esta nova
-- definição do tipo Database

type Database = [ (Person, [Book]) ]
```