Programação Funcional

Luiz Alberto do Carmo Viana

11 de junho de 2019

Resumo

Notas de aula para a disciplina de Programação Funcional, ministrada no curso de Ciência da Computação, no campus da UFC em Crateús.

Sumário

1 Introdução

Programar computadores é uma atividade que pode ser bastante prazeroza. Testar programas de computador, a fim de garantir que funcionem adequadamente, já não é tão agradável (ao menos para mim).

A crescente complexidade dos ambientes computacionais traz consigo linguagens de programação cada vez mais sofisticadas, dotadas de recursos necessários para lidar com os avanços técnicos dos computadores. Atualmente, por exemplo, é bastante comum nos depararmos com computadores dotados de múltiplos processadores ou *cores*. Para tirar proveito desse recurso, é preciso escrever programas que executem instruções em paralelo, com o auxílio de linguagens que forneçam os meios para lidar com os vários *cores* do processador.

Se considerarmos que *debugar* um programa simples já é desafiador, o que podemos dizer sobre corrigir um programa que executa, simultaneamente, diferentes instruções em alguns processadores? Uma vez que pudéssemos demonstrar a corretude do nosso programa, não haveria a preocupação de testá-lo, e assim esse problema desapareceria completamente.

O paradigma de Programação Funcional propõe o desenvolvimento de linguagens de programação que se aproximem o máximo possível da linguagem matemática. Essa escolha de *design* facilita a demonstração da corretude de um programa funcional, que não precisa ser testado. Dessa forma, o paradigma em estudo ganha bastante adequação para o desenvolvimento de programas que lidam com ambientes computacionais complexos.

2 Ferramentas

Nestas notas de aula, usaremos a linguagem de programação Haskell, na dsitribuição Linux de sua preferência. Admitimos, naturalmente, o uso do compilador GHC (Glasgow Haskell Compiler), e esperamos que sua versão seja a 8.4.4. Você pode conferir a versão instalada com o comando `ghci` (falaremos mais a respeito dele no futuro), que deve produzir uma saída como esta.

```
$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/
Prelude>
```

Existem diversas formas de se obter o GHC. Você pode tentar instalá-lo pelo gerenciador de pacotes da sua distribuição, ou tentar baixá-lo pelo site oficial. Recomendamos o uso do ghcup, que costuma instalar a versão mais recente do compilador.

3 Programando em Haskell

Aqui, reimplementamos as funções básicas de Haskell com o propósito de entender seu funcionamento. Isso também nos permite ganhar familiaridade com a linguagem e sua sintaxe, além de nos expor, aos poucos, às suas características.

Vamos desenvolver nossa reimplementação utilizando uma divisão em módulos mais simplificada que a estrutura modular oficial. Usaremos os seguintes arquivos:

- `Bool.hs`: aqui se encontram as funções relacionadas a operações com valores booleanos;
- `BSTree.hs`: contém as definições da estrutura de árvore binária de busca;
- `Either.hs`: implementa valores disjuntivos e funções para seu manuseio;
- `Functions.hs`: apresenta as operações elementares para o reuso de funções;
- `Graph.hs`: define algumas funções para manipulação de grafos;
- `Heap.hs`: implementação de uma fila de prioridade mínima;

- `InfinityTree.hs`: árvore binária infinita, isto é, sem folhas;
- `List.hs`: elenca diversas funções para o tratamento de listas;
- `Maybe.hs`: define um tipo para representação de falha, assim como facilidades para o seu manuseio;
- `Memo.hs`: traz funções que implementam memoização;
- `SeqTree.hs`: contém as definições de uma estrutura arbórea simples para a representação de sequências.

3.1 Trabalhando com listas

Iniciamos nossos estudos da linguagem Haskell tratando de um tipo composto que é clássico para o paradigma funcional: a *lista*. A lista é caracterizada por ser uma estrutura linear que contém elementos de um único tipo. Uma lista é uma estrutura que pode ou ser *vazia* ou ter uma *cabeça* (composta de um único elemento) e uma *cauda* (uma outra lista, contendo o mesmo tipo de elemento).

Vamos criar um arquivo `List.hs` com o seguinte conteúdo inicial. Este indica que iremos trabalhar com algumas definições de Prelude, bem como iremos fazer uso de conceitos definidos em outros módulos.

```
module List where

import Prelude (
        Eq, Num, Int, Char, Bool(True, False),
        (+), (-), (*), (<=), (==), (/=), error,
        Functor(fmap), (<$>), Applicative(pure, (<*>)),
        seq)

import Control.Applicative (Alternative(empty, (<|>)))

import Bool
import Functions
import Maybe
```

Iniciamos nossas implementações com uma operação muito básica: determinar o comprimento de uma lista. Dada sua natureza recursiva, podemos aproveitar a estrutura de uma lista para calcular recursivamente seu comprimento.

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

Vamos entender o código acima por partes. A primeira linha determina a assinatura da função `length`, que recebe uma lista de elementos do tipo `a` (pode ser um tipo qualquer), e retorna um elemento do tipo `Int`, isto é, um inteiro. Em Haskell, uma lista vazia é representada por `[]`. Intuitivamente, sabemos que uma lista vazia tem 0 elementos, e dizemos isso na segunda linha. Esse é o nosso caso base. Quando tomamos uma lista não vazia, sabemos que ela tem cabeça e cauda. Em Haskell, representamos uma lista não vazia como `(x:xs)`, que tem cabeça `x` e cauda `xs`. Essa lista tem um elemento `x`, e esse "um"precisa ser somado à quantidade de elementos da cauda `xs` (computada recursivamente, já que `xs` é uma lista). Agora vamos testar nosso código.

Antes de aprendermos a utilizar o compilador (comando `ghc`), vamos nos habituar a testar nosso código no interpretador (comando `ghci`). Vamos para o diretório onde está `List.hs` e executamos o seguinte.

```
$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/
Prelude> :l List
[1 of 1] Compiling List
Ok, one module loaded.
*List> length [1,2,4]
3
*List> :q
Leaving GHCi.
```

Como podemos ver, carregamos o arquivo no interpretador utilizando `:l List` (note que não precisamos escrever a extensão do arquivo). Quando passamos uma lista com três elementos, `length` responde de acordo. Para sair do interpretador, utilizamos `:q`.

Agora que entendemos o básico de como utilizar o interpretador, vamos nos concentrar em desenvolver outras funções. A função `null`, por exemplo, determina se uma lista é vazia ou não.

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

Recomendamos carregar `List.hs` no interpretador e fazer alguns testes. Não deve ser difícil entender que `null` recebe uma lista contendo elementos de um tipo qualquer e retorna um valor booleano (verdadeiro para listas vazias).

Vamos agora criar duas funções para acessar as componentes do tipo lista: cabeça e cauda.

```
head :: [a] -> a
head [] = error "head: empty list"
head (x:_) = x
```

Temos duas coisas novas no trecho de código acima. Primeiro, como não podemos retornar a cabeça de uma lista vazia, temos de lançar uma exceção. Isso é feito utilizando a função `error`, que recebe uma string e lança uma exceção (a assinatura de error é muito interessante). A segunda coisa ocorre na expressão `(x:_)`: o *underline* serve para ignorar o nome de um argumento de função (nesse caso, a cauda de uma lista decomposta). Como não precisamos mencionar a cauda da lista para retornar sua cabeça, consideramos boa prática não nomeá-la. Agora definimos a função `tail`, de forma análoga.

```
tail :: [a] -> [a]
tail [] = error "tail: empty list"
tail (_:xs) = xs
```

Podemos entender o par de funções `head` e `tail` como duas funções que decompõem uma lista separando seu primeiro elemento. Intuitivamente, podemos conceber as funções `last` e `init`, que decompõem uma lista separando seu último elemento.

```
last :: [a] -> a
last [] = error "last: empty list"
last (x:[]) = x
last (_:xs) = last xs
```

Observe que a implementação de `last` tem três casos, apesar de uma lista ter apenas duas definições: vazia ou com cabeça e cauda. É preciso tratar o caso de uma lista ter cabeça e uma cauda vazia (uma lista com apenas um elemento) por conta da definição recursiva. Observe que, se não houvesse o referido caso, a definição recursiva sempre levaria ao lançamento de uma exceção. Definimos agora `init`.

```
init :: [a] -> [a]
init [] = error "init:␣empty␣list"
init (_:[]) = []
init (x:xs) = x : init xs
```

A ocorrência de três casos em `init` é justificada da mesma forma que em `last`. Temos algo novo no caso recursivo: além de `(:)` ser usado para decompor listas, também pode ser usado para a construção de listas. Em `init`, usamos `(:)` para construir uma lista, a ser retornada, cuja cabeça é `x` e cuja cauda é o resultado de `init xs`.

Uma operação usual em listas (e outros tipos compostos lineares) é a *concatenação*, que consiste em tomar duas listas e retornar uma lista maior, resultado de justapor a primeira seguida da segunda. Vamos implementar essa função como um operador infixo, isto é, como uma função "binária"(essas aspas serão explicadas posteriormente) que ocorre entre seus argumentos.

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

A primeira coisa que observamos é como o operador `(++)` é definido na assinatura. Quando definimos a assinatura de operadores, colocamos seu nome entre parênteses. Quando definimos os casos de um operador, escrevemos seu nome entre seus dois argumentos. Quanto à definição de `(++)`, seu primeiro caso é bem simples. No último caso, definimos que o primeiro elemento da concatenação de duas listas não vazias é justamente o primeiro elemento da primeira lista, e os demais elementos da concatenação são definidos recursivamente.

Para facilitar o uso de `(++)` em certos contextos, definimos também a função `concat`, que recebe uma lista de listas e retorna uma lista simples, resultado da sucessiva concatenação das sublistas recebidas.

```
concat :: [[a]] -> [a]
concat [] = []
concat (xs:xss) = xs ++ concat xss
```

Chamamos a atenção para o estilo de nomenclatura utilizado. Listas simples costumam ter nomes com um único 's' no final, enquanto listas de listas são nomeadas com "ss"ao final.

Agora definimos como computar o inverso de uma lista. Nossa primeira tentativa poderia ser algo como o seguinte.

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

Essa definição, embora correta, não é a mais eficiente. Perceba que o operador `(++)` tem um custo linear em relação ao número de elementos de seu primeiro argumento. Se, para cada elemento de seu argumento, `reverse` usa `(++)`, usaríamos `(++)` um número linear de vezes. Como `(++)` tem custo linear, essa definição de `reverse` tem custo quadrático (em relação ao número de elementos de seu argumento).

Podemos definir `reverse` com custo linear (percorrendo seu argumento uma única vez) se fizermos uso de uma função auxiliar, da seguinte forma.

```
reverse :: [a] -> [a]
reverse xs = f xs [] where
        f [] acc = acc
        f (x:xs) acc = f xs (x:acc)
```

Vamos dar um tempo com nosso arquivo `List.hs` e criar um outro, `Bool.hs`. Nosso novo arquivo terá o seguinte conteúdo inicial.

```
module Bool where

import Prelude(Bool(True, False), Applicative(pure))

import Control.Applicative (Alternative(empty))

import Functions
```

Infelizmente não poderemos desenvolver nosso próprio tipo booleano, pois diversas funções em Haskell que são implementadas em baixo nível referenciam o tipo booleano usado pelo compilador. Por isso, importamos o tipo `Bool` e seus valores a partir de `Prelude`.

Em `Bool.hs`, vamos desenvolver duas funções bastante similares a operadores lógicos bem conhecidos.

```
and :: [Bool] -> Bool
and [] = True
and (True:xs) = and xs
and (False:_) = False
```

A função `and` toma uma lista de valores booleanos e retorna um booleano. Se a lista contém apenas valores `True`, então ela retorna `True`. Do contrário, ela retorna `False`. Perceba que, além de decompor a lista em cabeça e cauda, criamos casos de acordo com os possíveis valores da cabeça. Essas decomposições de tipos em partes (lista em cabeça e cauda), bem como a criação de casos de acordo com os possíveis valores de uma certa parte, são chamadas de *pattern matching*. A seguir, definimos a função `or` de forma similar.

```
or :: [Bool] -> Bool
or [] = False
or (True:_) = True
or (False:xs) = or xs
```

Exercício 1 *Implemente os operadores* `(&&)` *e* `(||)`*, ambos com assinatura* `Bool -> Bool -> Bool`*. Esses operadores funcionam de forma análoga a* `and` *e* `or`*, respectivamente, mas tomam necessariamente dois valores booleanos como argumentos.*

Voltando a editar `List.hs`, criamos duas operações muito usuais em programação funcional, que consistem em tomar ou descartar prefixos de una lista. As funções `take` e `drop` fazem precisamente isso.

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n - 1) xs
```

Dado um inteiro `n`, `take` retorna um prefixo de comprimento `n` da lista de entrada (caso essa seja vazia, apenas a lista vazia pode ser retornada como prefixo). Observe, mais uma vez, o uso de *pattern matching* por valores: o primeiro caso verifica se o inteiro `n` tem valor 0. O último caso diz que, se `n` não é nulo (esperamos que seja positivo), e a lista não é vazia, então sua cabeça certamente faz parte do prefixo a ser retornado. Os `n - 1` elementos restantes do prefixo são tomados, de forma recursiva, da cauda da lista dada como entrada.

Antes de escrever a função `drop`, vamos criar um condicional, o que nos permitirá um outro estilo de escrita. A função `cond` é bem simples de entender, e ficará escrita em `Bool.hs`.

```
cond :: Bool -> a -> a -> a
cond True x _ = x
cond False _ y = y
```

```
drop :: Int -> [a] -> [a]
drop _ [] = []
drop n xs = cond (n <= 0) xs (drop (n - 1) (tail xs))
```

Note que, nesse estilo de escrita, não usamos *pattern matching* para determinar a nulidade de `n`, nem para atribuir nomes às partes de uma lista não vazia. Usamos `cond` para tratar os valores de `n` (perceba que isso lida inclusive com o caso de `n` ter valor negativo) e `tail` para acessar a cauda da lista de entrada (graças ao primeiro caso, sabemos que `tail` nunca receberá uma lista vazia).

A função `splitAt` combina os resultados de `take` e `drop`. Para isso, ela faz uso de uma *tupla*.

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs = (take n xs, drop n xs)
```

Mais uma vez, essa definição simples não é a mais eficiente. Como é percebido, chamar `take` e `drop` com os mesmos argumentos implica em "passear"pelas mesmas posições da lista duas vezes. Implementando `splitAt` independentemente de `take` e `drop`, conseguimos resolver esse problema.

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt 0 xs = ([], xs)
splitAt _ [] = ([], [])
splitAt n (x:xs) = (x:ps, qs) where
        (ps, qs) = splitAt (n - 1) xs
```

Dessa vez, fizemos uso de `where` para receber o resultado da chamada recursiva, que é um par. Assim, podemos usar esse resultado para construir o retorno de `splitAt` de forma adequada.

Vamos criar nossa primeira função que exigirá uma restrição em sua assinatura. A função `isPrefixOf` recebe duas listas, e determina se a primeira é prefixo da segunda.

```
isPrefixOf :: Eq a => [a] -> [a] -> Bool
isPrefixOf [] _ = True
isPrefixOf _ [] = False
isPrefixOf (x:xs) (y:ys) = cond (x == y) (isPrefixOf xs ys) False
```

Dessa vez, a função não recebe listas contendo um tipo qualquer. O tipo dos elementos das listas deve implementar as funções da classe `Eq`. Em Haskell, uma *classe* é apenas um grupo de funções. Um exemplo de classe é `Eq`, que contém os operadores `(==)` e `(/=)`. Assim, se um tipo é *instância* da classe `Eq`, podemos usar o operador `(==)` sobre seus valores. Em `isPrefixOf`, isso é necessário para determinar se os elementos da primeira lista são iguais aos primeiros elementos da segunda.

Exercício 2 *Implemente a função* `isInfixOf :: Eq a => [a] -> [a] -> Bool`*, que decide se a primeira lista é sublista contínua da segunda.*

Exercício 3 *Implemente a função* `isSubsequenceOf :: Eq a => [a] -> [a] -> Bool`*, que decide se a primeira lista é uma subsequência da segunda, não necessariamente contínua.*

Por vezes, queremos percorrer duas listas simultaneamente. Uma forma de fazer isso é transformar as duas listas numa lista de pares, onde as primeiras e segundas posições vêm, respectivamente, da primeira e da segunda lista. A função `zip` faz justamente isso.

```
zip :: [a] -> [b] -> [(a, b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

Vamos apresentar agora as primeiras funções que tomam funções como argumento. Por ser uma linguagem funcional, o sistema de tipos de Haskell é bastante robusto. Isso se mostra no fato de que não apenas tipos simples, como `Int` e `Bool`, são recebidos e retornados por funções, mas também assinaturas de funções podem ser argumentos ou o valor de retorno de certas funções (funções de *alta ordem*). Dizemos que, em Haskell, as funções são *habitantes de primeira classe*.

Lembra que a função `(++)` foi dita "binária"? Apesar de parecer que ela recebe dois argumentos, na verdade ela recebe um só. O seu retorno, no entanto, não é uma lista, e sim uma função. Em verdade, todas as funções em Haskell recebem apenas um argumento, e são portanto *unárias*. Particularmente, `(++)` recebe uma lista `xs` e retorna uma outra função. Essa outra função, por sua vez, recebe uma lista `ys` e retorna a lista que é a concatenação de `xs` e `ys`. Devemos nos acostumar a ler as assinaturas na forma `(++) :: [a] -> ([a] -> [a])`. Dessa forma, uma função é determinada pelo primeiro `->` de sua assinatura: antes dele, está o tipo de seu único argumento; após ele, o tipo de seu retormo.

Para receber uma função como argumento, escrevemos sua assinatura entre parênteses. Temos, por exemplo, a função `takeWhile`.

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:xs) = cond (p x) (x : takeWhile p xs) []
```

Note que `p` é um argumento do tipo `a -> Bool`. Dizemos que `p` é um *predicado*, isto é, uma função que recebe um valor de um certo tipo e retorna um valor booleano. Por exemplo, tomando a assinatura `(<=) :: Ord a => a -> a -> Bool`, podemos criar o predicado `(0<=) :: (Ord a, Num a) => a -> Bool`, cujo argumento é um valor de um tipo numérico ordenável.

A função `takeWhile` toma um predicado e retorna uma função, que por sua vez toma uma lista e retorna seu maior prefixo cujos elementos satisfazem `p`. Seu par é a função `dropWhile`, que descarta o maior prefixo cujos elementos satisfazem um certo predicado.

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p xs = cond (p (head xs)) (dropWhile p (tail xs)) xs
```

Para esse contexto, o análogo da função `splitAt` se chama `span`. A função `span` combina os resultados de `takeWhile` e `dropWhile`, e já devemos ter uma ideia de como implementá-la de forma eficiente.

```
span :: (a -> Bool) -> [a] -> ([a], [a])
span _ [] = ([], [])
span p l@(x:xs) = cond (p x) (x:ps, qs) ([], l) where
        (ps, qs) = span p xs
```

Na definição de `span`, temos mais uma novidade sintática. Usamos `@` quando queremos nomear tanto um valor de um tipo composto quanto suas partes. Nesse caso, usamos `l@(x:xs)` para indicar que a lista passada como argumento se chama `l`, e também que sua cabeça e cauda se chamam, respectivamente, `x` e `xs`.

Em vez de usar `@`, poderíamos simplesmente escrever `x:xs` onde ocorre `l` na definição do segundo caso. No entanto, isso construiria uma nova lista igual a `l`. Por mais que tivessem a cauda compartilhada, ainda assim haveria algum desperdício de memória, e portanto o uso de `@` é preferível.

Exercício 4 *Implemente a função* `insert :: Ord a => a -> [a] -> [a]` *que, dado um elemento de um tipo ordenável e uma lista ordenada, insere o elemento na lista, preservando a ordenação.*

Exercício 5 *Implemente a função* `insertionSort :: Ord a => [a] -> [a]`*, que toma uma lista de elementos ordenáveis e retorna uma versão ordenada dessa lista.*

Vamos criar um novo arquivo, `Functions.hs`, que terá o seguinte conteúdo inicial.

```
module Functions where

import Prelude (Applicative(pure, (*>)), Monad((>>=)), seq)
```

Nele, definimos algo crucial para a expressividade e reuso em programação funcional: a composição de funções. Em Haskell, como era de se esperar, a composição de funções também é uma função.

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

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

Definimos `comp` para determinar a composição como uma função. Em seguida, definimos a composição como o operador `(.)`, que toma duas funções e retorna uma função (o resultado de passar `g` para a função retornada por `comp f`).

Exercício 6 *Implemente a função* `replicate :: Int -> a -> [a]` *que, dado um inteiro não-negativo* `n` *e um elemento* `x`*, cria uma lista de* `n` *elementos, todos iguais a* `x`*. Note que* `replicate` *pode ser definida como uma composição de* `take` *e* `repeat`*, essa última função proposta na Subseção 4.2.*

Uma função bastante comum de ser composta com predicados é a função `not`. Sua descrição não exige explicações, e é escrita em `Bool.hs`.

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

Voltando a editar `List.hs`, podemos agora definir a função `break`, cujo comportamento se assemelha ao de `span`. No entanto, `break` nega o predicado antes de particionar a lista recebida.

```
break :: (a -> Bool) -> [a] -> ([a], [a])
break p xs = span (not . p) xs
```

Além de trabalhar apenas tomando prefixos de listas, é perfeitamente plausível que haja uma função que filtre uma lista com base em um predicado. Essa função costuma ser chamada de `filter`.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = cond (p x) (x:xs') xs' where
        xs' = filter p xs
```

Apesar de todas as funções em Haskell serem unárias, conseguimos "simular" funções com mais argumentos retornando outras funções unárias para receber os argumentos restantes. Essa, no entanto, não é a única forma de fazer isso. Como tuplas são tipos compostos, podemos criar funções unárias que recebem uma tupla como seu único argumento, e assim poderíamos tratar cada posição da tupla recebida como um argumento independente. Quando usamos a primeira forma de "simular" mais de um argumento, estamos usando *curried functions* (como Curry é um sobrenome, uma tradução desse termo seria provavelmente tosca). Quando utilizamos tuplas para o mesmo fim, criamos *uncurried functions*.

Curried functions costumam tornar a escrita em Haskell mais expressiva, permitindo um uso mais flexível das funções. No entanto, algumas situações podem tornar desejável que certas funções não sejam aplicadas parcialmente, e daí as *uncurried functions* ganham justificativa. Nesse sentido, definimos a função `uncurry`, que recebe uma *curried function* "binária" e retorna a *uncurried function* equivalente. Ela deve ser escrita em `Functions.hs`.

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

Ironicamente, a função `uncurry` deve ser aplicada parcialmente para que retorne a *uncurried function* correspondente ao seu argumento. Assim, `uncurry (+)` seria equivalente a `(+)`, mas não permitiria aplicações parciais (como `(1+)`). Para tornar *uncurried* funções com mais de dois argumentos, seria preciso escrever a versão de `uncurry` apropriada.

3.2 Abstraindo o conceito de *loop*

Vimos anteriormente a função `filter`, que filtra os elementos de uma lista conforme um predicado. Em uma linguagem imperativa, teríamos usado um *loop* (como um *for*) e um condicional para obter um resultado similar. Em Haskell, as diversas formas de se tratar o conteúdo de uma lista devem ser suficientes para fazer tudo que é possível com o auxílio de *loops*.

A função `map` transforma o conteúdo de uma lista de acordo com uma função. Ela pode ser implementada como segue, e sua definição será escrita em `Functions.hs`.

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

Exercício 7 *Implemente a função* `inits :: [a] -> [[a]]`*, que retorna todos os prefixos da lista dada como entrada.*

Exercício 8 *Implemente a função* `tails :: [a] -> [[a]]`*, que retorna todos os sufixos da lista dada como entrada.*

Caso `f` tivesse mais argumentos, a função `map` não serviria. A função `zipWith` segue a mesma lógica de `map`, mas é usada para funções com "dois" argumentos. Também será escrita em `Functions.hs`.

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

Podemos inclusive utilizar `zipWith` para implementar `zip` como uma aplicação parcial usando `(,) :: a -> b -> (a, b)`. Voltamos ao nosso arquivo `List.hs` e reescrevemos `zip` como segue.

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

Até agora, nossas funções tomam listas como argumentos e retornam listas, sejam estas sublistas ou transformações da lista de entrada. Por vezes, é desejável fazer uma computação que envolva todos os elementos da lista. Um somatório, por exemplo, tomaria uma lista de valores de um tipo numérico e retornaria um único valor desse tipo. Em Haskell, o ato de transformar todos os elementos de uma lista `[a]` em um único elemento de um tipo `b` é chamado *folding*, e costuma ser realizado com o auxílio de uma função.

Podemos começar a "dobrar"os elementos de uma lista a partir de qualquer uma de suas duas extremidades. A função `foldl` começa pela esquerda, emquamto `foldr` começa pela direita. Manteremos a definição de ambas em `List.hs`.

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

Vamos explicar o que `foldl` está fazendo. No caso base, não há elementos da lista para "dobrar"e `foldl` retorna o acumulador, o argumento responsável por guardar o resultado das "dobras"já feitas. No segundo caso, a lista não está vazia, e `foldl` "dobra"a cabeça da lista usando a função `f` para atualizar o acumulador, e o resultado é passado como acumulador para a chamada recursiva.

Uma coisa importante de se notar é que a definição recursiva de `foldl` é apenas uma chamada de `foldl`. Funções com essa propriedade são ditas *recursivas por cauda*, e são eficientes no uso de memória, devido a otimizações que os compiladores modernos são capazes de fazer. Agora definimos `foldr`.

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

Não deve ser difícil perceber que a definição de `foldr` retorna um valor de um tipo como é proposto na assinatura. O mais intrigante é entender como `foldl` e `foldr` "dobram"os elementos na ordem que cada uma propõe. Isso está relacionado à forma como a função `f` é utilizada. Em `foldl`, `f` é usada para calcular um argumento de uma chamada de `foldl`, e como a definição recursiva é essencialmente essa chamada, `x` é o primeiro elemento da lista a ser computado por `f`. Já em `foldr`, a definição recursiva tem seu resultado dado por uma chamada de `f`, cujo segundo argumento é dado por `foldr`, dessa vez aplicada à cauda. Assim, se o retorno de `f` for totalmente dependente de seu segundo argumento, `x` será o último elemento da lista a ser computado por `f`, uma vez que a chamada recursiva de `foldr` ocorrerá antes, e irá tratar de todos os elementos da cauda `xs`.

Como exemplos do uso de `foldl`, podemos descrever as seguintes funções em `List.hs`.

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

```
prod :: Num a => [a] -> a
prod = foldl (*) 1
```

Para ilustrar o uso de `foldr`, podemos reescrever `filter` da seguinte forma.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = foldr f [] xs where
        f x ys = cond (p x) (x:ys) ys
```

Note como a definição de `f` se assemelha à definição anterior de `filter`. Perceba que, pela definição de `foldr`, `ys` pode ser vista como a cauda já filtrada de acordo com `p`. Além disso, como o retorno de `f` (uma lista) não depende totalmente de `ys` (caso `p x == True`, a cabeça não é elemento de `ys`), `foldr` não cumpre o seu papel. Ela sempre o cumpriria em uma linguagem de *avaliação estrita*, isto é, numa linguagem em que a computação dos argumentos precede a chamada da função.

Aparentemente falha por conta das características de Haskell, `foldr` faz algo que `foldl` não pdoe fazer: o retorno parcial de seu resultado. Isso é notável quanto ao retorno de listas: `foldl`, por ser recursiva por cauda, precisa computar a lista inteira antes de poder retornar seu primeiro elemento, enquanto `foldr` disponibiliza os elementos tão logo estejam computados. O impacto disso fica evidente com o auxílio desta definição de `takeWhile`, que deixaremos escrita em `List.hs`.

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p xs = foldr f [] xs where
        f x ys = cond (p x) (x:ys) []
```

Perceba que há uma única diferença dessa definição de `takeWhile` para a definição de `filter`. Suponha que apenas os três primeiros elementos de uma lista satisfaçam um certo predicado. `foldr` verificaria apenas os quatro primeiros elementos, não importa o tamanho da lista, enquanto que operações semelhantes com `foldl` precisariam percorrer a lista toda, pelo fato de `foldl` ser recursiva por cauda.

Exercício 9 *Implemente as funções* `and`, `or` *e* `map` *como chamadas de* `foldr`.

Exercício 10 *Um outro conceito de* loop *é a iteração, que consiste em, dada uma função e um elemento inicial, aplicar a função a esse elemento, e repetir esse processo com o resultado dessa aplicação. Implemente a função* `iterate :: (a -> a) -> a -> [a]` *que, dados uma função* `f` *e um elemento* `x`*, cria uma lista infinita com os resultados de sucessivas aplicações de* `f` *em* `x`*, isto é,* `iterate f x == [x, f x, f (f x), ...]`*. Perceba que* `iterate` *não é muito útil por si só.*

3.3 Usando tipos para o tratamento de erros

Vimos anteriormente que `error` é capaz de indicar erros de execução, interrompendo o fluxo usual do programa com o lançamento de uma exceção. No entanto, `error` não nos dá a oportunidade de tratar esses erros, muito menos fornece uma forma simples de indicar que uma função pdoe falhar. Ao lermos a assinatura de `head`, por exemplo, nada nos diz que existe a chance de ela lançar uma exceção. Vamos criar um tipo justamente com o propósito de representar uma computação que pode falhar.

Criamos o arquivo `Maybe.hs` com o conteúdo inicial a seguir.

```
module Maybe where

import Prelude (Bool(True, False),
                Functor(fmap),
                Applicative((<*>)),
                Monad((>>=)))

import Bool
import Functions

data Maybe a = Nothing | Just a
```

Aqui, temos o primeiro uso de `data`, que serve para a definição de um tipo. Criamos o tipo `Maybe` que, dado um tipo `a` como argumento, representa um conjunto de valores em dois casos: `Maybe a` pode estar vazio (`Nothing`) ou pode conter um único valor do tipo `a` (`Just a`).

A possibilidade de `Maybe a` conter nenhum valor do tipo `a` nos permite representar uma falha com `Nothing`, pois se uma computação não é bem sucedida, ela produz nenhum valor.

Como exemplo de seu uso, podemos importar o módulo `Maybe` em `List.hs` e definir versões bem comportadas de `head` e `tail`.

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x

safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_:xs) = Just xs
```

Exercício 11 *Implemente a função* `uncons :: [a] -> Maybe (a, [a])` *que, dada uma lista, retorna uma tupla composta, respectivamente, por sua cabeça e sua cauda. Note que a função* `uncons` *evita lançar uma exceção caso receba uma lista vazia.*

Vemos que `safeHead` e `safeTail` não lançam exceções, mas representam seus casos falhos retornando `Nothing`. É sempre preferível usar o sistema de tipos para tratar os erros, em vez de apenas lançar uma exceção. Como a validação dos tipos de um programa é feita em tempo de compilação, o compilador consegue nos dizer se todas as possibildiades de erro estão tratadas antes de executarmos nosso programa.

Vamos agora desenvolver um aparato para facilitar o tratamento de valores do tipo `Maybe a`. Os seguintes predicados são úteis para facilitar futuras definições.

```
isNothing :: Maybe a -> Bool
isNothing Nothing = True
isNothing _ = False

isJust :: Maybe a -> Bool
isJust = not . isNothing
```

Agora definimos a função `maybe`.

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe y f Nothing = y
maybe _ f (Just x) = f x
```

Pela assinatura de `maybe`, podemos entender seu objetivo. A função `maybe` recebe um valor do tipo `b` e uma função `f`, além de um valor do tipo `Maybe a`. Com a ajuda de `y`, `maybe` consegue tratar uma possível falha em seu "terceiro"argumento: caso esse seja `Nothing`, `y` é retornado como um valor padrão; caso contrário, `x` é aplicado a `f` para produzir o valor de retorno.

Perceba ainda que `maybe` é feita para ser aplicada parcialmente. Uma vez aplicados `y` e `f`, `maybe` retorna uma versão de `f` que tolera falhas em seu argumento.

```
fromMaybe :: a -> Maybe a -> a
fromMaybe y Nothing = y
fromMaybe _ (Just x) = x
```

A função `fromMaybe` também serve para definir um valor padrão em caso de falha, mas de forma mais simples. Por não receber uma função, é útil para definir valores padrão quando um valor do tipo `Maybe a` deveria ser usado em uma expressão.

```
catMaybes :: [Maybe a] -> [a]
catMaybes [] = []
catMaybes (Nothing:xs) = catMaybes xs
catMaybes ((Just x):xs) = x : catMaybes xs
```

A função `catMaybes` remove os valores `Nothing` de uma lista de elementos do tipo `Maybe a`, enquanto também extrai os valores agregados aos `Just`.

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe f = catMaybes . map f
```

Por fim, a função `mapMaybe` transforma uma lista de elementos do tipo `a` usando uma função que pode falhar, e em seguida remove as transformações que falharam. Repare como a divisão entre esses dois passos é evidenciada com `(.)`, e mais uma vez, a composição de funções promove o reuso de código.

3.4 Representando valores disjuntivos

Nesta subseção, implementamos o tipo `Either a b`, cujos valores representam uma disjunção entre valores dos tipos `a` e `b`. Escrevemos suas definições em `Either.hs`, que tem o seguinte conteúdo inicial.

```
module Either where

import Prelude (Bool(True, False))

import Bool
import Functions

data Either a b = Left a | Right b
```

Por vezes, percebe-se o uso de `Either` apenas como um `Maybe` "informativo", já que `Either [Char] b` pode representar um erro com `Left [Char]`, permitindo uma string que o descreva. Esse, no entanto, é um uso muito limitado de `Either`, que facilita a descrição de alguns algoritmos.

Desenvolvemos facilidades para a manipulação de `Either`. As funções que seguem são escritas também em `Either.hs`.

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x) = f x
either _ g (Right y) = g y
```

A função `either` toma duas funções para "resolver" a disjunção provocada por `Either`. É irresistível traçar uma analogia com construções como *switch* e *case*, típicas de linguagens procedurais. No entanto, o *switching* não ocorre entre valores de um certo tipo, mas sim entre dois tipos.

```
lefts :: [Either a b] -> [a]
lefts [] = []
lefts (Left x : xs) = x : lefts xs
lefts (Right _ : xs) = lefts xs

rights :: [Either a b] -> [b]
rights [] = []
rights (Left _ : xs) = rights xs
rights (Right x : xs) = x : rights xs
```

A função `lefts` (`rights`) filtra os elementos de tipo `a` (`b`) e os retorna em uma lista. Usa-se *pattern matching* para isso, e as definições são imediatas.

```
isLeft :: Either a b -> Bool
isLeft (Left _) = True
isLeft _ = False

isRight :: Either a b -> Bool
isRight = not . isLeft
```

As definições de `isLeft` e `isRight` são elementares. Partimos para as próximas definições.

```
fromLeft :: a -> Either a b -> a
fromLeft _ (Left x) = x
fromLeft x (Right _) = x

fromRight :: b -> Either a b -> b
fromRight x (Left _) = x
fromRight _ (Right x) = x
```

As funções `fromLeft` e `fromRight` retornam valores padrão caso não recebam o caso correspondente, do contrário retornam o valor nele contido.

Exercício 12 *Implemente a função* `partitionEithers :: [Either a b] -> ([a], [b])`.

3.5 Árvores binárias de busca

Em programação funcional, definimos o conceito das operações sem levar em conta como seria sua implementação na arquitetura que utilizamos. Isso é suficiente para caracterizar as linguagens desse paradigma como de alto nível, o que lhes confere muita expressividade, já que a escrita de código não menciona detalhes alheios à lógica das definições. Há, no entanto, um preço a se pagar por essa característica.

Por se distanciar dos detalhes da arquitetura e abordar a programação por um viés matemático, as linguagens funcionais não têm contato com estruturas de dados cujo conceito é dependente da arquitetura. É o caso, por exemplo, dos *arrays*, que são dotados de acesso em tempo constante a qualquer uma de suas posições. Uma lista, por outro lado, garante apenas acesso em tempo linear a seus elementos, o que é uma desvantagem séria. Para garantir a eficiência dos programas funcionais, é preciso construir uma estrutura de dados que alivie esse problema.

Vamos criar um arquivo `BSTree.hs`, que conterá as definições necessárias para o uso de árvores binárias de busca. Apesar de não terem acesso em tempo constante a seus elementos, mas apenas em tempo logarítmico (quando balanceadas), elas proporcionam um tempo de acesso muito superior ao das listas. É verdade que esse tempo de acesso ainda não é tão bom quanto o de um *array*, mas é eficiente o bastante (e o obtemos sem nos incomodar com detalhes da arquitetura).

```
module BSTree where

import Prelude (Ord, Ordering(LT, EQ, GT), compare,
                Eq, (==),
                Bool(True, False),
                Functor(fmap)
               )

import Bool
import Functions ((.))
import Maybe

data BSTree k v = Empty
                | Branch (k, v) (BSTree k v) (BSTree k v)
```

Isso define uma árvore binária de busca, com chaves do tipo `k` e valores do tipo `v`. Temos dois casos para nossa árvore: ela pode ser vazia (`Empty`) ou pode ser ramificada (`Branch`), contendo um par de chave e valor (`(k, v)`) e duas sub-árvores, sendo assim definida recursivamente.

Estamos definindo um *tipo parametrizado*, isto é, um tipo que precisa de outros tipos para estar completo. Apenas `BSTree` não é um tipo bem definido. Para completar a definição desse tipo, é preciso informar que tipo servirá de chave (`k`) e que tipo servirá de valor (`v`). Como exemplo, `BSTree Int [Char]` é uma árvore binária de busca chaveada por inteiros e valorada por strings. A lista, inclusive, é outro tipo parametrizado, já que precisa determinar o tipo de seus elementos para estar bem definida. O tipo `Maybe a` também ilustra o uso de um parâmetro para a definição de um tipo.

Vamos escrever uma função que retorna uma árvore vazia. Faremos com que este seja o único modo de criar uma `BSTree` fora desse módulo.

```
empty :: BSTree k v
empty = Empty
```

É apenas isso. Basta retornar o valor do primeiro caso. Agora, definimos como deve ocorrer uma inserção em nossa árvore.

```
insert :: Ord k => BSTree k v -> (k, v) -> BSTree k v
insert Empty (key', val') = Branch (key', val') empty empty
insert bst@(Branch (key, val) ltree rtree) (key', val') =
        decide (compare key' key) where
                decide LT = Branch (key, val) ltree' rtree
                decide EQ = bst
                decide GT = Branch (key, val) ltree rtree'
                ltree' = insert ltree (key', val')
                rtree' = insert rtree (key', val')
```

No seu primeiro caso, `insert` transforma uma árvore vazia em uma árvore ramificada com as duas sub-árvores vazias. No segundo caso, é preciso tratar a inserção de forma a preservar as invariantes de uma árvore binária de busca: não pode haver chaves duplicadas e todas as chaves contidas na sub-árvore esquerda (direita) devem ser menores (maiores) que a chave presente na raiz. A função auxiliar `decide` compara a chave a ser inserida (`key'`) com a chave da raiz (`key`) por meio de `compare`, que por sua vez retorna um valor do tipo `Ordering`: `LT` (*less than*), `EQ` (*equals*) ou `GT` (*greater than*). Caso `decide` receba `EQ`, a nova chave é duplicada e retorna-se `bst`, o que não modifica a árvore. Caso `decide` receba `LT` (`GT`), a operação de inserção é feita recursivamente na sub-árvore esquerda (direita), conforme a definição de `ltree'` (`rtree'`).

Note como a preguiça de Haskell afeta a definição de `insert`. Apesar de ambas `ltree'` e `rtree'` estarem definidas, no máximo uma delas será computada. Isso ocorre porque, em todos os três casos, no máximo uma delas é utilizada para definir o retorno.

Além disso, é preciso chamar a atenção para a natureza *imutável* dos valores de tipos em Haskell. Uma função nunca altera o valor associado aos identificadores de seus argumentos. Dessa forma, `insert` não altera a árvore recebida, mas retorna uma nova árvore com o par de chave e valor inserido.

A propriedade de que valores associados a identificadores são imutáveis não quer dizer que uma implementação de Haskell seja necessariamente um desastre em termos de consumo de memória. Como os valores são imutáveis, não há prejuízo em tipos compostos (como listas e árvores) partilharem alguns de seus campos, como listas partilharem caudas ou árvores partilharem sub-árvores, já que essas partes em comum nunca serão alteradas. A imutabilidade, inclusive, permite que um compilador pratique otimizações muito agressivas, mesmo em programas com múltiplas *threads*.

Definimos agora uma forma de consultar um valor do tipo `BSTree k v`. A função `lookup` toma uma árvore e uma chave, e verifica se a árvore contém a chave.

```
lookup :: Ord k => BSTree k v -> k -> Maybe v
lookup Empty _ = Nothing
lookup (Branch (key, val) ltree rtree) key' =
        decide (compare key' key) where
                decide LT = lookup ltree key'
                decide EQ = Just val
                decide GT = lookup rtree key'
```

Como não há garantias de que a árvore contém a chave, e portanto um valor associado a ela, `lookup` deve retornar `Maybe v`. A lógica de busca também baseia-se nas invariantes de uma árvore binária de busca, o que torna a estrutura dessa definição similar à da definição de `insert`. Podemos definir rapidamente uma função que verifica se uma árvore contém uma certa chave.

```
contains :: Ord k => BSTree k v -> k -> Bool
contains bst = isJust . lookup bst
```

Além de inserir valores associados a chaves em nossa árvore, é interessante que possamos atualizar o valor de uma chave. Isso nos permitiria utilizar mais facilmente a `BSTree` em certas situações.

```
update :: Ord k => BSTree k v -> k -> (v -> v) -> BSTree k v
update Empty _ _ = empty
update (Branch (key, val) ltree rtree) key' f =
        decide (compare key' key) where
                decide LT = Branch (key, val) ltree' rtree
                decide EQ = Branch (key, f val) ltree rtree
                decide GT = Branch (key, val) ltree rtree'
                ltree' = update ltree key' f
                rtree' = update rtree key' f
```

Vamos escrever funções para determinar as chaves máximas e mínimas de uma árvore, junto com seu valor.

```
maxKey :: BSTree k v -> Maybe (k, v)
maxKey Empty = Nothing
maxKey (Branch keyval _ Empty) = Just keyval
maxKey (Branch _ _ rtree) = maxKey rtree

minKey :: BSTree k v -> Maybe (k, v)
minKey Empty = Nothing
minKey (Branch keyval Empty _) = Just keyval
minKey (Branch _ ltree _) = minKey ltree
```

Note como não precisamos restringir `k` a ser instância da classe `Ord` nessas duas funções. Isso é possível porque não fazemos comparações. A busca pela chave máxima (mínima) utiliza apenas a estrutura da árvore e suas invariantes: caso a sub-árvore direita (esquerda) não seja vazia, a chave máxima (mínima) tem de estar lá. Ambas as funções sequer "abrem"o conteúdo do par `keyval`, usando esse identificador para a tupla inteira em vez de associar identificadores individuais às componentes do par.

Vamos agora definir como é feita a remoção de uma chave da `BSTree`.

```
remove :: Ord k => BSTree k v -> k -> BSTree k v
remove Empty _ = empty
remove bst@(Branch (key, val) ltree rtree) key' =
        decide (compare key' key) where
                decide LT = Branch (key, val) ltree' rtree
                decide GT = Branch (key, val) ltree rtree'
                decide EQ = removeRoot bst
                ltree' = remove ltree key'
                rtree' = remove rtree key'
                removeRoot Empty = Empty
                removeRoot (Branch _ lt Empty) = lt
                removeRoot (Branch _ Empty rt) = rt
                removeRoot (Branch _ lt rt) =
                        Branch (keyMinR, valMinR) lt rt' where
                                Just (keyMinR, valMinR) = minKey rt
                                rt' = remove rt keyMinR
```

Essa função não é tão complexa quanto parece. Basicamente, `remove` busca pela chave a ser removida e, se ela é encontrada, a sub-árvore da qual ela é raiz é passada para a função `removeRoot`. A função `removeRoot` faz exatamente o que seu nome diz, e para isso há quatro casos (referentes ao número de sub-árvores não vazias). O primeiro caso nunca será usado (por quê?), e apenas deixa completa a definição de `removeRoot`. Os dois próximos casos são imediatos, e temos o último caso, onde a árvore em questão tem suas duas sub-árvores não vazias: tomamos o par de menor chave na sub-árvore direita para substituir o par da raiz, e em seguida o removemos da sub-árvore direita, para não haver chaves duplicadas. Dessa forma, a antiga chave da raiz não estará presente na árvore a ser retornada.

Há duas observações interessantes. A primeira é que `keyMinR` nunca usará o último caso de `removeRoot`. Para perceber isso mais claramente, basta ver a definição de `minKey`. A segunda observação é sobre como o retorno de `minKey` é tomado. Temos a liberdade de admitir que seu retorno será um `Just (k, v)` pelo fato de que `removeRoot` nunca passa uma árvore vazia para `minKey`.

Assim como definimos `map` (em `Functions.hs`) para aplicar uma função em uma lista, podemos declarar uma função `map` dentro de `BSTree.hs` com o mesmo propósito. Mais adiante, veremos como evitar conflitos entre funções homônimas.

```
map :: (v1 -> v2) -> BSTree k v1 -> BSTree k v2
map _ Empty = empty
map f (Branch (key, val) ltree rtree) =
        Branch (key, f val) ltree' rtree' where
                ltree' = map f ltree
                rtree' = map f rtree
```

É interessante que a lista e `BSTree` possam ambas ter seu conteúdo transformado. Inclusive, isso é esperado de tipos que representam estruturas de dados. Mais adiante, veremos que essa propriedade ocorre com frequência em diversos tipos.

Se quisermos comparar duas árvores por igualdade, devemos fazer `BSTree k v` ser uma instância da classe `Eq`. Isso pode ser feito da seguinte forma.

```
instance (Eq k, Eq v) => Eq (BSTree k v) where
        Empty == Empty = True
        Empty == _     = False
        _     == Empty = False
        (Branch (key, val) ltree rtree) == (Branch (key', val') ltree' rtree') =
                and [key == key', val == val', ltree == ltree', rtree == rtree']
```

Nessa definição, exigimos que `k` e `v` sejam comparáveis por igualdade para determinar o comparador de igualdade de `BSTree k v`. Sua definição não deve ser difícil de entender. O que deve chamar a atenção aqui é o fato de que essa não é a única definição plausível de igualdade

entre árvores. É possível pensar, por exemplo, que duas árvores são estruturalmente iguais se têm o mesmo conjunto de chaves, dispostas da mesma forma. Nesses casos, onde há mais de uma definição (semanticamente) plausível para tornar um tipo instância de uma classe, existe um recurso da linguagem para "encapsular" um tipo como outro, de forma que ele possa ser instância de uma classe mais de uma vez. Isso será abordado futuramente.

Por fim, para concluir o módulo `BSTree.hs`, definimos sua lista de exportação, ou seja, as definições do módulo que serão visíveis para quem o importa. Essa é uma prática usual para esconder a definição de um tipo e funções auxiliares. Modificamos o começo de `BSTree.hs` da seguinte forma.

```
module BSTree (BSTree, empty,
               insert, update, remove, map,
               lookup, contains) where
```

Módulos como `BSTree.hs`, que contém a definição de um tipo, costumam ter funções homônimas às definidas para outros tipos. Para evitar conflitos entre nomes de funções, é sempre bom importar esses módulos como `qualified`. Assim, `import qualified BSTree as BST` deixa vísiveis as definições exportadas de `BSTree.hs` a partir do símbolo `BST`. Quando formos utilizar a função `empty`, por exemplo, ela será referenciada como `BST.empty`. Perceba que essa prefixação é suficiente para resolver conflitos entre funções homônimas definidas em módulos distintos.

Exercício 13 *Escreva a função* `breadth :: BSTree k v -> [(k, v)]` *que, dada uma árvore binária de busca, retorna seus pares de chave e valor em largura. Isso quer dizer que os elementos são retornados, a cada nível, da esquerda para a direita, começando pelo nível da raiz. Use a concatenação de listas para simular o comportamento de uma fila.*

Exercício 14 *Escreva a função* `leaves :: BSTree k v -> Int`*, que retorna o número de folhas da árvore binária de busca dada como entrada.*

Exercício 15 *Escreva as funções* `inOrder`*,* `preOrder` *e* `postOrder`*, todas com assinatura* `BSTree k v -> [(k, v)]`*, que retornam uma lista com os pares de chave e valor contidos na árvore de entrada. A ordem dos pares nas listas é dada de acordo com o passeio sugerido pelo nome da função.*

3.6 Representando sequências de forma eficiente

Como sabemos, listas são a estrutura de dados padrão para se trabalhar em Haskell e cumprem o papel de representar sequências de elementos. Sua definição simples, no entanto, não nos permite operar com tais sequências de forma eficiente. Se quisermos acessar o `k`-ésimo elemento de uma lista, precisamos fazer um número de operações proporcional a `k`. Isso fica evidente na definição do operador `(!!) :: [a] -> Int -> a`, que se encontra em `List.hs`.

```
(!!) :: [a] -> Int -> a
[] !! _ = error "(!!): empty list"
(x:_) !! 0 = x
(_:xs) !! k = xs !! (k - 1)
```

Exercício 16 *Implemente* `(!?) :: [a] -> Int -> Maybe a`*, que deve ser uma versão segura de* `(!!)`*.*

Sempre que é necessário acessar elementos de uma sequência em ordem arbitrária, o tempo linear de acesso torna as listas representações indesejáveis. Assim, devemos propor uma representação que permita um acesso mais eficiente. Embora não possamos garantir acesso em tempo constante, como nas linguagens imperativas, podemos garantir acesso em tempo logarítmico.

Uma ideia seria usar uma `BSTree` onde as chaves representam as posições da sequência. No entanto, o nosso tipo `BSTree` não garante um balanceamento em sua definição. Pior ainda, se inserirmos a sequência em ordem crescente (ou decrescente) de posições, teremos uma representação linear da sequência (tal como é a lista, mas gastando ainda mais memória). Devemos lembrar que uma árvore binária de busca sem operações de balanceamento é razoavelmente eficiente apenas quando sua construção se dá de maneira arbitrária.

No entanto, como temos uma ideia clara do que queremos representar de forma arbórea, é perfeitamente possível que consigamos garantir um balanceamento baseado na ordem em que as chaves são inseridas. Por exemplo, podemos admitir a inserção das chaves apenas em ordem crescente e conceber uma representação em árvore que seja naturalmente balanceada.

Vamos definir nossa estrutura de dados no arquivo `SeqTree.hs`, que terá o seguinte conteúdo inicial.

```
module SeqTree where

import Prelude (Int, (+), (-), (^),
                Ord, (<), (<=), max)

import Bool
import Maybe

data SeqTree a = Empty
               | Leaf a
               | Branch Int Int (SeqTree a) (SeqTree a)
```

A primeira coisa notável em nossa definição é que não usamos um tipo para representar nossas chaves, uma vez que sequências são todas indexadas por inteiros. Outra observação está no fato de que apenas as folhas contém elementos do tipo `a`, enquanto os nós internos da árvore contém dois valores inteiros. Isso quer dizer que o conteúdo de todas as posições de nossa sequência estará armazenado em folhas, e os nós internos terão informações para garantir o balanceamento da árvore.

De fato, é preciso explicar o que representam os valores inteiros em `Branch`. O primeiro valor representa a altura do nó, e nos auxilia a determinar quantos elementos cabem em sua sub-árvore. Se um nó tem altura `h`, e `SeqTree a` representa uma árvore binária, então sua sub-árvore pode ter no máximo 2^h folhas, e portanto pode conter no máximo 2^h elementos do tipo `a`. Já o segundo valor determina a quantidade de folhas que a sub-árvore daquele nó contém. Dessa forma, podemos determinar se uma sub-árvore está cheia comparando esses dois valores, o que vai permitir que nossa operação de inserção garanta algum balanceamento para a árvore.

Antes de prosseguir, vamos definir alguns *getters* para `SeqTree`, que serão bastante úteis para estabelecer algumas conveções.

```
height :: SeqTree a -> Int
height Empty = 0
height (Leaf _) = 0
height (Branch h l ltree rtree) = h
```

Usamos `height` para convencionar que a altura de árvores vazias e de árvores com um único nó é 0. À primeira vista parece estranho que folhas não tenham altura 1, mas perceba: se permitirnos que folhas tenham altura 1, estaremos admitindo que elas comportam $2^1 = 2$ elementos, o que não faz sentido. Por fim, o terceiro caso estabelece o primeiro valor inteiro de `Branch` como sua altura.

```
leaves :: SeqTree a -> Int
leaves Empty = 0
leaves (Leaf _) = 1
leaves (Branch h l ltree rtree) = l
```

A função `leaves` retorna o número de folhas de uma `SeqTree`. `Empty` não tem folhas e `Leaf` tem exatamente uma folha. O terceiro caso estabelece que o segundo valor inteiro de `Branch` representa seu número de folhas.

Para construir valores de `SeqTree`, podemos usar seus três construtores (`Empty`, `Leaf` e `Branch`). Vamos, no entanto, evitar o uso do construtor `Branch` e preferir o uso da função `makeBranch`, com a finalidade de encapsular alguns cálculos.

```
makeBranch :: SeqTree a -> SeqTree a -> SeqTree a
makeBranch tx ty = Branch h' l' tx ty where
        h' = max (height tx) (height ty) + 1
        l' = leaves tx + leaves ty
```

Utilizar `makeBranch` é mais simples que utilizar `Branch`, pois altura e número de folhas já estão determinados. A altura da árvore criada é a altura da maior de suas sub-árvores, mais um. Já o número de folhas da árvore é a soma do número de folhas de suas sub-árvores.

Vamos criar agora *getters* para tomar as sub-árvores de uma `SeqTree`. Suas definições não exigem muitas explicações, mas observamos que há a convenção de que árvores vazias ou com apenas um nó têm ambas as suas sub-árvores vazias.

```
leftTree :: SeqTree a -> SeqTree a
leftTree Empty = Empty
leftTree (Leaf _) = Empty
leftTree (Branch h l ltree rtree) = ltree

rightTree :: SeqTree a -> SeqTree a
rightTree Empty = Empty
rightTree (Leaf _) = Empty
rightTree (Branch h l ltree rtree) = rtree
```

Agora precisamos definir a inserção de forma a garantir um balanceamento para `SeqTree`. Vamos fazer isso permitindo a inserção de elementos apenas após o último elemento da sequência representada por uma `SeqTree`.

Antes de definirmos a operação de inserção em uma `SeqTree`, faz-se necessário criar um tipo para representar a comparação de um elemento de um tipo ordenável em relação a dois outros elementos do mesmo tipo.

```
data Ordering = LL | LG | GL | GG

compare :: Ord a => a -> a -> a -> Ordering
compare x p q = cond (x < p)
                     (cond (x < q) LL LG)
                     (cond (x < q) GL GG)
```

Basicamente, criamos esse `Ordering` personalizado e sua função `compare` com o objetivo estético de expressar essa lógica de comparação fora de `insert`. A função `insert` é definida a seguir.

```
insert :: SeqTree a -> a -> SeqTree a
insert Empty x = Leaf x
insert ltree@(Leaf _) x = makeBranch ltree (Leaf x)
insert branch x = decide (compare (leaves branch) halfCapacity capacity) where
        decide LL = makeBranch lbranch' rbranch
        decide GL = makeBranch lbranch rbranch'
        decide GG = makeBranch branch (Leaf x)
        lbranch = leftTree branch
        rbranch = rightTree branch
        lbranch' = insert lbranch x
        rbranch' = insert rbranch x
        halfCapacity = 2^(height branch - 1)
        capacity = 2^(height branch)
```

Os dois primeiros casos de `insert` lidam com `Empty` e `Leaf`: inserir um elemento em uma árvore vazia resulta em uma única folha; inserir um elemento em uma folha resulta em uma árvore de altura 1 com duas folhas. Em seu último caso, `insert` tem de se preocupar com o número de folhas de `branch`: se `branch` tem menos folhas que o máximo de folhas que pode haver em `lbranch` (`halfCapacity`), então a inserção é feita em `lbranch`; caso `branch` não esteja com seu número máximo de folhas, a inserção é feita em `rbranch`; caso `branch` tenha atingido seu número máximo de folhas, `branch` torna-se a sub-árvore esquerda de uma nova `SeqTree`, cuja sub-árvore direita consiste de uma folha.

Agora, definimos a função `lookup`, cujo objetivo é retornar o valor associado a uma posição da sequência. Note que consideramos nossas sequências indexadas a partir de 1, e não de 0.

```
lookup :: SeqTree a -> Int -> Maybe a
lookup Empty _ = Nothing
lookup (Leaf x) 1 = Just x
lookup (Leaf _) _ = Nothing
lookup branch n = cond  (n <= halfCapacity)
                        (lookup lbranch n)
                        (lookup rbranch n') where
        halfCapacity = 2^(height branch - 1)
        lbranch = leftTree branch
        rbranch = rightTree branch
        n' = n - halfCapacity
```

Primeiro, temos os três casos elementares. No último caso, `lookup` verifica se o índice da posição buscada (`n`) é limitado pelo número máximo de folhas que uma sub-árvore de `branch` pode ter. Caso seja, e como `insert` garante que `branch` tem elementos em `rbranch` apenas se `lbranch` estiver cheia, então a busca pelo valor da posição `n` continua em `lbranch`. Do contrário, o valor associado a `n` não pode estar em uma das folhas de `lbranch`, e a busca deve continuar em `rbranch`, mas com uma modificação: `rbranch` representa uma subsequência que começa após os `halfCapacity` primeiros elementos da sequência representada por `branch`; dessa forma, o `n`-ésimo elemento da sequência de `branch` deverá estar na posição `n - halfCapacity` da subsequência de `rbranch`.

```
update :: SeqTree a -> Int -> (a -> a) -> SeqTree a
update Empty _ _ = Empty
update (Leaf x) 1 f = Leaf (f x)
update leaf@(Leaf _) _ _ = leaf
update branch n f = cond (n <= halfCapacity)
                         (makeBranch lbranch' rbranch)
                         (makeBranch lbranch rbranch') where
        halfCapacity = 2^(height branch - 1)
        lbranch = leftTree branch
        rbranch = rightTree branch
        lbranch' = update lbranch n f
        rbranch' = update rbranch n' f
        n' = n - halfCapacity
```

Acima, temos a função `update`, que atualiza o conteúdo de uma posição da sequência. A decisão sobre qual sub-árvore irá sofrer a atualização é a mesma de `lookup`. No entanto, em vez de retornar o elemento da posição `n`, `update` reconstrói as sub-árvores necessárias para que `f x` torne-se o valor da posição `n` da sequência representada por `branch`.

```
remove :: SeqTree a -> SeqTree a
remove Empty = Empty
remove (Leaf _) = Empty
remove (Branch _ _ ltree Empty) = remove ltree
remove (Branch _ _ ltree rtree) = makeBranch ltree (remove rtree)
```

Temos agora nossa função `remove`, que sempre remove o último elemento da sequência. Perceba que `remove` apenas deleta a folha que estiver mais à direita, o que a torna bem simples.

Além disso, perceba que o terceiro caso de `remove` é responsável por reduzir a estrutura de uma `SeqTree`, eliminando sub-árvores vazias.

Aproveitamos também para definir como transformar uma sequência de elementos. A função `map` explora a estrutura de `SeqTree`, alterando o conteúdo de suas folhas.

```
map :: (a -> b) -> SeqTree a -> SeqTree b
map _ Empty = Empty
map f (Leaf x) = Leaf (f x)
map f (Branch _ _ ltree rtree) = makeBranch ltree' rtree' where
        ltree' = map f ltree
        rtree' = map f rtree
```

Antes de preparar a lista de exportação de nosso módulo, fornecemos uma maneira de criar uma `SeqTree` vazia.

```
empty :: SeqTree a
empty = Empty
```

Por fim, definimos nossa lista de exportação, alterando o começo de `SeqTree.hs` da seguinte forma.

```
module SeqTree (SeqTree, empty, insert, update, lookup, remove, map) where
```

Assim, temos concluído as definições necessárias para `SeqTree`. Futuramente, utilizaremos essa estrutura para resolver alguns problemas.

3.7 Compilando um programa

Até o presente momento, escrevemos alguns módulos com definições úteis e os testamos utilizando um interpretador, o `ghci`. Nesta breve subseção, mostramos quais os passos necessários para se compilar um programa em Haskell.

Dado um arquivo `Module.hs`, utilizamos o seguinte comando para compilá-lo.

```
ghc Module -W
```

Esse comando compila `Module.hs`, produzindo um tipo de saída conforme seu conteúdo. Aqui, `-W` habilita os *warnings* do `ghc` que, em suas últimas versões, tornaram-se opcionais. Certamente, é sempre bom usar `-W`.

Se `Module.hs` tem em sua primeira linha algo como `module Module where`, então o comando acima produz um arquivo `Module.o`. Como exemplo, temos a compilação de `List.hs`.

```
$ ghc List -W
[1 of 4] Compiling Bool             ( Bool.hs, Bool.o )
[2 of 4] Compiling Functions        ( Functions.hs, Functions.o )
[3 of 4] Compiling Maybe            ( Maybe.hs, Maybe.o )
[4 of 4] Compiling List             ( List.hs, List.o )
$
```

Podemos perceber que, como `List` importa outros módulos, esses foram recursivamente compilados. Assim, a rotina de compilação do `ghc` usualmente dispensa o uso de ferramentas como `make`. Note ainda que nenhum executável foi criado, mas apenas arquivos objeto.

Se `Module.hs` tem `module Main where` como sua primeira linha, então o `ghc` espera que haja uma função `main :: IO ()` definida em `Module.hs` (`IO` será explicada futuramente). Caso haja, `ghc` gera um executável `Module`. Como exemplo, vamos criar um arquivo `Hello.hs` com o seguinte conteúdo.

```
module Main where

import Prelude (IO, putStrLn, getLine, (>>=))

import Functions
import List

main :: IO ()
main = getLine >>= putStrLn . reverse
```

Há muitas coisas novas sendo usadas em `Hello.hs`. Basicamente, `IO` representa um valor que surge de operações de entrada e saída. Essa é a forma de Haskell separar funções com retorno *impuro* de funções com retorno simples. O valor `getLine :: IO [Char]` retorna uma string, obtida a partir do teclado, dentro de `IO`. Como o valor `getLine` pode mudar, a depender do que se digita no teclado, ele não é considerada puro. A função `putStrLn :: String -> IO ()` recebe uma string e a imprime na tela, retornando uma tupla vazia dentro de `IO`. Como isso envolve uma operação de entrada e saída, o retorno de `putStrLn` é representado por `IO`. Por fim, o operador `(>>=) :: Monad m => m a -> (a -> m b) -> m b` pode ser visto como uma composição de funções mais sofisticada (entraremos em detalhes sobre ele futuramente).

Agora, compilamos `Hello.hs`.

```
$ ghc Hello -W
[1 of 5] Compiling Bool             ( Bool.hs, Bool.o )
[2 of 5] Compiling Functions        ( Functions.hs, Functions.o )
[3 of 5] Compiling Maybe            ( Maybe.hs, Maybe.o )
[4 of 5] Compiling List             ( List.hs, List.o )
[5 of 5] Compiling Main             ( Hello.hs, Hello.o )
Linking Hello ...
$
```

Observe que, dessa vez, o processo de compilação se estende até a fase de *linking*. Mais uma vez, `ghc` compila os módulos necessários recursivamente. Podemos testar o que o executável `Hello` faz.

```
$ ./Hello
Hello
olleH
$
```

Isso deve ser suficiente para que possamos compilar programas, em um primeiro momento.

3.8 Implementando um grafo

Aqui, desenvolvemos em `Graph.hs` algumas definições para lidar com grafos. Nosso objetivo é ilustrar o uso de `SeqTree`. Um grafo $G = (V, E)$ tem um conjunto de vértices V e um conjunto de arestas E, sendo que uma aresta relaciona dois vértices. Seu conteúdo inicial se encontra a seguir.

```
module Graph where

import Prelude (Int, (+), (-), Bool(True, False),
                (==), (<=))

import Bool
import Functions
import List
import Maybe
import qualified SeqTree as Seq

type Vertex = Int

data Graph = G Int (Seq.SeqTree [Vertex])
```

Usamos valores inteiros para representar nossos vértices, e deixamos isso expresso de forma clara com o uso de `type`, que apenas cria sinônimos para tipos já existentes. Dessa forma, um grafo com n vértices tem seus vértices numerados de 1 a n. Nosso tipo `Graph` tem dois campos: um inteiro, indicando seu número de vértices; uma `SeqTree [Vertex]`, que associa a cada vértice v uma lista com os outros vértices das arestas relacionadas a v. Vamos desenvolver uma função para criar um grafo com nenhum vértice.

```
empty :: Graph
empty = G 0 Seq.empty
```

A função `empty` é bem simples. Ela apenas cria um grafo com 0 vértices e, portanto, com uma `SeqTree` vazia. A seguir, criamos um modo de adicionar um vértice ao grafo.

```
addVertex :: Graph -> Graph
addVertex (G n adj) = G (n + 1) (Seq.insert adj [])
```

Ao adicionar um vértice, `addVertex` precisa atualizar o número de vértices do grafo. Além disso, o vértice recém-criado faz parte de nenhuma aresta, portanto inserimos `[]` em `adj`. Aproveitamos `addVertex` para criar uma função que retorna grafos sem arestas.

```
edgeless :: Int -> Graph
edgeless n = cond (n <= 0) empty (addVertex g) where
        g = edgeless (n - 1)
```

A função a seguir verifica se um grafo tem um certo vértice.

```
hasVertex :: Graph -> Vertex -> Bool
hasVertex (G n _) v = and [1 <= v, v <= n]
```

Como consideramos que um grafo tem seus `n` vértices numerados entre 1 e `n`, `hasVertex` verifica justamente isso. Vale notar também que, se um grafo tem nenhum (zero) vértices, então `hasVertex` retorna `False` para qualquer vértice recebido como entrada.

Definimos a função `vertices`, que retorna uma lista com os vértices do grafo, por uma questão de conveniência. Fazemos uso de `iterate`, proposta no Exercício 10.

```
vertices :: Graph -> [Vertex]
vertices (G n _) = take n (iterate (+1) 1)
```

Agora vamos definir a adição de uma aresta.

```
addEdge :: Graph -> Vertex -> Vertex -> Graph
addEdge g@(G n adj) u v = cond verticesExist g' g where
        verticesExist = and [g `hasVertex` u, g `hasVertex` v]
        g' = G n adj'
        adj' = Seq.update adj u (v:)
```

Primeiro, `addEdge` verifica se o grafo `g` tem os vértices `u` e `v`. Aqui, notamos mais uma novidade sintática: funções "binárias" podem ser usadas como operadores infixos, isto é, entre seusa argumentos. Para isso, o nome da função deve estar entre crases (no código acima, o nome da função aparece entre aspas simples porque o LaTeX interpreta crases soltas como aspas simples (mas olha só, ele tem um comando para imprimir seu próprio nome de um jeito bem bizarro)). Caso `u` e `v` sejam vértices de `g`, a lista de `u` é atualizada de forma que `v` seja vizinho de `u`. Isso indica que há uma aresta de `u` para `v`, já que `v` aparece na lista de `u`. Com isso, estamos implementando um grafo onde as arestas têm direção, isto é, estamos implementando um grafo direcionado.

Os vizinhos de um vértice `v` são os vértices que aparecem em sua lista. Note que, a partir de `v`, podemos ir para qualquer vértice de sua lista, enquanto que o contrário nem sempre é verdade. A função `neighbors` retorna os vizinhos de um vértice em um grafo, aproveitando o fato de que `adj`, de acordo com a definição anterior, armazena justamente os vizinhos de cada vértice.

```
neighbors :: Graph -> Vertex -> [Vertex]
neighbors (G _ adj) v = fromMaybe [] (Seq.lookup adj v)
```

Na definição acima, note o uso de `fromMaybe` para tratar o caso de `v` não ser um vértice do grafo. Nesse caso, convencionamos que se um vértice não pertence ao grafo, ele não tem vizinhos. Para verificar se é possível ir de um vértice `u` para um certo vértice `v`, é preciso verificar se `v` é vizinho de `u`.

```
hasEdge :: Graph -> Vertex -> Vertex -> Bool
hasEdge g u v = f (neighbors g u) where
        f = isJust . find (==v)
```

Na definição acima, tomamos os vizinhos do vértice `u`. Em seguida, verificamos se `v` se encontra dentre os vizinhos de `u`. A função `isJust`, por fim, retorna o valor booleano apropriado. Perceba que, pela definição de `addEdge`, `hasEdge g u v == hasEdge g v u` nem sempre é `True`. Fizemos uso de `find`, proposta na Subseção 4.2.

Essas operações são suficientes para a construção de grafos. Sobre elas, seria possível definir algumas operações mais sofisticadas.

É preciso observar, ainda, que como `SeqTree` é limitada em sua remoção, nossa implementação de grafo não pode ter uma remoção elementar de vértices. Perceba também que, como cada vértice tem seus vizinhos representados como uma lista, a remoção de uma aresta não seria eficiente. O mais adequado seria representar os vizinhos de um vértice como uma árvore binária de busca balanceada, mas não temos implementado tal estrutura.

Após a constatação de que `Graph` não pode ter operações de remoção viáveis, fechamos nosso módulo com a seguinte alteração em seu início.

```
module Graph (Graph, empty, edgeless,
              addVertex, hasVertex, vertices,
              addEdge, neighbors, hasEdge) where
```

Isso conclui as operações básicas em `Graph.hs`.

3.9 Filas de prioridade

Vamos desenvolver em `Heap.hs` as definições necessárias para a manipulação de filas de prioridade. Vamos desenvolver uma fila de prioridade mínima.

O conteúdo inicial de `Heap.hs` é dado a seguir.

```
module Heap where

import Prelude (Ord, (<=))

import Bool
import Maybe

data Heap k v = Empty
              | Heap (k, v) [Heap k v]
```

Nosso tipo `Heap k v` representa uma *pairing heap*. `Heap k v` associa a cada valor de tipo `v` uma chave de tipo `k`, que é tratada como sua prioridade. Em sua definição, temos dois casos: `Heap` pode ser vazia, representada por `Empty`; pode ter um par de chave e valor em sua raiz, bem como uma lista de *heaps*. Assim, `Heap` pode ser vista como uma árvore (não necessariamente binária, já que uma lista pode ter qualquer número de elementos).

A ideia de manter uma lista de *heaps* permite que a execução de certas operações seja postergada o máximo possível. Com isso, o custo de execução de uma sequência de (digamos n) operações pode ser desbalanceado: por exemplo, as $n-1$ primeiras operações podem ser postergadas, enquanto a n-ésima precisa realizar o trabalho de todas as n operações. Em cenários como esse, é preciso considerar o *custo amortizado* de cada operação, isto é, considerar a média dos custos de todas as operações. Logo teremos ilustrações disso.

A função `lookup` retorna o elemento da raiz de `Heap`, que deve ser seu elemento prioritário. Sua definição não exige esclarecimentos e se encontra a seguir.

```
lookup :: Heap k v -> Maybe (k, v)
lookup Empty = Nothing
lookup (Heap keyval _) = Just keyval
```

Agora, definimos `merge`, que combina os elementos de duas filas de prioridade.

```
merge :: Ord k => Heap k v -> Heap k v -> Heap k v
merge Empty h2 = h2
merge h1 Empty = h1
merge h1@(Heap x@(key1, _) hs1) h2@(Heap y@(key2, _) hs2) =
        cond (key1 <= key2) h1' h2' where
                h1' = Heap x (h2:hs1)
                h2' = Heap y (h1:hs2)
```

Perceba que `merge` é muito peculiar. Independente do tamanho das filas de prioridade que `merge` recebe, ela executa em tempo constante! Ela sequer é recursiva. O que `merge` faz é muito simples: dadas duas filas de prioridade, a fila com o elemento mais prioritário passa a conter a outra. Observe ainda que "colocar" uma fila dentro da outra consiste apenas em adicionar uma fila à lista de filas da outra (essas ficam "esperando" que seus elementos sejam processados).

Agora que temos definido `merge`, podemos definir `insert` como a combinação de duas *heaps*. A função `insert` é descrita abaixo e faz exatamente isso.

```
insert :: Ord k => Heap k v -> (k, v) -> Heap k v
insert h keyval = merge (Heap keyval []) h
```

Para implementar a remoção, precisamos criar uma função que faça todo o trabalho que `merge` decidiu "procrastinar". A função `mergeAll` cumpre esse papel, e faz uso de `merge` para isso.

```
mergeAll :: Ord k => [Heap k v] -> Heap k v
mergeAll [] = Empty
mergeAll [h] = h
mergeAll (h1:h2:hs) = merge (merge h1 h2) (mergeAll hs)
```

Exercício 17 *A função* `mergeAll` *poderia ser definida como* `mergeAll = foldl merge Empty`*. Existe alguma desvantagem em definí-la dessa forma mais sucinta? Pense nas estruturas que as duas definições produzem.*

Aqui, `mergeAll` cria uma estrutura arbórea, combinando diversas *heaps* em uma só. Podemos ver sua definição em dois tempos: primeiro, ela define a combinação das *heaps* duas a duas, da esquerda para a direita; depois, ela faz a combinação cumulativa desses pares, combinando a *heap* mais à direita com as demais *heaps*, uma de cada vez e da direita para a esquerda.

Agora estamos aptos a definir a remoção do elemento mais prioritário. A função `pop` apenas remove a raiz da *heap*, em seguida combinando sua lista de *heaps* em uma só, que é retornada pela função. Sua definição elementar é dada abaixo.

```
pop :: Ord k => Heap k v -> Heap k v
pop Empty = Empty
pop (Heap _ hs) = mergeAll hs
```

Podemos entender agora que existe um motivo para a definição de `mergeAll`. Em uma operação de remoção, o fato de `mergeAll` combinar as *heaps* aos pares faz com que o número de *heaps* a serem combinadas em uma remoção futura seja em torno da metade do número de *heaps* combinadas nessa remoção.

Com isso, podemos analisar o seguinte cenário, que começa com uma *heap* vazia: após uma sequência de n inserções de elementos (no pior caso, em ordem decrescente de prioridade), a primeira remoção custa $O(n)$ e, por conta do "segundo tempo" de `mergeAll`, as $\lceil \frac{n}{2} \rceil$ remoções seguintes custam $O(1)$; graças ao "primeiro tempo" de `mergeAll`, a próxima remoção custa $O(n)$, mas agora apenas as $\lceil \frac{n}{4} \rceil$ remoções seguintes custam $O(1)$; assim, uma sequência de n remoções tem $O(\log n)$ remoções custando $O(n)$, enquanto as demais custam $O(1)$, e portanto as n remoções custam, juntas, $O(n \log n)$; em média, cada uma das n remoções custa $O(\frac{n \log n}{n}) = O(\log n)$, e esse é o custo amortizado da remoção em nossa `Heap`.

Nossa última operação serve para alterar a chave do elemento mais prioritário de uma *heap*. Com isso, `Heap` permite remoções e alterações apenas no elemento mais prioritário. A definição de `changeKey` consiste em remover o elemento da raiz e reinserí-lo com uma nova chave.

```
changeKey :: Ord k => Heap k v -> k -> Heap k v
changeKey Empty _ = Empty
changeKey h@(Heap (_, val) _) key' = insert (pop h) (key', val)
```

Para permitir que o usuário crie uma fila de prioridades vazia, encapsulamos `Empty` como segue.

```
empty :: Heap k v
empty = Empty
```

Para encerrar o módulo, definimos a lista de exportação de `Heap.hs`, alterando seu início.

```
module Heap (Heap, empty, lookup, insert, pop, changeKey) where
```

Isso deve ser o bastante para termos uma fila de prioridade bem definida.

Exercício 18 *Uma lista pode ser utilizada como uma pilha, dado que permite inserção e remoção em* $O(1)$ *na sua extremidade esquerda. Tome o tipo* `Queue a`*, definido como segue.*

```
module Queue(Queue, empty, enqueue, dequeue) where

import Prelude ()

data Queue a = Q [a] [a]

empty :: Queue a
empty = Q [] []

enqueue :: Queue a -> a -> Queue a
enqueue = undefined

dequeue :: Queue a -> Maybe (a, Queue a)
dequeue = undefined
```

Implemente `enqueue` *e* `dequeue`*, e argumente que* `dequeue` *tem custo amortizado* $O(1)$*. Escreva sua implementação em* `Queue.hs`*.*

3.10 Classes

Nesta subseção, tratamos de descrever algumas classes importantes de Haskell. Lembramos que uma classe é apenas um conjunto de funções, e dizemos que um tipo é instância de uma classe quando implementa suas funções. Por força de analogia, o conceito de classe em Haskell muito se assemelha ao de interface em Java.

Começamos com uma classe bem simples, `Eq`. Essa classe já foi vista anteriormente, e contém comparadores relacionados com o conceito de igualdade. Usando `:info Eq` no `ghci`, obtemos uma saída parecida com essa.

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

        (/=) :: a -> a -> Bool
        x /= y = not (x == y)
        {-# MINIMAL (==) | (/=) #-}
```

A despeito do que vínhamos fazendo (reimplementando quase tudo), não estamos aptos a redefinir as classes padrão de Haskell. Isso se dá porque há tipos cuja definição é intrínseca ao compilador, e assim não podemos "abrir" esses tipos e torná-los instâncias de classes. Por exemplo, não seria muito prático tornar `Int` uma instância de `Eq` sem usar sua representação binária (e como tentamos nos manter fiéis ao paradgma, temos de abstrair a representação de `Int` (apesar disso, Haskell tem operações a nível de *bit*)).

Dito isso, não escreveremos as definições de classe aqui apresentadas em nenhum arquivo. Elas apenas servem para ilustrar as definições padrão das funções.

Voltando à classe `Eq`, percebemos que ambas as suas funções têm definições padrão. Nesse caso, basta implementar uma delas para que um tipo se torne instância de `Eq`. Isso é dito explicitamente no comentário logo abaixo da classe. Como exemplo, tornar as listas instâncias de `Eq` poderia ser feito como segue.

```
instance Eq a => Eq [a] where
        [] == [] = True
        [] == _ = False
        _ == [] = False
        (x:xs) == (y:ys) = and [x == y, xs == ys]
```

Podemos ver que duas listas são comparáveis por igualdade apenas quando o tipo de seus elementos também é. Isso é dito explicitamente na restrição feita sobre `a`, na primeira linha.

Algo interessante de se notar é que, na definição acima, o comparador `(==)` é usado de duas formas distintas. No último caso, `x == y` faz menção à implementação de `(==)` que é definida

em `Eq a`, enquanto `xs == ys` é referente à instância que está sendo definida (é uma chamada recursiva), já que `xs` e `ys` são listas.

Exercício 19 *Faça de* `Maybe` *uma instância de* `Eq`.

A classe `Ord` contém os comparadores relacionados com o conceito de ordenação. Sua definição pode ser vista a seguir.

```
class Eq a => Ord a where
        compare :: a -> a -> Ordering
        compare x y = cond (x <= y) (cond (x == y) EQ LT) GT

        (<) :: a -> a -> Bool
        x < y = (compare x y) == LT

        (<=) :: a -> a -> Bool
        x <= y = (compare x y) /= GT

        (>) :: a -> a -> Bool
        (>=) :: a -> a -> Bool
        max :: a -> a -> a
        min :: a -> a -> a
        {-# MINIMAL compare | (<=) #-}
```

Exercício 20 *Complete as definições padrão da classe* `Ord`*. Tente fazê-las em termos de* `compare` *ou* `(<=)`*.*

Como exemplo de uma de suas instâncias, mostramos como a lista pdoe ser descrita como uma instância de `Ord`.

```
instance Ord a => Ord [a] where
        [] <= _ = True
        _ <= [] = False
        (x:xs) <= (y:ys) = cond (x == y) (xs <= ys) (x < y)
```

Exercício 21 *Faça de* `Maybe` *uma instância de* `Ord`*.*

Tratamos agora de duas classes, ambas relacionadas com a conversão de valores. A classe `Show` contém funções relacionadas com a conversão de tipos em strings. As instâncias de `Show`, portanto, podem todas ser representadas como strings. A classe `Read`, por sua vez, agrupa as funções relacionadas com a conversão de strings em valores de outros tipos. Assim, as instâncias de `Read` podem ter seus valores obtidos a partir de strings.

```
class Show a where
        showsPrec :: Int -> a -> ShowS
        show :: a -> String
        showList :: [a] -> ShowS
        {-# MINIMAL showsPrec | show #-}

class Read a where
        readsPrec :: Int -> ReadS a
        readList :: ReadS [a]
        GHC.Read.readPrec :: Text.ParserCombinators.ReadPrec.ReadPrec a
        GHC.Read.readListPrec :: Text.ParserCombinators.ReadPrec.ReadPrec [a]
        {-# MINIMAL readsPrec | readPrec #-}
```

Enquanto podemos criar instâncias da classe `Show` de forma simples, o mesmo não pode ser dito de `Read`. Utilizamos as instâncias já existentes de `Read` para converter strings em valores dos tipos básicos de Haskell. Para isso, usamos `read :: Read a => String -> a`, uma função que nem mesmo faz parte da classe `Read`. Exemplificamos a seguir o uso de `show` e `read`.

```
$ ghci
Prelude> show 25
"25"
Prelude> show [1, 2, 3]
"[1,2,3]"
Prelude> read "4"
*** Exception: Prelude.read: no parse
Prelude> read "4" :: Int
4
Prelude> read "4" :: Double
4.0
Prelude> read "[1,␣2,␣3]"
*** Exception: Prelude.read: no parse
Prelude> read "[1,␣2,␣3]" :: Int
*** Exception: Prelude.read: no parse
Prelude> read "[1,␣2,␣3]" :: [Int]
[1,2,3]
Prelude> read "[1,␣2,␣3]" :: [Double]
[1.0,2.0,3.0]
Prelude> :q
Leaving GHCi.
$
```

Podemos perceber que `show` e `read` são úteis principalmente quando operando com arquivos (vamos aprender a utilizar arquivos em algum laboratório). Exemplificamos como lista poderia ser feita instância de `Show`.

```
instance Show a => Show [a] where
        show [] = "[]"
        show xs = "[" ++ f xs ++ "]" where
                f = concat . intercalate "," . map show
```

Nessa definição, apenas o segundo caso precisa de atenção. A função `f` transforma todos os elementos da lista em strings (podemos fazer isso por conta de `Show a`), em seguida intercala a lista resultante com `","`, e por fim concatena todas as strings da lista. A nossa implementação de `show` apenas chama `f` e cerca seu resultado com colchetes.

Exercício 22 *Faça de* `Maybe` *uma instância de* `Show`.

Em seguida, temos a classe `Bounded`, com as funções `minBound` e `maxBound`. Os tipos que são instâncias de `Bounded` têm limites inferior e superior. São poucos os tipos que instanciam `Bounded`: `Word`, `Ordering`, `Int`, `Char` e `Bool`. Além desses, tuplas com componentes desse tipo instanciam `Ordering`.

Para tipos numéricos, temos a classe `Num`.

```
class Num a where
        (+) :: a -> a -> a
        (-) :: a -> a -> a
        (*) :: a -> a -> a
        negate :: a -> a
        abs :: a -> a
        signum :: a -> a
        fromInteger :: Integer -> a
        {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
```

Os tipos `Word`, `Integer`, `Int`, `Float` e `Double` são instâncias de `Num`, e suas definições são dadas em baixo nível. Observe ainda que `Integer` não é instância de `Bounded`. Isso se dá porque `Integer` é um tipo que representa inteiros com precisão arbitrária, isto é, o limite de representação de `Integer` é dado pela memória do computador em uso, e não por uma quantidade constante de *bits*. A desvantagem de `Integer` é que suas operações são mais custosas que as de `Int`.

Note ainda que `Num` não contém a operação de divisão, já que nem todos os tipos numéricos conseguem representar precisamente resultados de divisões arbitrárias. Para isso existe a classe `Fractional`.

```
class Num a => Fractional a where
        (/) :: a -> a -> a
        recip :: a -> a
        fromRational :: Rational -> a
        {-# MINIMAL fromRational, (recip | (/)) #-}
```

Em `Fractional`, estão contidas as operações de divisão, inversa e conversão a partir de `Rational`, e esse representa racionais em precisão arbitrária usando dois `Integer`s. Estendendo `Fractional`, temos a classe `Floating`, que agrupa operações envolvendo trigonometria, potências e logaritmos. Como seu nome sugere, `Floating` tem como instâncias apenas representações numéricas de pontos flutuantes: `Float` e `Double`.

A classe `Real` trata de conversões para o tipo `Rational`, e portanto contém apenas a função `toRational`. Já a classe `Integral` trata de divisões com restos em inteiros. A classe `RealFrac`, por sua vez, cuida de truncamentos e arredondamentos.

```
class (Real a, Fractional a) => RealFrac a where
        properFraction :: Integral b => a -> (b, a)
        truncate :: Integral b => a -> b
        round :: Integral b => a -> b
        ceiling :: Integral b => a -> b
        floor :: Integral b => a -> b
        {-# MINIMAL properFraction #-}
```

Das funções de `RealFrac`, apenas `properFraction` exige esclarecimentos. Essa função retorna um valor numérico decomposto em sua parte inteira e sua parte fracionária. Encerrando as classes de tipos numéricos, temos `RealFloat`, que reúne diversas funções a respeito de representação em ponto flutuante.

Na próxima subseção, tratamos de classes que abstraem o uso de funções, atribuindo semânticas a estruturas de dados.

3.11 `Functor`, `Applicative` e `Alternative`

Aqui, tratamos das três classes mencionadas no título. Antes de prosseguir, contudo, vamos adicionar a seguinte definição a `Functions.hs`.

```
($) :: (a -> b) -> a -> b
f $ x = f x

infixl 0 $
```

É isso mesmo. O operador `($)` aplica seu segundo argumento à função recebida como primeiro argumento. Embora pareça fútil, o fato de Haskell permitir aplicações parciais de funções dá utilidade a `($)`. A última linha atribui nível de precedência 0 a `($)` (o mais baixo possível), e também o faz associativo pela esquerda.

O operador `($)` é apresentado nesse contexto por representar aplicações "normais" de função. No texto a seguir, veremos operadores que representam aplicações "especiais" de função.

Tratamos inicialmente de `Functor`, que tem a seguinte definição.

```
class Functor (f :: * -> *) where
        fmap :: (a -> b) -> f a -> f b
        (<$) :: a -> f b -> f a
        {-# MINIMAL fmap #-}
```

Precisamos fazer uma pausa para explicar o que significa `* -> *`. Em Haskell, valores têm tipos (`1` pode ser do tipo `Int` ou `Integer`, por exemplo), assim como tipos têm *kinds* (*kind* pode significar tipo, então seria estranho traduzir). Basicamente, um *kind* é um tipo de tipo, e cada *kind* é caracterizado pelo número e *kinds* de seus parâmetros. Há os tipos que não são parametrizados (têm 0 parâmetros), como `Int` e `Char`. Esses já estão completos e têm *kind* `*`. Há os tipos que têm exatamente um tipo completo como parâmetro. Esses têm *kind* `* -> *`. Listas, `Maybe` e `SeqTree` têm *kind* `* -> *`, e se tornam completos apenas quando parametrizados por um tipo completo (`[Char]`, `Maybe Int`, `SeqTree (Maybe Char)`).

Podemos interpretar o *kind* `* -> *` como a classe de tipos que precisam de um tipo completo (`*`) para produzir (`->`) um tipo completo (`*`). Podemos entender `* -> * -> *` de forma análoga, e esse é o *kind* de `BSTree`.

Há mais detalhes a se explicar sobre *kinds*. Como esses não são necessários aqui, os desconsideramos.

Voltando a falar de `Functor`, vemos que há duas funções. A função `fmap` aplica uma função aos elementos contidos em um valor de um tipo parametrizado. O operador `(<$)`, por sua vez, substitui todos os elementos de um valor de um tipo parametrizado por um certo elemento. Como exemplo de `Functor`, temos as listas, e sua instanciação é ilustrada a seguir.

```
instance Functor [] where
        fmap = map
```

Essa definição é padrão de Haskell, e não precisamos implementá-la. Devemos, no entanto, instanciar `Maybe` como `Functor`, e a definição seguinte deve se encontrar em `Maybe.hs`.

```
instance Functor Maybe where
        fmap _ Nothing = Nothing
        fmap f (Just x) = Just (f x)
```

Perceba que não é preciso definir `(<$)`, uma vez que existe a implementação padrão `(x <$) = fmap f where f _ = x`. Fazemos também de `BSTree k` uma instância de `Functor`, pondo o código que segue em `BSTree.hs`.

```
instance Functor (BSTree k) where
        fmap = map
```

Exercício 23 *Faça de* `SeqTree` *uma instância de* `Functor`*.*

Exercício 24 *Como* `Either` *poderia instanciar* `Functor`*?*

Uma vez que temos visto algumas instâncias da classe `Functor`, apresentamos o operador `(<$>) :: Functor f => (a -> b) -> f a -> f b`. Esse operador equivale à função `fmap`, e sua assinatura sugere uma comparação com `($)`. Trazemos uma breve ilustração de seu uso.

```
Prelude> :t ($)
($) :: (a -> b) -> a -> b
Prelude> :t (<$>)
(<$>) :: Functor f => (a -> b) -> f a -> f b
Prelude> (*2) $ 3
6
Prelude> (*2) $ [1,2,3]

<interactive>:4:1: error:
        * Non type-variable argument in the constraint: Num [a]
          (Use FlexibleContexts to permit this)
        * When checking the inferred type
                it :: forall a. (Num a, Num [a]) => [a]
Prelude> (*2) <$> [1,2,3]
[2,4,6]
Prelude> (*2) <$> Nothing
Nothing
Prelude> (*2) <$> (Just 4)
Just 8
```

Vemos a distinção entre `($)` e `(<$>)`. Enquanto `($)` permite a aplicação de uma função a valores "soltos", o operador `(<$>)` aplica a função a valores "contidos" em instâncias de `Functor`. Com isso, damos os primeiros passos para interpretar estruturas de dados como "contextos".

Como Haskell tem suas funções como habitantes de primeira classe, é perfeitamente possível que funções sejam guardadas em estruturas de dados. Podemos imaginar que essas funções estão sob um certo "contexto", e definitivamente queremos utilizá-las (senão não faria sentido guardá-las). Essa é a motivação básica de `Applicative`: aplicar funções contidas em estruturas de dados.

```
class Functor f => Applicative (f :: * -> *) where
        pure :: a -> f a
        (<*>) :: f (a -> b) -> f a -> f b
        GHC.Base.liftA2 :: (a -> b -> c) -> f a -> f b -> f c
        (*>) :: f a -> f b -> f b
        (<*) :: f a -> f b -> f a
        {-# MINIMAL pure, ((<*>) | liftA2) #-}
```

Percebemos que instâncias de `Applicative` devem ter o mesmo *kind* de instâncias de `Functor`. Mais ainda: instâncias de `Applicative` devem ser também instâncias de `Functor`.

A função `pure` é responsável por "contextualizar" um valor, de acordo com a expressão em que é usada. O operador `(<*>)` aplica as funções contidas em seu primeiro argumento aos valores contidos no segundo. Trataremos das demais funções de `Applicative` em breve.

Como exemplo, listas são instância de `Applicative`.

```
instance Applicative [] where
        pure x = [x]

        [] <*> _ = []
        _ <*> [] = []
        (f:fs) <*> xs = f <$> xs ++ fs <*> xs
```

Enquanto instância de `Applicative`, o tipo lista representa computações não-determinísticas. Ilustramos esse conceito a seguir.

```
Prelude> xs = [1,2,3]
Prelude> ys = [10,20,30]
Prelude> (+) <$> xs <$> ys

<interactive>:8:1: error:
        * Couldn't match expected type 'Integer -> b'
                        with actual type '[Integer -> Integer]'
        * Possible cause: '(<$>)' is applied to too many arguments
        In the first argument of '(<$>)', namely '(+) <$> xs'
        In the expression: (+) <$> xs <$> ys
        In an equation for 'it': it = (+) <$> xs <$> ys
        * Relevant bindings include it :: [b] (bound at <interactive>:8:1)
Prelude> (+) <$> xs <*> ys
[11,21,31,12,22,32,13,23,33]
Prelude> xs = []
Prelude> (+) <$> xs <*> ys
[]
Prelude> xs = [2]
Prelude> (+) <$> xs <*> ys
[12,22,32]
Prelude> (+) <$> pure 5 <*> ys
[15,25,35]
Prelude> (+) <$> pure 5 <*> pure 7
12
```

Vemos que a expressão `(+) <$> xs <*> ys` retorna os resultados de todas as aplicações possíveis de `(+)` aos elementos de `xs` (como primeiro argumento) e `ys` (como segudo argumento). Primeiro, `(+) <$> xs` é uma lista de funções, mais precisamente as aplicações parciais de `(+)` aos elementos de `xs`. Com isso, `(+) <$> xs` é mais que um `Functor`, o que fica evidente com a expressão `(+) <$> xs <$> ys`, rejeitada pelo interpretador.

Se pensarmos em `xs` e `ys` como resultados de computações não-determinísticas, isto é, de computações que poden ter um número arbitrário de resultados (inclusive nenhum), passamos a ver a expressão `(+) <$> xs <*> ys` como a soma de dois valores não-determinísticos, e tal expressão deve representar todas as possibilidades de resultado.

Exercício 25 *Implemente a função* `sequences :: [a] -> [[a]]` *que, dada uma lista de elementos* `xs`*, retorna a lista de todas as sequências possíveis formadas por elementos de* `xs`*. As funções* `iterate` *e* `concat` *são úteis aqui. Pensar de forma indutiva e não-determinística nos leva a definir* `sequences` *em uma linha!*

Exercício 26 *Implemente a função* `subsets :: [a] -> [[a]]` *que, dada uma lista* `xs` *sem elementos repetidos, retorna uma lista de listas, onde cada sublista representa um subconjunto de* `xs`*. Pense de forma indutiva: como se obtém os subconjuntos de* `x:xs` *a partir dos subconjuntos de* `xs`*?*

Exercício 27 *Implemente a função* `sequenceA :: Applicative f => [f a] -> f [a]`*, que toma uma lista de valores* `f a` *e constrói uma lista de valores* `a`*, que é retornada dentro de* `f`*.*

Essa, no entanto, não é a única interpretação plausível que tornaria listas uma instância de `Applicative`. É possível pensar, por exemplo, que listas poderiam instanciar `Applicative` em termos de `zipWith`. Como um tipo pode instanciar uma classe no máximo uma vez, adicionamos as seguintes definições a `List.hs`.

```
newtype ZipList a = ZL [a]

zipList :: [a] -> ZipList a
zipList xs = ZL xs

getZipList :: ZipList a -> [a]
getZipList (ZL xs) = xs
```

Aqui, vemos nosso primeiro uso de `newtype`. A palavra `newtype` permite a construção de tipos muito simples, com um único caso e um único campo. Com essas restrições, os tipos criados com `newtype` não são compostos, mas apenas encapsulamentos de tipos já existentes. Isso não é útil para criar estruturas de dados, mas é muito conveniente para permitir que um tipo instancie uma classe mais de uma vez, agora em suas versões encapsuladas. Melhor ainda: como `newtype` pode apenas criar tipos dessa forma, o encapsulamento é desfeito em tempo de compilação, portanto o uso de tipos criados com `newtype` não traz custos adicionais para a execução de um programa.

Agora, devemos fazer de `ZipList` uma instância de `Functor` e `Applicative`, nessa ordem.

```
instance Functor ZipList where
        fmap f (ZL xs) = ZL (fmap f xs)

instance Applicative ZipList where
        pure = zipList . repeat
        (ZL fs) <*> (ZL xs) = ZL (zipWith ($) fs xs)
```

Note que `ZipList` torna-se instância de `Functor` com a mesma semântica das listas. No entanto, `ZipList` é instância de `Applicative` com uma semântica iterativa, tornando-se distinta das listas nesse aspecto. Ilustramos o uso de `ZipList` a seguir.

```
*List> xs = [1,2,3]
*List> ys = [10,20,30]
*List> (+) <$> xs <*> ys
[11,21,31,12,22,32,13,23,33]
*List> getZipList ((+) <$> zipList xs <*> zipList ys)
[11,22,33]
*List> getZipList ((+) <$> pure 5 <*> zipList ys)
[15,25,35]
```

Um outro exemplo de `Applicative` é `Maybe`. Definimos `Maybe` como `Applicative` pelo código que segue, escrito em `Maybe.hs`.

```
instance Applicative Maybe where
        pure x = Just x

        Nothing <*> _ = Nothing
        _ <*> Nothing = Nothing
        (Just f) <*> (Just x) = Just (f x)
```

Claramente, `Maybe` representa computações que podem falhar, tendo nenhum ou exatamente um resultado. Ilustramos esse conceito a seguir.

```
Prelude> (+) <$> Just 3 <*> Just 4
Just 7
Prelude> (+) <$> Just 3 <*> pure 4
Just 7
Prelude> (+) <$> Just 3 <*> Nothing
Nothing
Prelude> (+) <$> Nothing <*> pure 4
Nothing
```

Com isso, vemos que o uso de `Maybe` enquanto `Applicative` nos permite criar, a partir de computações que podem falhar, computações maiores com essa mesma semântica. O uso de `(<$>)` e `(<*>)` nos poupa do trabalho de tratar os casos de `Maybe`, e nos diz que computações com partes que podem falhar são computações que podem falhar como um todo.

Destacamos o papel de `pure`, evidenciado por esses exemplos. Mesmo que uma de nossas computações tenha uma semântica especial, pode haver partes dela que sejam simples computações determinísticas, feitas com o uso de valores *puros*. A função `pure` é usada justamente

para "promover" valores puros para os contextos de funções com semânticas especiais. Seu uso é indispensável, uma vez que a função `main` de um programa em Haskell é necessariamente uma função com semântica especial.

O operador `(<*)` tem o mesmo comportamento de `(*>)`, porém o papel de seus argumentos é permutado. Dito isso, nos concentramos em `(*>)`, cujo papel é aplicar a semântica de seu primeiro argumento (ignorando seus valores) no segundo. Vamos exemplificar o que isso significa.

```
Prelude> [] *> pure 3
[]
Prelude> [1,2,3] *> pure 3
[3,3,3]
Prelude> [1,2,3] *> [10,20]
[10,20,10,20,10,20]
Prelude> pure 3 *> pure 5
5
Prelude> Just 3 *> pure 5
Just 5
Prelude> Nothing *> pure 5
Nothing
Prelude> Just 3 *> Nothing
Nothing
Prelude> pure 3 *> Nothing
Nothing
```

Isso evidencia que `(*>)` é definido como `x *> y = f <$> x <*> y where f _ q = q`. O operador `(*>)` costuma ser utilizado quando se quer tomar apenas o valor semântico de uma computação, ignorando seu resultado. Por exemplo, `find p xs *> f xs` vale `Nothing` ou `Just (f xs)`, a depender se há um elemento em `xs` (ignorado por `(*>)`) satisfazendo `p`.

Por fim, a função `liftA2` traz uma outra forma de se utilizar `Applicative`. `liftA2` encapsula a aplicação explícita de `(<$>)` e `(<*>)`. Assim, as expressões `f <$> x <*> y` e `liftA2 f x y` são equivalentes.

Exercício 28 *Faça de* `BSTree k` *uma instância de* `Applicative`*. Qual seria uma semântica adequada para esse tipo? Lembre-se de produzir uma árvore binária de busca válida.*

Exercício 29 *Faça de* `SeqTree` *uma instância de* `Applicative`*. Como se trata de uma representação linear de dados,* `SeqTree` *sofre do mesmo dilema que as listas.*

Exercício 30 *Faça de* `Either a` *uma instância de* `Applicative`*.*

Uma função muito interessante de se implementar é `when`, que modifica a semântica de um `Applicative` conforme um valor booleano. Basicamente, `when` pode ser usada para condicionar a realização de uma certa computação. Sua definição é escrita em `Bool.hs`.

```
when :: Applicative f => Bool -> f () -> f ()
when True x = x
when False _ = pure ()

unless :: Applicative f => Bool -> f () -> f ()
unless = when . not
```

Além de `when`, definimos também sua versão negada, `unless`. Por fim, temos `forever :: Applicative f => f a -> f b`, que repete uma computação indefinidamente.

```
forever :: Applicative f => f a -> f b
forever x = x *> forever x
```

Agora vamos falar sobre `Alternative`, definida em `Control.Applicative`.

```
class Applicative f => Alternative (f :: * -> *) where
        empty :: f a
        (<|>) :: f a -> f a -> f a
        some :: f a -> f [a]
        many :: f a -> f [a]
        {-# MINIMAL empty, (<|>) #-}
```

A classe `Alternative` é instanciada por tipos que, de alguma forma, trazem o conceito de ser vazio ou não. A função `empty` retorna a representação "vazia" de um tipo, que geralmente é inferido pela expressão em que `empty` é utilizada. O operador `(<|>)` representa uma espécie de união, e combina dois valores com a propriedade de que `x <|> y == empty` sse `x == y == empty`.

As listas são instância de `Alternative`, da seguinte forma.

```
instance Alternative [] where
        empty = []
        (<|>) = (++)
```

Com isso, podemos instanciar `ZipList` como `Alternative` com a seguinte definição, que se encontra em `List.hs`.

```
instance Alternative ZipList where
        empty = zipList empty
        (ZL xs) <|> (ZL ys) = ZL (xs <|> ys)
```

Em se tratando de listas ou `ZipLists`, a definição de `Alternative` é bem simples e intuitiva. O tipo `Maybe` é instanciado como `Alternative` com as seguintes definições, que se encontram em `Maybe.hs`.

```
instance Alternative Maybe where
        empty = Nothing

        Nothing <|> y = y
        x <|> _ = x
```

As instâncias de `Alternative` são particularmente úteis para representar os resultados de computações que podem falhar. Enquanto computações não-determinísticas, representadas por listas, apenas têm seus resultados mesclados, o tipo `Maybe` apresenta um comportamento mais interessante: o segundo argumento de `(<|>)` é utilizado apenas quando o primeiro argumento representa uma falha. Dessa forma, podemos criar uma sequência de computações que podem falhar, de tal forma que cada uma delas é executada apenas quando todas as anteriores têm falhado.

Exercício 31 *Escreva a função* `asum :: Alternative f => [f a] -> f a`. `foldr` *é útil para isso.*

Descrevemos a função `guard`, que permite "traduzir" um valor booleano, `True` ou `False`, para um valor semântico de `Alternative`, não-vazio ou `empty`, respectivamente. Essa definição é escrita em `Bool.hs`.

```
guard :: Alternative f => Bool -> f ()
guard True = pure ()
guard False = empty
```

Numa seção futura, trataremos de `Monad`, que leva adiante as abstrações definidas em `Functor` e `Applicative`.

3.12 `fix` e Programação Dinâmica

Em Haskell, fazemos uso de recursão tanto para definir tipos como para controlar operações. É usual que as definições recursivas tenham um caso base, em que há uma expressão simples. No entanto, temos nos deparado com funções recursivas que não têm um caso base e que, não fosse a preguiça de Haskell, teriam suas chamadas sempre resultando em estouros de pilha. É o caso das funções `repeat` e `cycle`, propostas na Subseção 4.2.

Temos em `nats` mais um exemplo desse tipo de definição.

```
nats :: [Int]
nats = 1 : map (+1) nats
```

Claramente, `nats` é baseada na definição indutiva dos naturais, e de fato representa uma lista infinita contendo os naturais. Observamos o seguinte comportamento: uma vez que `map (+1)` retorne uma lista, essa será a cauda da lista retornada por `(1:)`; além disso, a lista retornada por `(1:)` é justamente o argumento passado para `map (+1)`. Podemos fazer uma análise similar para `repeat` e `cycle`. Reescrevemos `nats` de forma que isso fique evidente.

```
nats :: [Int]
nats = ((1:) . (map (+1))) nats
```

Se admitirmos que `f = (1:) . (map (+1))`, temos que `nats = f nats`. Nesse momento, pode ser bastante estranho e desconfortável perceber que a definição de `nats` depende exclusivamente de `f`, uma vez que `nats`, por mais que seja o argumento passado para `f`, é o retorno de `f (f (f (...)))`. Isso pode dar um nó na mente, mas é útil a ponto de generalizarmos esse conceito para funções com argumento e retorno de mesmo tipo. Escrevemos a seguinte definição em `Functions.hs`.

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

Embora simples, `fix` é um tanto estranha. Ela toma `f` e deve retornar um elemento de tipo `a`. Para isso, ela passa um elemento de tipo `a` para `f`, que é justamente `fix f`. No fim das contas, `fix f == f (f (f (f (...))))`.

Bem, não seria muito interessante passar uma função de assinatura `Int -> Int` para `fix`, por exemplo, afinal não poderíamos usar seu retorno.

```
*Functions Prelude> fix (+1)
*** Exception: stack overflow
*Functions Prelude>
```

Para usá-la de forma útil, é interessante que o retorno de `f` possa ser construído incrementalmente. O retorno de `nats`, por exemplo, cumpre esse requisito por ser uma lista.

```
nats :: [Int]
nats = fix $ (1:) . (map (+1))
```

Embora `fix` não tenha caso base, `nats` pode ser usada mesmo que seja escrita dessa forma, por conta da preguiça de Haskell.

Exercício 32 *Escreva as funções* `repeat` *e* `cycle` *em termos de* `fix`*.*

Exercício 33 *Defina* `iterate` *em termos de* `fix`*.*

Exercício 34 *Implemente* `forever` *em termos de* `fix`*.*

Exercício 35 *Tome a função* `range` *definida como segue.*

```
range :: Int -> [Int]
range 0 = [0]
range x = x : range (x - 1)
```

Escreva `range` *em termos de* `fix`*.*

Exercício 36 *Tome a seguinte função.*

```
f :: (Int -> Int) -> Int -> Int
f _ 0 = 1
f rec n = n * rec (n - 1)
```

A função `f` *parece estar calculando fatorial, mas é bizarra. Ela "conhece" a definição de fatorial, mas não podemos dizer que ela é recursiva, pois se define em termos de* `rec`*. No entanto, se* `rec` *calculasse fatorial,* `f rec` *seria umaa definição plausível de fatorial. Defina* `fact :: Int -> Int`*, que calcula fatorial, em termos de* `f` *e* `fix`*.*

No Exercício 36, nos deparamos com uma prática um tanto peculiar: criar uma versão não-recursiva de uma função recursiva, fazendo sua chamada recursiva em termos de uma outra função de mesma assinatura, recebida como argumento. Embora o Exercício 36 use essa "decomposição" de maneira fútil, podemos ir mais adiante. É possível, por exemplo, "decompor" uma função recursiva `f` e deixar suas chamadas recursivas a cargo de uma versão "especial" de `f`. Esse conceito pode ser utilizado para, dentre outras coisas, implementar versões memoizadas de funções recursivas.

Para ilustrar essa ideia, vamos tomar um caso clássico.

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

Basta uma verificação superficial para constatar que `fib` retorna o `n`-ésimo número de Fibonacci, e que essa não é uma boa implementação. Por diversão, vamos tomar medidas do quão ruim ela é. Para isso, vamos criar um arquivo `Memo.hs`, com o seguinte conteúdo inicial (provisório), e escrever a definição de `fib` nesse arquivo.

```
module Memo where

import Prelude (Int, (-), (+))

import Functions
import List
```

Agora vamos aos testes.

```
Prelude> :l Memo
[1 of 1] Compiling Memo             ( Memo.hs, interpreted )
Ok, one module loaded.
*Memo> :set +s
*Memo> fib 10
55
(0.01 secs, 92,888 bytes)
*Memo> fib 15
610
(0.01 secs, 388,032 bytes)
*Memo> fib 20
6765
(0.01 secs, 3,655,320 bytes)
*Memo> fib 25
75025
(0.12 secs, 39,882,672 bytes)
*Memo> fib 30
832040
(0.79 secs, 441,642,736 bytes)
*Memo> fib 35
9227465
(8.21 secs, 4,897,222,696 bytes)
*Memo>
```

Vamos analisar rapidamente o uso de tempo e espaço. Observamos que, da chamada `fib 25` para `fib 30`, o tempo de execução aumentou cerca de $6,5$ vezes, enquanto de `fib 30` para `fib 35` o aumento foi de aproximadamente $10,4$ vezes. Isso não é muito animador. Quanto ao uso de espaço, não é preciso muita experiência para estranhar o fato de `fib 35` ter gasto quase cinco gigabytes de memória. Nesse pequeno experimento, a única coisa boa que aprendemos foi o uso de `:set +s` para exibir tempo de execução e uso de memória.

Exercício 37 *Considere a seguinte definição.*

```
fix2 :: (a -> a) -> a
fix2 f = y where y = f y
```

Observe o seguinte experimento.

```
*Functions Prelude> :set +s
*Functions Prelude> f x = fix (x:)
(0.00 secs, 0 bytes)
*Functions Prelude> g x = fix2 (x:)
(0.00 secs, 0 bytes)
*Functions Prelude> head (drop 10000000 (f 2))
2
(1.21 secs, 640,063,368 bytes)
*Functions Prelude> head (drop 10000000 (g 2))
2
(0.07 secs, 63,416 bytes)
```

Explique o que torna `fix2` *mais eficiente que* `fix`*.*

Em vista do desastre que é a atual implementação de `fib`, propomos uma solução eficiente e elegante, que é praticamente um mascote da linguagem Haskell.

```
fib :: Int -> Int
fib = (fibs !!) where
        fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Nessa implementação, `fib` recebe `n` e toma o `n`-ésimo elemento de uma lista. Essa lista é justamente uma definição recursiva para a sequência de Fibonacci.

Escrita dessa forma, `fib` é eficiente porque, uma vez que os elementos de `fibs` são calculados, não há a necessidade de recalculá-los. Note como a primeira versão de `fib` precisa calcular um número de Fibonacci mais de uma vez. Enfim, vamos aos testes.

```
*Memo> :set +s
*Memo> fib 35
9227465
(0.14 secs, 84,040 bytes)
*Memo> fib 35
9227465
(0.00 secs, 71,760 bytes)
*Memo>
```

Perceba que, na primeira chamada `fib 35`, o elemento correspondente em `fibs` ainda não havia sido calculado. Como a segunda chamada não precisava calcular esse elemento, mas apenas retorná-lo, seu tempo de execução é irrisório.

Essa refatoração de `fib` exibe uma maneira muito elegante de se determinar a sequência de Fibonacci, e com isso armazená-la como uma lista na memória, produzindo uma espécie de *cache* para `fib`. No entanto, essa abordagem não se generaliza para outras funções, já que usa algo característico dos números de Fibonacci para produzir uma *cache*.

Uma terceira forma de escrever `fib` é apresentada abaixo.

```
fib :: Int -> Int
fib = fix f where
        f _ 0 = 0
        f _ 1 = 1
        f rec n = rec (n - 1) + rec (n - 2)
```

É aqui que entramos na toca do coelho. Primeiro, `fix` permite que a função `f` receba `f (f (f (f (...))))` como seu primeiro argumento. Segundo, isso é equivalente à nossa primeira definição de `fib`. Então por que escrevê-la de uma forma mais complicada? Antes de responder a essa pergunta, testamos essa definição.

```
*Memo> :set +s
*Memo> fib 35
9227465
(22.00 secs, 11,892,075,208 bytes)
*Memo>
```

Essa versão é ainda pior! Mas não tiramos proveito da "decomposição" da primeira versão de `fib`, aqui expressa como `f`. Se `rec` fosse uma versão eficiente de `f`, teríamos obtido uma implementação eficiente para `fib`.

Como `fib` tem assinatura `Int -> Int` e espera um argumento não-negativo, tratamos de criar um esquema de *cache* adequado para funções com essas características.

```
memo :: (Int -> b) -> Int -> b
memo f = (ys !!) where
        xs = iterate (+1) 0
        ys = map f xs
```

A definição de `memo` é bastante ingênua, estúpida até. Para uma função `f :: Int -> Int` que espere argumentos não-negativos, `f` e `memo f` são equivalentes. No entanto, há uma diferença na forma como produzem seus retornos. Dado um argumento `n`, todas as chamadas `f n` teriam o mesmo custo computacional. Por outro lado, fazendo `g = memo f`, a primeira chamada `g n` teria o mesmo custo de `f n`, valor esse que ficaria salvo em `ys`, de forma que as chamadas seguintes `g n` apenas buscariam pelo valor salvo.

Tomando a primeira versão de `fib`, é tentador considerar que `memofib = memo fib` seja eficiente. Perceba, todavia, que `memofib` tem o mesmo custo de `fib` sempre que chamada para um novo argumento.

```
*Memo> memofib = memo fib
*Memo> :set +s
*Memo> memofib 35
9227465
(8.17 secs, 4,897,239,864 bytes)
*Memo> memofib 35
9227465
(0.00 secs, 71,760 bytes)
*Memo> memofib 36
14930352
(13.08 secs, 7,923,836,624 bytes)
*Memo> memofib 37
24157817
(21.13 secs, 12,820,992,200 bytes)
*Memo>
```

Esse experimento evidencia outra coisa muito chata sobre `memofib`. As duas chamadas `memofib 35` nos mostram que os retornos de `fib` realmente estão sendo salvos por `memo`. Porém, uma vez salvos `fib 35` e `fib 36`, era de se esperar que `memofib 37` tivesse um custo bem mais baixo que o apresentado. É por isso que precisamos de `fix`.

Tomando `f` como a versão "decomposta" de `fib`, `f rec` é uma implementação eficiente de `fib` desde que `rec` também seja. Ora, nesse caso `memo (f rec)` é uma implementação eficiente de `fib`. Usamos `fix` para ligar as duas pontas.

```
fib :: Int -> Int
fib = fix (memo . f) where
        f _ 0 = 0
        f _ 1 = 1
        f rec n = rec (n - 1) + rec (n - 2)
```

E aqui começa a festa do chá. Vamos usar a sanidade que nos resta para testar essa definição de `fib`.

```
*Memo> :set +s
*Memo> fib 35
9227465
(0.02 secs, 1,366,096 bytes)
*Memo> fib 35
9227465
(0.00 secs, 71,760 bytes)
*Memo> fib 36
14930352
(0.00 secs, 255,872 bytes)
*Memo> fib 36
14930352
(0.00 secs, 72,616 bytes)
*Memo> fib 37
24157817
(0.00 secs, 265,512 bytes)
*Memo> fib 37
24157817
(0.00 secs, 72,736 bytes)
*Memo>
```

Conseguimos mais uma versão de `fib` verdadeiramente eficiente. Para entender como ela funciona, note que `f` ignora seu primeiro argumento nos casos base. Isso e a preguiça de Haskell impedem `fix` de provocar um estouro de pilha. Observe também que, graças a `fix`, `g = memo . f` recebe `g (g (g (g (...))))` como primeiro argumento. Para entender a importância disso, perceba que `memo . f` retorna uma versão eficiente de `f` quando recebe uma versão eficiente de `f`. Nesse caso, basta passar para `memo . f` a função `memo . f` recebendo uma versão eficiente de `f`. Mas isso contém o mesmo problema da situação anterior! É aí que entra `fix`, permitindo passar `memo . f` para `memo . f` infinitamente.

Com tudo isso, criamos uma boa implementação para `fib` baseada em *cache*, mas dessa vez a implementação da *cache* é genérica. Bem, genérica para funções `f :: Int -> Int` que esperam argumentos não-negativos: "decompomos" `f` em `f' :: (Int -> Int) -> Int -> Int` e fazemos `fix (memo . f')`. Mas e quanto a funções quaisquer `a -> b`?

Antes de abordar essa questão, vamos resolver uma mais simples. Representar a *cache* como uma lista infinita não é a abordagem mais eficiente, por conta de seu acesso em tempo linear. Vamos, portanto, representar a *cache* como uma árvore binária. Para isso, precisamos de uma árvore binária infinita, definida em `InfinityTree.hs` como segue.

```
module InfinityTree (InfinityTree(Branch), find) where

import Prelude (Int, (==), (-), divMod,
                Functor(fmap), Applicative(pure, (<*>)))

import Bool

data InfinityTree a = Branch a (InfinityTree a) (InfinityTree a)

instance Functor InfinityTree where
        fmap f (Branch x lt rt) = Branch (f x) (fmap f lt) (fmap f rt)

instance Applicative InfinityTree where
        pure x = Branch x (pure x) (pure x)

        (Branch f flt frt) <*> (Branch x xlt xrt) =
                Branch (f x) (flt <*> xlt) (frt <*> xrt)

find :: InfinityTree a -> Int -> a
find (Branch x _ _) 0 = x
find (Branch _ lt rt) n = cond (r == 1)
                               (find lt q)
                               (find rt (q - 1)) where
    (q, r) = divMod n 2
```

A definição de `InfinityTree` representa uma árvore infinita, já que não permite folhas em sua estrutura. A única função de acesso a seus elementos é `find`, que "passeia" pela árvore conforme a paridade de um `Int` dado como entrada: um número ímpar (par) leva a busca para a esquerda (direita).

Com isso, podemos escrever uma versão mais eficiente de `memo`, que representa a *cache* em uma estrutura arbórea.

```
memo :: (Int -> b) -> Int -> b
memo f = find ftree where
        natTree = Branch 0 oddTree evenTree where
                oddTree = ((+1) . (*2)) <$> natTree
                evenTree = ((+2) . (*2)) <$> natTree
        ftree = f <$> natTree
```

Perceba que apenas a estrutura utilizada internamente sofreu modificações. Com isso, a nova implementação de `memo` tem a mesma assinatura da anterior, e portanto nossa definição de `fib` não precisa de alterações. Vamos fazer novos testes.

```
*Memo> :set +s
*Memo> fib 35
9227465
(0.01 secs, 1,440,632 bytes)
*Memo> fib 35
9227465
(0.00 secs, 69,872 bytes)
*Memo> fib 36
14930352
(0.00 secs, 240,064 bytes)
*Memo> fib 36
14930352
(0.00 secs, 70,680 bytes)
*Memo> fib 37
24157817
(0.00 secs, 246,248 bytes)
*Memo> fib 37
24157817
(0.00 secs, 70,680 bytes)
*Memo>
```

Exercício 38 *Verifique se existe ganho de desempenho em definir* `fib` *em termos de* `fix2`*, definida no Exercício 37.*

Isso evidencia que a nova implementação de `memo` traz algum ganho. Infelizmente, como é preciso planejar as próximas aulas, não abordaremos memoização para funções `a -> b`. Isso fica para a próxima versão da disciplina.

Exercício 39 *Escreva uma função* `decomp37 :: Int -> Maybe (Int, Int)` *que, dada uma entrada não-negativa n, decide se existem $x, y \in \mathbb{N}$ tais que $n = 3x + 7y$, e caso existam, retorna o par (x, y). É fácil escrever essa função tirando proveito de* `Maybe` *ser instância de* `Applicative` *e* `Alternative`*. Use* `memo` *e* `fix` *para criar uma implementação eficiente.*

Exercício 40 *Escreva uma função* `primeFactors :: Int -> [Int]`*, que retorna a decomposição em fatores primos de uma entrada não-negativa n. Para facilitar, crie uma lista infinita com todos os primos. Basta encontrar o menor primo p divisor de n e retorná-lo como cabeça, fazendo da cauda a chamada recursiva para n/p. Use* `memo` *e* `fix` *para implementar memoização em* `primeFactors`*.*

Exercício 41 *Dadas moedas de $2, 3$ e 7 centavos, vamos expressar x centavos, se possível, com o menor número de moedas. A função $f(x) = 1 + \min\{f(x-2), f(x-3), f(x-7)\}$ representa uma relação de recorrência que resolve esse problema. Determine os casos base e desenvolva uma função* `minCoins :: Int -> Maybe Int`*. Note que, para valores altos de x, a função f é computada mais de uma vez para alguns argumentos. Use* `memo` *e* `fix` *para evitar computar f mais de uma vez para o mesmo argumento. Tire proveito de* `Maybe` *ser* `Applicative`*.*

3.13 Mônadas

Já temos visto aplicações simples de funções, que costumam ser representadas pelo operador `($) :: (a -> b) -> a -> b`. Vimos também aplicações de funções simples a elementos contidos em um certo contexto. Esse tipo de aplicação é representado pelo operador `(<$>) :: Functor f => (a -> b) -> f a -> f b`. Inclusive, quando uma estrutura de dados permite que seu conteúdo seja trnsformado por uma função simples, ela pode ser vista como uma instância de `Functor`. Por último, nos deparamos com aplicações de funções, em um certo contexto, a elementos em um contexto semelhante. Essa forma de aplicação está associada ao operador `(<*>) :: Applicative f => f (a -> b) -> f a -> f b`. Quando uma estrutura de dados permite que seu conteúdo seja operado por funções contidas em uma estrutura similar, ela pode ser feita uma instância de `Applicative`.

Agora, vamos analisar funções que produzem um contexto a partir de elementos simples. Tais funções são ditas *monádicas*, e contextos que podem ser criados a partir de elementos simples são representados por estruturas que chamamos de *mônadas*. Tais tipos são agrupados na seguinte classe.

```
class Applicative m => Monad (m :: * -> *) where
        (>>=) :: m a -> (a -> m b) -> m b
        (>>) :: m a -> m b -> m b
        return :: a -> m a
        fail :: String -> m a
        {-# MINIMAL (>>=) #-}
```

Aqui, vemos um operador que utilizamos bastante nos laboratórios. Finalmente vamos entender `(>>=)`!

Em linhas gerais, o operador `(>>=)` toma valores do tipo `a` contidos em `m a`, onde `m` é uma mônada, e transforma cada um deles em `m b`, produzindo uma estrutura de tipo `m (m b)`. Vale ressaltar que isso também seria possível com um `Functor`, mas `Functor` não é o bastante para lidar com esse aninhamento de contextos. Em seguida, `(>>=)` "combina" os contextos interno e externo, produzindo a saída de tipo `m b`.

Antes de entendermos essa "combinação", vamos falar a respeito das demais funções que compõem a classe `Monad`. O operador `(>>)` ignora o conteúdo de `m a`, mas faz com que seu valor semântico influencie `m b`, produzindo a saída. Não há diferença entre `(>>)` e `(*>)`. Como `Applicative` é mais abrangente que `Monad`, preferimos fazer uso de `(*>)`.

A função `return` é equivalente a `pure`, e sua presença em `Monad` apenas cumpre o papel de não "quebrar" *legacy code*. Isso se deve a uma época em que as instâncias de `Monad` não precisavam ser instâncias de `Applicative`. Preferimos sempre o uso de `pure`, pois `return` leva esse nome para, dada a forma como é usada em *do notation*, fazer analogia ao paradigma imperativo. Por fim, a presença de `fail` em `Monad` é depreciada, e portanto a ignoramos.

Uma função muito importante para a concepção matemática de mônadas, mas que não faz parte de `Monad`, é `join :: Monad m => m (m a) -> m a`. Dificilmente precisaremos utilizá-la para instanciar `Monad`, mas conhcê-la pode nos permitir um melhor entendimento da "combinação" de contextos. Inclusive, a única finalidade de `join` é "combinar" contextos.

Vamos apresentar nossa primeira mônada, fazendo com que `Maybe` instancie `Monad`. As definições que seguem devem ser escritas em `Maybe.hs`.

```
instance Monad Maybe where
        Nothing >>= _ = Nothing
        (Just x) >>= f = f x
```

Analisar uma instância de `Monad` nos permite ver com mais clareza o que faz `(>>=)`, e porque ele costuma ser chamado de *binding operator*. Para `Maybe`, `(>>=)` retorna `Nothing` tão logo seu primeiro argumento é `Nothing`, do contrário aplica seu conteúdo a `f`, produzindo a saída.

Exercício 42 *Faça de* `Maybe` *uma instância de* `Monad`*, definindo* `(>>=)` *como uma chamada de* `maybe`*.*

No contexto de `Maybe`, o operador `(>>=)` permite a criação de uma cadeia de computações que falha quando uma de suas partes falha. Com o uso de `pure`, podemos encadear computações determinísticas, que produzem exatamente um resultado, com computações que podem falhar. Dessa forma, uma computação que produz `m a` pode ter seu resultado passado como entrada para uma computação que produz `m b`, e essa pode ser "encaixada" na entrada de uma terceira computação. Dizemos que `(>>=)` permite o *binding* de computações.

Agora, vamos ilustrar como o tipo lista instancia a classe `Monad`.

```
instance Monad [] where
        xs >>= f = concat $ f <$> xs
```

Podemos perceber que essa implementação de `(>>=)` evidencia a "combinação" de contextos, isto é, transforma uma lista de listas em uma lista simples. Primeiro, `f <$> xs` tem tipo `[[b]]`, e depois esse aninhamento de contextos é resolvido com `concat`. Para listas, portanto, `join = concat`.

Para o exercício que segue, vamos precisar da função identidade. Sua definição é escrita em `Functions.hs`.

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

Exercício 43 *Defina* `join` *em termos de* `(>>=)` *e* `id`*.*

Exercício 44 *Aqui, assumimos que são dadas implementações para as funções* `fmap` *e* `join`*. Escreva uma implementação para* `(>>=)` *em termos de* `fmap` *e* `join`*.*

Como listas representam computações não-determinísticas, chamamos a atenção para o impacto de `concat` quando usamos `(>>=)` para construir uma cadeia de computações não-determinísticas. Tome, por exemplo, uma cadeia de k computações não-determinísticas, em que cada uma produz dois resultados. O uso de um `concat` por *binding* vai produzir, ao final da cadeia, uma lista representando os 2^k resultados possíveis.

Exercício 45 *De forma similar a* `foldl :: (b -> a -> b) -> b -> [a] -> b`*, implemente a função* `foldM :: Monad m => (b -> a -> m b) -> b -> [a] -> m b`*.*

Exercício 46 *Tome a função* `f xs x = [xs, x:xs]`*. Podemos interpretar* `f` *como uma computação não-determinística, que pode retornar tanto* `xs` *como* `x:xs`*. Reimplemente* `subsets`*, proposta inicialmente no Exercício 26, como uma chamada de* `foldM`*.*

É natural nos perguntarmos o porquê dessa tendência de Haskell interpretar estruturas como semânticas, encapsulando-as em aplicadores especiais de funções. Haskell poderia muito bem deixar mais explícita a pureza de suas funções se não usasse tal artifício, e em troca escreveríamos definições mais extensas. No entanto, é esse artifício que permite uma representação para a impureza. Vamos falar da mônada `IO`.

```
Prelude> :info IO
newtype IO a
        = GHC.Types.IO (GHC.Prim.State# GHC.Prim.RealWorld
                -> (# GHC.Prim.State# GHC.Prim.RealWorld, a #))
-- Defined in 'GHC.Types'
instance Applicative IO -- Defined in 'GHC.Base'
instance Functor IO -- Defined in 'GHC.Base'
instance Monad IO -- Defined in 'GHC.Base'
instance Monoid a => Monoid (IO a) -- Defined in 'GHC.Base'
instance Semigroup a => Semigroup (IO a) -- Defined in 'GHC.Base'
```

Não entre em pânico. Basicamente, isso significa que `IO a` encapsula uma função que, dado um estado do mundo real, retorna um (possivelmente novo) estado mundo real junto de um valor do tipo `a`. Dessa forma, não devemos esperar que, dada `f :: a -> IO b`, `f x` tenha sempre o mesmo valor, uma vez que o valor contido em `IO b` depende também do estado do mundo real. Inclusive, `IO b` não apenas depende, mas pode alterar o estado do mundo real.

```
Prelude> f _ = getLine
Prelude> :t f
f :: p -> IO String
Prelude> :t (==) <$> f 1 <*> f 1
(==) <$> f 1 <*> f 1 :: IO Bool
Prelude> (==) <$> f 1 <*> f 1
aaa
aaa
True
Prelude> (==) <$> f 1 <*> f 1
aaa
bbb
False
```

Perceba que não é verdade que `f 1` é sempre igual a `f 1`. O conteúdo da `IO` que `f` retorna pode mudar, mesmo que seja usado o mesmo argumento. Mas como isso pode ser possível? As função de Haskell são puras! A função `f`, inclusive, é pura, mas seu retorno não é o valor de um tipo simples. O retorno de `f` traz consigo a semântica da impureza, representada pela mônada `IO`. Em suma, `f` é pura como qualquer outra função, mas seu retorno é impuro. É como se `IO` fosse mais uma advertência do que um tipo.

A necessidade prática de se trabalhar com impurezas obriga Haskell a ter uma forma de lidar com elas, do contrário Haskell seria inútil, incapaz de interagir com as coisas do mundo real (como a arquitetura de um computador). Em seus anos iniciais, havia diversas propostas de como as impurezas deveriam estar presentes em Haskell. Em algum momento houve o consenso de atribuir semântica a tipos de *kind* `* -> *`, e um tipo com esse *kind* teria o papel de indicar a impureza de seu conteúdo. Assim, houve a adoção das classes `Functor`, `Applicative` e `Monad`, representando conceitos que já existiam na Matemática.

Como sua implementação envolve elementos de baixo nível do compilador, não podemos "abrir" `IO` da mesma forma que fazemos com `Maybe` ou listas. Aí vem a outra sacada da adoção de `Functor`, `Applicative` e `Monad`: com as funções definidas nessas classes, não precisamos "abrir" nenhuma de suas instâncias. Embora possamos "abrir" `Maybe`, é mais coveniente usar as funções de `Functor`, `Applicative` e `Monad` para tratar esse tipo. No caso de `IO`, não se trata de coveniência. Essas três classes permitem que trabalhemos com valores impuros sem nos preocupar de onde vem sua impureza, ou como tais valores são implementados. Assim, `IO` é uma espécie de mônada "mágica", e o encapsulamento que promove permite que as definições de Haskell não se atenham a detalhes de baixo nível.

Agora, vamos lidar com algumas funções que nos auxiliam no tratamento de mônadas. Primeiro, definimos `mapM` em `Functions.hs`.

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f [] = pure []
mapM f (x:xs) = f x >>= \x' ->
                mapM f xs >>= \xs' ->
                pure  (x' : xs')
```

Nessa definição, fizemos uso de *funções anônimas*. Essencialmente, podemos definir uma função sem dá-la um nome utilizando `\` seguido de seus argumentos, com `->` indicando a definição da função. Por exemplo, podemos definir uma função sem dá-la um nome por meio de uma aplicação parcial, como em `(+1)`, ou através de um função anônima, como em `\x -> x + 1`.

Funções anônimas são especialmente úteis para definir o segundo argumento de `(>>=)`. Quando `(>>=)` "abre" seu primeiro argumento, é interessante podermos referenciar o seu "conteúdo". Usamos uma função anônima para receber esse "conteúdo", e assim podemos referenciá-lo através do argumento da função. Na definição de `mapM`, existem duas funções anônimas: a primeira recebe um argumento `x'` ("conteúdo" de `f x`) e é definida como uma chamada de `(>>=)`; a segunda, que ocorre dentro da primeira, recebe um argumento `xs'` ("conteúdo" de `mapM f xs`) e retorna `x':xs'` no contexto adequado. Note que, devido a essas

duas funções anônimas serem aninhadas, podemos usar o argumento da primeira dentro da segunda.

Preferimos essa definição de `mapM`, pois com ela podemos falar de funções anônimas e evidenciá-las como um bom recurso ao utilizar-se `(>>=)`. Existe, no entanto, uma definição bem mais simples de `mapM`.

Exercício 47 *Defina* `mapM` *em termos de* `map` *e* `sequenceA`*, proposta no Exercício 27.*

O Exercício 47 pode nos deixar pensando que `mapM` não precisa ser restrita a `Monad`, mas apenas a `Applicative`. De fato, isso é verdade. No entanto, como o primeiro argumento de `mapM` é uma função `a -> m b`, uma função que cria um contexto a partir de um elemento simples, ela é restrita a `Monad` não por necessidade, mas pelo fato de receber uma função monádica.

Vamos definir `mapM_ :: Monad m => (a -> m b) -> [a] -> m ()`, que ignora os resultados produzidos pela função monádica. A definição que segue deve ser escrita em `Functions.hs`.

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f [] = pure ()
mapM_ f (x:xs) = f x *> mapM_ f xs
```

Eis uma breve ilustração do uso de `mapM_`.

```
*Functions> import Prelude (putStrLn)
*Functions Prelude> mapM_ putStrLn ["aaa", "bbb", "ccc"]
aaa
bbb
ccc
```

Exercício 48 *Implemente* `sequenceA_ :: Applicative f => [f a] -> f ()`*, que é similar a* `sequenceA`*, proposta no Exercício 27, mas ignora os resultados das computações.*

Exercício 49 *Defina* `mapM_` *em termos de* `sequenceA_` *e* `map`*.*

Após uma longa sequência de abstrações, tratamos de coisas mais concretas. Vamos lidar com aspectos da linguagem que tem impacto direto no desempenho de nossos programas.

3.14 Avaliação estrita e Paralelismo

O conteúdo desta subseção é baseado no livro *Parallel and Concurrent Programming in Haskell*, de Simon Marlow.

Como bem sabemos, Haskell é uma linguagem de programação com avaliação preguiçosa, isto é, as definições de valores são avaliadas apenas quando esses são necessários. Podemos verificar isso.

```
Prelude> x = 1 + 1 :: Int
Prelude> :sprint x
x = _
Prelude> x
2
Prelude> :sprint x
x = 2
```

Aqui, observamos o uso de `:sprint`, um comando especial do `ghci` que não faz parte da linguagem Haskell. A função de `:sprint` é verificar o valor associado a um identificador, sem forçar sua avaliação: quando o identificador já foi avaliado, seu valor é exibido; caso contrário, exibe-se `_`. Percebemos, portanto, que `x` não foi avaliado assim que foi definido. Quando exibimos o valor de `x`, no entanto, ele precisa ser avaliado. Vamos tentar algo um pouco mais elaborado.

```
Prelude> x = 1 + 1 :: Int
Prelude> y = x + 1 :: Int
Prelude> :sprint x
x = _
Prelude> :sprint y
y = _
Prelude> seq y "eita"
"eita"
Prelude> :sprint x
x = 2
Prelude> :sprint y
y = 3
```

Claramente, `x` e `y` não foram avaliados assim que definidos. Vamos dizer que `x` e `y` estavam *suspensos*. Jamais chegamos a exibir os valores de `x` e `y`, e mesmo assim eles foram avaliados.

Isso aconteceu por conta de `seq :: a -> b -> b`, que faz a avaliação do segundo argumento implicar na avaliação do primeiro. Assim, quando o segundo argumento de `seq` precisa ser avaliado, a avaliação do primeiro é forçada. Apesar do que seu nome sugere, `seq` não garante que, uma vez que seja necessário avaliar o segundo argumento, o primeiro argumento seja avaliado antes (embora, na prática, isso costume acontecer). Vale ressaltar que `seq` não pode ser definida em Haskell, já que é implementada a baixo nívvel pelo compilador.

Utilizando `seq`, podemos criar um operador que representa a avaliação estrita em Haskell. A definição que segue deve ser escrita em `Functions.hs`.

```
($!) :: (a -> b) -> a -> b
f $! x = seq x (f x)

infixl 0 $!
```

Vamos testar o nosso novo operador.

```
*Functions Prelude> import Prelude (Int, (+))
*Functions Prelude> f _ = 0
*Functions Prelude> x = 1 + 1 :: Int
*Functions Prelude> :sprint x
x = _
*Functions Prelude> f x
0
*Functions Prelude> :sprint x
x = _
*Functions Prelude> f $ x
0
*Functions Prelude> :sprint x
x = _
*Functions Prelude> f $! x
0
*Functions Prelude> :sprint x
x = 2
```

Vemos que `f` recebe um argumento, mas o ignora e apenas retorna `0`. Assim, `f x` é avaliada sem que `x` deixe de estar suspenso. O mesmo pode ser dito de `f $ x`. Já `f $! x` usa `seq` para garantir que o argumento passado para `f` tem sua avaliação forçada assim que `f x` é requerido.

É necessário justificar a existência de um aplicador de função que representa avaliação estrita, ou seja, um operador que faz a avaliação do argumento ser forçada, tão logo seja

forçada a chamada da função. Em resumo, a preguiça de Haskell tem um custo de espaço. De forma mais elaborada, isso quer dizer que representar um valor suspenso demanda mais memória que avaliá-lo imediatamente. É fácil entender o porquê: enquanto suspenso, o valor é representado por um *thunk*, uma estrutura de dados que guarda a referência da função que o define, assim como referências para os argumentos que devem ser passados para essa função. Essas referências podem também estar apontando para valores suspensos (lembre-se que funções, inclusive, são valores como quaisquer outros), representados por outros *thunks*. No fim das contas, um valor suspenso pode acabar sendo representado por uma estrutura arbórea, construída a partir de *thunks* que apontam para novos *thunks*.

Um bom estudo de caso é `foldl`, que como bem sabemos é recursiva por cauda. Isso quer dizer que `foldl` executa em espaço constante, certo? Na pilha, sim, mas não na *heap*. Como é recursiva por cauda, `foldl` tem sua chamada corrente substituida por sua chamada recursiva, nunca ocupando mais que um *frame* na pilha. No entanto, a atualização do acumulador de `foldl` fica suspensa, afinal ele só poderia ser utilizado após `foldl` atingir seu caso base. Como os *thunks* são armazenados na *heap*, o acumulador final acaba sendo representado por uma cadeia de *thunks*, localizada na *heap*. Há uma surpresa desagradável, no entanto: quando o acumulador final precisa ser avaliado, a função registrada em seu *thunk* volta para a pilha, para ser avaliada com os devidos argumentos, e ao menos um desses (o valor anterior do acumulador) está suspenso e agora precisa ser avaliado. Isso implica que a cadeia de *thunks* que `foldl` produziu na *heap* volta para a pilha, convertida agora em uma pilha de chamadas de função. Precisamos consertar `foldl`. Escrevemos a seguinte definição em `List.hs`.

```
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' _ acc [] = acc
foldl' f acc (x:xs) = foldl' f $! (f acc x) $ xs
```

Temos o uso de `($!)` para forçar a avaliação de `f acc x` junto com a chamada de `foldl'`. Isso faz com que `foldl'` não construa uma cadeia de *thunks*, uma vez que as atualizações do acumulador são avaliadas imediatamente, já que `foldl'` é recursiva por cauda. Preferimos sempre o uso de `foldl'` a `foldl`.

Vamos lidar agora com tipos parametrizados. Tome o seguinte exemplo.

```
Prelude> x = 1 + 1 :: Int
Prelude> y = 2 + 2 :: Int
Prelude> p = (x, y)
Prelude> :sprint p
p = _
Prelude> :t fst
fst :: (a, b) -> a
Prelude> fst p
2
Prelude> :sprint p
p = (2,_)
```

Assim que definido, o par `p` fica suspenso. Ao utilizarmos `fst` (a assinatura deixa claro o que ela faz), a primeira componente de `p` é avaliada. Como não foi necessário avaliar a segunda componente, `p` é avaliado parcialmente. Tome este outro exemplo.

```
Prelude> x = 1 + 1 :: Int
Prelude> p = (x, x)
Prelude> :sprint p
p = _
Prelude> fst p
2
Prelude> :sprint p
p = (2,2)
```

Observa-se que, para avaliar a primeira componente de `p`, é necessário avaliar `x`. No entanto, como as componentes de `p` são ambas referências para o *thunk* de `x`, `p` acaba sendo avaliado por completo. Vamos brincar um pouco com listas.

```
*List> xs = map (+1) [1..10] :: [Int]
*List> :sprint xs
xs = _
*List> f _ = 0
*List> f xs
0
*List> :sprint xs
xs = _
*List> f $! xs
0
*List> :sprint xs
xs = _ : _
*List> length xs
10
*List> :sprint xs
xs = [_,_,_,_,_,_,_,_,_,_]
*List> sum xs
65
*List> :sprint xs
xs = [2,3,4,5,6,7,8,9,10,11]
```

Nota-se que `xs`, como esperado, não é avaliada assim que definida. Logo após `f $! xs`, `xs` é parcialmente avaliada. Como `f` ignora `xs`, sabemos que a forma como `xs` foi avaliada em `f $! xs` é fruto apenas de `seq`. Isso indica que, se o primeiro argumento de `seq` é um tipo parametrizado, ela não o avalia completamente, mas apenas evidencia sua estrutura.

Após `length xs`, percebemos que a estrutura de `xs` é avaliada por completo, uma vez que `length` precisa decompor seu argumento e, recursivamente, sua cauda. O conteúdo de `xs`, no entanto, é avaliado apenas depois de `sum xs`.

Agora, vamos tratar de `par :: a -> b -> b`, definida em `Control.Parallel`. Assim como `seq`, `par` é definida a baixo nível. A função `par` põe a avaliação de seu primeiro argumento para ocorrer paralelamente à avaliação de seu segundo, quando essa ocorrer. Vamos ilustrar seu uso.

```
*List> import Control.Parallel
*List Control.Parallel> import Prelude (seq)
*List Control.Parallel Prelude> sum' = foldl' (+) 0
*List Control.Parallel Prelude> x = sum' [1..10000000] :: Int
*List Control.Parallel Prelude> y = sum' [1..10000000] :: Int
*List Control.Parallel Prelude> :set +s
*List Control.Parallel Prelude> seq x y
50000005000000
(9.68 secs, 8,640,075,072 bytes)
*List Control.Parallel Prelude> x = sum' [1..10000000] :: Int
(0.00 secs, 0 bytes)
*List Control.Parallel Prelude> y = sum' [1..10000000] :: Int
(0.00 secs, 0 bytes)
*List Control.Parallel Prelude> par x y
50000005000000
(4.28 secs, 4,320,073,040 bytes)
```

No exemplo acima, `seq x y` força a avaliação de `x` assim que a de `y` é forçada. Como `y` é o retorno de `seq x y`, e esse é exibido no terminal, a avaliação de `x` precisa também ser feita, já que a de `y` é forçada. No caso de `par x y`, a avaliação de `x` ocorre simultaneamente a de `y`. Isso não é toda a explicação da redução de tempo apresentada.

É preciso esclarecer algo: se o processador que rodou o exemplo acima tivesse apenas um *core*, as avaliações de `x` e `y` seriam feitas em simultâneo, mas iriam concorrer pelo uso do único *core*, e daí não haveria redução de tempo. No entanto, o `ghc` é esperto o bastante para distribuir

o trabalho em paralelo de forma equilibrada entre os *cores*. Usamos o comando `lscpu` para saber as características de nosso processador.

```
$ lscpu
Arquitetura:                x86_64
Modo(s) operacional da CPU: 32-bit, 64-bit
Ordem dos bytes:            Little Endian
Tamanhos de endereco:       39 bits physical, 48 bits virtual
CPU(s):                     4
Lista de CPU(s) on-line:    0-3
Thread(s) per nucleo:       2
Nucleo(s) por soquete:      2
Soquete(s):                 1
...
```

Vamos entender a parte que nos importa da saída de `lscpu`: o sistema operacional "enxerga" quatro CPU's; um pouco abaixo, vemos que há dois *cores* no processador, cada um com duas *threads*. O sistema operacional nos diz que as quatro *threads* do processador são CPU's. No entanto, cada par de *threads* disputa os recursos de um mesmo *core*. Daí, o número de trabalhos que esse processador consegue executar verdadeiramente em paralelo não é o número de CPU's, mas o número de *cores*. Isso significa que, nesse processador, fazer uso de paralelismo nos traria um fator de redução de tempo limitado por 2. Chamamos esse fator de *speedup*.

No entanto, há tecnologias de certos processadores que podem distorcer a noção básica de *speedup*. Por exemplo, é possível estruturar um *core* de tal forma que as *threads* que o dividem, quando executando certos tipos de operação, concorram minimamente pelos recursos do *core*. Assim, o *speedup* poderia, na prática, ter um limite um pouco menor que o número de *threads*.

Junto de `par`, `Control.Parallel` também define `pseq :: a -> b -> b`. Além do comportamento herdado de `seq`, `pseq` assegura que a avaliação de seu primeiro argumento vai ocorrer necessariamente antes da avaliação de seu segundo. Essa diferença sutil entre `seq` e `pseq` permite um maior controle quando paralelizamos nosso código.

Para entender a necessidade de `pseq`, devemos estar cientes de que Haskell deixa em aberto a ordem de avaliação das partes de uma expressão. Isso é sensato, inclusive, já que uma expressão simples (sem a semântica de impureza) deve ter o mesmo valor, independente da ordem em que suas partes são avaliadas. Postergar a escolha de uma ordem pode abrir algumas oportunidades de otimizar o código. Uma preocupação a menos para o programador, certo? Em aplicações de execução serial, sim.

Tomemos um exemplo bem simples: `x + y`. Quando da avaliação de `x + y`, o compilador não nos dá garantias de quem será avaliado primeiro: `x` ou `y`. Para o resultado dessa soma, a ordem de avaliação de fato não importa. Mas e quanto a `par x (x + y)`?

Quando escreve-se `par x (x + y)`, a intenção é que `x` seja avaliado paralelamente a `x + y`, e assim poderia haver ganho de desempenho: caso a avaliação de `x + y` comece por `y`, temos ganho; caso comece por `x`, temos feito em vão a avaliação de `x` em paralelo, já que essa acaba sendo feita duas vezes. Assim, apenas o uso de `par` nos entrega o risco de gastar o tempo de um *core* com trabalho redundante.

Vamos analisar `par x (pseq y (x + y))`. Aqui, a avaliação de `x` é feita em paralelo com a avaliação de `pseq y (x + y)`, e essa força a avaliação de `y` a ocorrer antes da avaliação de `x + y`. Assim, temos que, quando `x + y` for avaliado, `y` já deve ter sido avaliada, assim como `x`, que deve ter sido avaliado em paralelo. Vemos, pois, que o uso de `pseq` para controlar a ordem de avaliação de expressões nos permite fazer avaliações em paralelo que não serão redundantes.

No caso de `x + y`, sabemos que `x` e `y` são tipos numéricos, o que indica que eles devem ser avaliados por completo. Mas quando se toma algo como `xs ++ ys`, em que `xs` e `ys` são duas listas, o uso de `seq` e `pseq` poderia forçar apenas uma avaliação superficial desses valores. Assim, uma simples combinação de `par` e `pseq` poderia apenas determinar, em paralelo, se `xs` é vazia ou tem cabeça e cauda. Isso faria quase nenhum trabalho ser feito em paralelo. Para remediar essa situação, vamos criar uma função que força a avaliação completa de uma lista. A definição a seguir deve ficar em `List.hs`.

```
force :: [a] -> [a]
force [] = []
force l@(x:xs) = seq x (seq (force xs) l)
```

Vamos entender o que `force` está fazendo. A chamada mais externa de `seq` garante que seu primeiro argumento (a cabeça de `l`) vai ser avaliada quando seu segundo argumento for avaliado. Esse segundo argumento é outra chamada de `seq`, e essa garante que `force xs` vai ser avaliada quando `l` for avaliada. Isso quer dizer que, quando `l` for avaliada, serão avaliadas tanto sua cabeça como sua cauda, e essa última de forma recursiva. Vale ressaltar que `force` não cria novas listas, uma vez que sempre retorna a mesma referência que recebe. Testamos o comportamento de `force`.

```
*List> xs = map (+1) [1..10] :: [Int]
*List> head xs
2
*List> :sprint xs
xs = 2 : _
*List> xs = map (+1) [1..10] :: [Int]
*List> ys = force xs
*List> head ys
2
*List> :sprint ys
ys = [2,3,4,5,6,7,8,9,10,11]
*List> :sprint xs
xs = [2,3,4,5,6,7,8,9,10,11]
```

Com `force` em mãos, estamos aptos a processar listas em paralelo, com a certeza de que essas serão avaliadas por completo. Com a próxima ferramenta, poderemos "quebrar" uma lista em "pedaços" iguais. Essa definição deve ficar em `List.hs`.

```
chunks :: Int -> [a] -> [[a]]
chunks _ [] = []
chunks k xs = ps : chunks k qs where
    (ps, qs) = splitAt k xs
```

A essa altura, a definição de `chunks` não deve ser difícil de entender. Está claro que `chunks k xs` retorna `xs` dividida em partes, todas (talvez não a última) com `k` elementos.

Usando `chunks` e `force`, podemos criar uma estratégia para paralelizar a avaliação de uma lista. Vamos supor que a lista `xs` tem muitos elementos, e todos eles são caros de avaliar. Definimos, inicialmente, `ys = concat $ chunks 10 xs`, supondo que avaliar um grupo de `10` elementos de `xs` é custoso o bastante. Claramente, `xs == ys`, mas `ys` oferece uma oportunidade evidente de paralelismo. Poderíamos fazer com que a avaliação de cada um dos "pedaços" de `xs` ocorresse em paralelo. A definição que segue deve ser escrita em `List.hs`.

```
parChunks :: Int -> [a] -> [[a]]
parChunks k = parallel . chunks k where
    parallel [] = []
    parallel l@(xs:xss) = par (force xs) (pseq (parallel xss) l)
```

Vamos explicar o que `parChunks` está fazendo. Primeiro, `chunks` "quebra" a lista em diversas partes, e em seguida `parallel` força a avaliação de cada uma delas em paralelo: a chamada de `par` faz com que a avaliação de `force xs` ocorra paralelamente à avaliação de `pseq (parallel xss) l`, e essa garante que a avaliação de `parallel xss` (que vai forçar os demais "pedaços" em paralelo) vai ser feita antes da avaliação de `l`.

Em vez de `ys = concat $ chunks 10 xs`, usamos `ys = concat $ parChunks 10 xs`, que força a avaliação de cada "pedaço" de `xs` em paralelo, antes de concatená-los. Esse `10` que utilizamos é apenas ilustrativo. É preciso fazer testes com diversos valores para esse parâmetro, e assim descobrir qual valor traz o maior ganho de tempo. Vamos aplicar essa ideia em um exemplo bem artificial.

```
module Main where

import Prelude (Int, (+), (^), IO, print)

import List

sum' :: [Int] -> Int
sum' = foldl' (+) 0

xs :: [Int]
xs = [sum' [1..10^9], sum' [2..10^9], sum' [3..10^9], sum' [4..10^9]]

ys :: [Int]
ys = concat (parChunks 1 xs)

main :: IO ()
main = print ys
```

Como `xs` tem quatro elementos, `parChunks 1 xs` cria quatro trabalhos paralelos para avaliar `xs`. Criamos um arquivo `Test.hs` com essas definições. Vamos compilar esse arquivo com o seguinte comando.

```
$ ghc -O2 -threaded Test -W
```

Esse comando faz o compilador gerar código muito otimizado (`-O2`), e acoplar um *runtime system* (`-threaded`), que gerencia os trabalhos feitos em paralelo. Agora, realizamos alguns testes com o executável criado.

```
$ time ./Test +RTS -N1
[500000000500000000,500000000499999999,500000000499999997,500000000499999994]

real    0m39,523s
user    0m38,978s
sys     0m0,530s
```

Primeiro, `time` mede o tempo de execução do comando que vem em seguida. O tempo `real` é o tempo real de execução do programa, enquanto `user` e `sys` indicam, respectivamente, quanto tempo o processo gastou rodando código de usuário e código do *kernel*.

A chamada do executável `Test` apresenta o parâmetro `+RTS`, que indica que os parâmetros seguintes dizem respeito ao *runtime system*. O único parâmetro que vem depois de `+RTS` é `-N1`, que instrui o *runtime system* a distribuir todos os trabalhos em uma *thread*. Assim, mesmo fazendo uso de paralelismo, `-N1` não nos permite ter ganhos, pois mesmo as avaliações paralelizadas são realizadas todas pela mesma *thread*, e portanto não ocorrem de forma simultânea. Vamos tentar `-N2`.

```
$ time ./Teste +RTS -N2
[500000000500000000,500000000499999999,500000000499999997,500000000499999994]

real    0m33,065s
user    0m44,820s
sys     0m1,494s
```

A primeira coisa que notamos é uma redução de tempo bem pequena, de $39,523$ para $33,065$ segundos, resultando em um *speedup* de quase $1,2$. O *speedup* aqui obtido passou longe de 2, e para o nosso processador, esse deve ser o limite teórico de *speedup*. Não adianta testar os parâmetros de `-N3` em diante, pois pelo menos duas *threads* concorreriam pelos recursos de um mesmo *core*.

Há uma observação interessante a ser feita sobre os tempos `user` e `sys`. No primeiro teste, a soma de `user` e `sys` é bastante próxima de `real`. No segundo teste, porém, essa soma

ultrapassa `real`. Isso ocorre porque `user` e `sys` acumulam os devidos tempos de cada uma das *threads*.

Vamos fazer uma modificação em `Test.hs`. Substituimos a expressão `parChunks 1 xs` por `parChunks 2 xs`, fazendo `xs` ser avaliada, dessa vez, por dois trabalhos paralelos. Recompilamos `Test.hs` e voltamos aos testes.

```
$ time ./Teste +RTS -N1
[500000000500000000,500000000499999999,500000000499999997,500000000499999994]

real    0m39,544s
user    0m39,056s
sys     0m0,470s
```

Esse novo teste nos tomou quase o mesmo tempo de execução que o primeiro teste. Vamos testar com `-N2`.

```
 time ./Teste +RTS -N2
[500000000500000000,500000000499999999,500000000499999997,500000000499999994]

real    0m21,273s
user    0m41,760s
sys     0m0,646s
```

Agora sim! A redução de tempo foi de 39,544 para 21,273 segundos, o que nos dá um *speedup* um pouco maior que $1,85$. Esse *speedup* está bem mais próximo do limite teórico de 2. Novamente, não faz sentido usar mais *threads* do que o número de *cores*. Com esses dois grupos de experimentos, observamos o impacto que o parâmetro de `parChunks` tem sobre o *speedup* que podemos atingir.

Nosso pequeno exemplo artificial pode deixar a impressão de que a escolha do parâmetro de `parChunks` deve ser tal que haja exatamente um "pedaço" da lista para cada *core* avaliar. Embora plausível, essa escolha não nos dá escalabilidade. Isso quer dizer que, se prepararmos um programa para tirar proveito de um certo número de *cores*, ele precisará ser modificado para executar eficientemente em outro número de *cores*.

Quanto maiores os "pedaços" em que partimos a lista, maiores serão os trabalhos a serem realizados em paralelo. Para haver um bom ganho de tempo com paralelismo, é necessário que os trabalhos paralelizados sejam custosos a ponto de compensar o *overhead* introduzido pelo *runtime system*. O parâmetro ideal de `parChunks` varia conforme a natureza das tarefas que se está paralelizando. Geralmente, descobrir esse valor ideal requer alguma experimentação.

4 Laboratórios

Aqui se encontram as atividades a serem desenvolvidas durante as aulas práticas. Cada subseção corresponde a uma aula em laboratório.

4.1 Laboratório 1

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab1.hs`. Seu conteúdo inicial deve ser o seguinte.

```
module Lab1 where

import Prelude (Eq, (+), (*), Int)
import List(foldl, foldr)
```

1. Reescreva as funções `length`, `(++)`, `concat` e `reverse` como chamadas de `foldl` ou `foldr` (a que parecer mais adequada). Transcreva essas novas definições para `List.hs`.
2. Escreva a função `squares` que, dada uma lista de inteiros, retorna uma lista contendo o quadrado desses inteiros.
3. Escreva a função `count :: Eq a => a -> [a] -> Int`, que retorna o número de ocorrências de um elemento em uma lista. Além de deixá-la em `Lab1.hs`, copie essa função para `List.hs`.

4.2 Laboratório 2

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab2.hs`. Seu conteúdo inicial deve ser o seguinte.

```
module Lab2 where

import Prelude(Eq, Fractional, (/))

import Bool
import Functions
import List
import Maybe
```

1. Escreva a função `repeat :: a -> [a]` que, dado um elemento `x` de tipo `a`, retorna uma lista infinita cujos elementos são todos iguais a `x`.
2. Escreva a função `cycle :: [a] -> [a]` que, dada uma lista `xs` como entrada, retorna uma lista infinita formada por repetir continuamente os elementos de `xs`, em ordem. Desejamos que `cycle [] == []`.
3. Escreva a função `intercalate :: a -> [a] -> [a]` que, dado um elemento `x` e uma lista `ys`, retorna uma lista que consiste dos elementos de `ys` intercalados por `x`. Desejamos que `intercalate _ [] == []` e `intercalate _ [y] == [y]`, pois nesses casos `ys` não tem elementos a serem intercalados por `x`.
4. Escreva a função `safeDiv :: (Eq a, Fractional a) => a -> a -> Maybe a` que faz divisões de forma segura, isto é, representa a divisão por zero como uma falha.
5. Escreva uma implementação para `find :: (a -> Bool) -> [a] -> Maybe a`. Perceba que é possível implementar `find` utilizando composições de `dropWhile`, `safeHead` e `not`. Essa é apenas uma sugestão. Implemente como achar mais adequado.

4.3 Laboratório 3

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab3.hs`. Seu conteúdo inicial deve ser o seguinte.

```
module Lab3 where

import Prelude (Ord, (<=), Eq, (==), Bool(True, False))

import Bool
import Functions
import List
```

1. Implemente a função `group :: Eq a => [a] -> [[a]]`, que recebe uma lista `xs` e retorna uma lista de listas, onde cada sublista é uma subsequência contínua maximal de elementos iguais de `xs`. Por exemplo, `group [1, 1, 2, 2, 2, 3, 3, 1, 1, 1] == [[1, 1], [2, 2, 2], [3, 3], [1, 1, 1]]`. A função `span` pode ser de grande ajuda aqui.

2. Agora, generalizamos a função `group` como `groupBy`, fazendo com que `group` seja equivalente a `groupBy (==)`. Implemente `groupBy :: (a -> a -> Bool) -> [a] -> [[a]]`.

3. Implemente as funções `all` e `any`, ambas com assinatura `(a -> Bool) -> [a] -> Bool`, que decidem, respectivamente, se todos ou algum elemento de uma lista satisfazem o predicado dado como entrada. Perceba que elas podem ser dadas como composições de `and`, `or` e `map`.

4. Escreva a função `merge :: Ord a => [a] -> [a] -> [a]` que, dadas duas listas ordenadas como entrada, retorna uma lista ordenada com todos os elementos das listas recebidas.

5. Escreva a função `split :: [a] -> ([a], [a])` que, dada uma lista `xs` como entrada, retorna duas listas: a primeira contém os elementos das posições ímpares de `xs`, e a segunda os elementos das posições pares de `xs`. Aqui, estamos admitindo que a cabeça de uma lista está na posição 1.

6. Agora, utilizando `merge` e `split`, implementadas anteriormente, desenvolva a função `mergesort :: Ord a => [a] -> [a]`.

4.4 Laboratório 4

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab4.hs`. Seu conteúdo inicial deve ser o seguinte.

```
module Main where

import Prelude (Eq, (==), (/=),
                Int, (+), Char,
                IO, readFile, putStrLn, show,
                (>>=), pure)

import Bool
import Functions
import List

import qualified BSTree as BST
```

Para esse laboratório, vamos utilizar o arquivo auxiliar `heyjude.txt`, que contém a letra da referida música. Como você deve ter notado, o arquivo `Lab4.hs` contém um módulo chamado

`Main`. Isso indica que vamos criar um executável `Lab4`. Para informações sobre como compilar um programa em Haskell, veja a Subseção 3.7.

As atividades a seguir devem ser um guia para o desenvolvimento de um programa que imprime a contagem da ocorrência de palavras no arquivo `heyjude.txt`.

1. Escreva a função `remove :: Eq a => a -> [a] -> [a]`, que remove todas as ocorrências de um certo elemento em uma lista. Perceba que `remove` pode ser implementada como uma chamada de `filter`.

2. Escreva a função `split :: Eq a => a -> [a] -> [[a]]` que, dados um elemento `x` e uma lista `ys`, retorna as sublistas de `ys` entre as ocorrências de `x`. Por exemplo, `split 0 [0, 1, 2, 0, 2, 5, 7, 0, 0, 0, 3, 0] == [[1, 2], [2, 5, 7], [3]]`. As funções `span` e `dropWhile` são bastante úteis aqui.

3. Utilize a função `split` para escrever as funções `lines` e `words`, ambas com assinatura `[Char] -> [[Char]]`. A função `lines` (`words`) deve particionar uma string em substrings que representem suas linhas (palavras). Lembre-se que linhas são separadas por `'\n'` e palavras por `' '`.

4. Escreva a função `count :: [[Char]] -> BST.BSTree [Char] Int` que, dada uma lista de strings, retorna uma árvore binária de busca onde as chaves são essas strings e cada uma é valorada pelo número de vezes que ocorre na lista. As funções `foldr`, `BST.empty`, `BST.contains`, `BST.insert` e `BST.update` são muito úteis para isso.

5. Vamos desenvolver a função `process :: [Char] -> [Char]`, que junta todas as partes do nosso programa. Essencialmente, `process` vai limpar a string de entrada, removendo caracteres indesejáveis, contar a ocorrência das palavras e em seguida preparar a saída do programa, que é seu retorno. Podemos decompor `process` nessas três partes.

```
process :: [Char] -> [Char]
process = makeOutput . countWords . clean where
        clean = undefined
        countWords = undefined
        makeOutput = concat . intercalate "\n" . map f . BST.inOrder
        f (str, c) = str ++ ":␣" ++ show c
```

   Escreva implementações para `clean` e `countWords`. A função `clean` deve remover os caracteres `','`, `'('`, `')'`, `'?'` e `'!'`, e isso pode ser feito com uma composição de `remove`s. A função `countWords` deve construir a árvore com a contagem de ocorrência das palavras, e isso pode ser expresso como uma composição das funções `lines`, `words`, `map`, `concat` e `count`.

6. Com todas as funções desenvolvidas, estamos aptos a finalizar nosso programa. Nossa `main` (que não pode ser totalmente explicada agora) é apresentada a seguir.

```
main :: IO ()
main = readFile "heyjude.txt" >>= pure . process >>= putStrLn
```

   Agora, basta compilar `Lab4.hs` e conferir a saída do executável.

4.5 Laboratório 5

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab5.hs`. Seu conteúdo inicial deve ser o seguinte.

```
module Main where

import Prelude (Eq, (==), (/=),
                Int, (+), Char,
                IO, readFile, putStrLn, show,
                (>>=), pure)

import Bool
import Functions
import List

import qualified BSTree as BST
```

Vamos criar um programa ligeiramente diferente do criado na Subseção 4.4. Em vez de imprimir as palavras em ordem alfabética, vamos imprimí-las em ordem decrescente de frequência, isto é, as palavras com maior número de ocorrências serão impressas primeiro.

Para tanto, é preciso criar uma função auxiliar `sortBy`, que ordena uma lista de acordo com um comparador, também dado como entrada. Basta generalizar `mergesort`, proposta na Subseção 4.3.

Descobrindo-se o comparador adequado para produzir a ordenação desejada, uma simples alteração no programa da Subseção 4.4 é suficiente para concluir esse laboratório. Com isso, esperamos evidenciar o reuso obtido quando expressamos programas funcionais por meio de composições.

Esta subseção se encontra escrita em um estilo distinto das anteriores por uma razão. Aqui, queremos também avaliar a capacidade de conceber assinaturas e tipos para as funções, em vez de apenas criá-las a partir de assinaturas pré-determinadas.

4.6 Laboratório 6

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab6.hs`. Seu conteúdo inicial deve ser o seguinte.

```
module Main where

import Prelude (Char, Bool(False), pure, (>>=),
                IO, readFile, putStrLn, read, show)

import Bool
import Functions
import List
import Maybe

import qualified Graph as G
```

Vamos trabalhar com grafos. Em particular, vamos desenvolver um programa que resolve o Problema da Celebridade, um clássico das maratonas de programação.

Dado um grafo $G = (V, E)$, dizemos que um vértice $v \in V$ é uma celebridade se todo outro vértice "conhece" v e v "conhece" ninguém. Perceba que um grafo tem no máximo uma celebridade. Aqui, estamos usando "u conhece v" para indicar a existência da aresta $(u, v) \in E$.

Para começar, nossa `main` será como nos outros laboratórios.

```
main :: IO ()
main = readFile "g1.txt" >>= pure . process >>= putStrLn
```

No entanto, o arquivo `g1.txt` contém a codificação de um grafo, em vez de um conteúdo arbitrário. Sua primeira linha contém um inteiro, indicando o número de vértices. As demais linhas contêm exatamente dois inteiros, indicando as extremidades das arestas.

Nossa função `process` tem quatro etapas: dividir o conteúdo do arquivo por linhas, construir o grafo correspondente, resolver o Problema da Celebridade e produzir a saída.

```
process :: [Char] -> [Char]
process = makeOutput . hasCelebrity . buildGraph . lines
```

Para que possamos nos concentrar no problema em si, eis a definição de `buildGraph`.

```
buildGraph :: [[Char]] -> G.Graph
buildGraph [] = G.empty
buildGraph (n:es) = foldr f gn es' where
        es' = map words es
        gn = G.edgeless (read n)
        f e g = G.addEdge g u v where
                (u:v:_) = map read e
```

Note que, nesse contexto, `read` consegue inferir o tipo de seu retorno. Isso é possível por conta das assinaturas das funções que estão em `Graph.hs`.

Em um primeiro momento, podemos supor que `makeOutput = show`. Assim, o retorno booleano de `hasCelebrity` é impresso na tela, indicando se o grafo tem uma celebridade. Obviamente, é mais útil dizer qual é a celebridade, caso haja alguma. No entanto, esse improviso nos deixa apenas com `hasCelebrity` indefinida.

```
hasCelebrity :: G.Graph -> Bool
hasCelebrity g = verify candidate where
        verify = undefined
        candidate = undefined
```

Vamos utilizar uma solução bem simples para o problema. Como um grafo tem no máximo uma celebridade, em um primeiro momento queremos encontrar um vértice candidato a celebridade. Isso pode ser feito com uma pilha, e as listas cumprem esse papel. Colocamos todos os vértices em uma pilha. Se há pelo menos dois vértices na pilha, digamos u e v, verificamos se a aresta (u, v) existe: caso exista, u "conhece" alguém e não pode ser uma celebridade; caso contrário, v não é "conhecido" por um outro vértice e não pode ser celebridade. Em ambos os casos, eliminamos um vértice e o outro volta para a pilha. Quando tivermos um único vértice na pilha, esse será nosso candidato. Precisamos retornar o candidato dentro de um `Maybe`, para podermos tratar o caso da pilha (lista) vazia.

Uma vez determinado o candidato, caso haja algum, a função `verify` testa as condições de celebridade para ele.

```
verify = maybe False f where
        f c = and [cond1, not cond2] where
        cond1 = undefined
        cond2 = undefined
```

Nessa definição, `maybe` nos poupa de tratar explicitamente os casos de `Maybe`. O valor `cond1` indica se todos os outros vértices "conhecem" o candidato, enquanto `cond2` indica se o candidato "conhece" algum outro vértice. As funções `and`, `or`, `zipWith`, `G.hasEdge`, `repeat` e `remove` são úteis para definí-los.

Aqui, o executável já deve estar pronto. Teste-o, modificando o conteúdo de `g1.txt` para isso. Uma vez seguro do comportamento do programa, altere-o para que a celebridade (caso haja uma) seja exibida como saída.

4.7 Laboratório 7

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab7.hs`. Antes de abordar nosso objetivo primário, vamos criar uma variante de `Heap.hs`.

Baseado na estrutura definida em `Heap.hs`, crie uma estrutura para representar uma fila de prioridade máxima. Copie o conteúdo de `Heap.hs` para `MaxHeap.hs`. Além de renomear o módulo do arquivo adequadamente, note que apenas uma função de `MaxHeap.hs` precisa sofrer alterações.

Agora que temos implementações para filas de prioridade mínima e máxima, vamos aproveitá-las e criar uma fila de prioridade mediana. A finalidade de tal estrutura de dados é permitir acesso em tempo constante à mediana de um conjunto de elementos, mesmo que esse conjunto sofra alterações. Por simplicidade, consideramos que, dada uma lista `xs` de n elementos, sua mediana é o $\lceil \frac{n}{2} \rceil$-ésimo elemento de uma versão ordenada de `xs`.

Dada uma lista inicial `xs` de n elementos do tipo `(k, v)`, com `k` ordenável, vamos criar um algoritmo `median :: Ord k => [(k, v)] -> Maybe (k, v)`, que encontra a mediana de `xs` (de acordo com `k`) em tempo linear, no caso médio. O algoritmo que `median` implementa é o *quickselect*, inspirado nas mesmas ideias do *quicksort*. Para implementá-lo, `filter` é bastante útil.

Uma vez determinada a mediana de `xs`, digamos `m`, dividimos `xs` em duas listas: a dos elementos menores ou iguais a `m` (`ps`) e a dos elementos maiores que `m` (`qs`). Por fim, criamos uma fila de prioridade máxima (mínima) com os elementos de `ps` (`qs`).

Após isso tudo, obtemos duas filas de prioridade: a máxima agrupa os elementos com prioridade menor ou igual a da mediana; a mínima agrupa os elementos com prioridade superior a da mediana. Perceba que a diferença entre as quantidades de elementos em cada fila de prioridade é no máximo um: se n é par, as duas *heaps* têm o mesmo número de elementos; se n é ímpar, a *heap* máxima tem exatamente um elemento a mais que a *heap* mínima.

A ideia principal dessa estrutura de dados está no fato de que, ao inserir ou remover exatamente um elemento do conjunto representado, ou o "sucessor" ou o "antecessor" da mediana torna-se a nova mediana. O "antecessor" ("sucessor") é indicado como o elemento mais prioritário da *heap* máxima (mínima).

Quando queremos tomar a mediana, escolhemos sempre o elemento mais prioritário da *heap* com mais elementos. Em caso de empate, preferimos o elemento mais prioritário da *heap* máxima. Aqui, fica claro que é preciso armazenar, além das *heaps*, a quantidade de elementos de cada uma.

Ao inserirmos um elemento `x`, fazemos a inserção de acordo com a relação entre `x` e a mediana atual `m`: se `x` tem mais prioridade que `m`, inserimos `x` na *heap* mínima; do contrário, inserimos `x` na *heap* máxima.

Sempre que é feita uma alteração na estrutura de dados, é preciso rebalancear as *heaps* caso uma tenha pelo menos dois elementos a mais que a outra. Isso é feito removendo o elemento mais prioritário da *heap* com mais elementos, e inserindo-o na outra *heap*. A quantidade de vezes que essa operação deve ser feita pode ser controlada com o uso de recursão.

É esperado que sejam implementadas as operações `fromList`, `lookup`, `insert` e `pop` para essa estrutura de dados. Observe que `fromList` falha caso receba uma lista vazia. Além dessas, a função auxiliar `rebalance` deve ser chamada sempre que uma alteração for feita.

Com a estrutura implementada, escreva um programa para testá-la. Podemos trabalhar com elementos de tipo `(Int, ())`.

Uma ideia seria receber um arquivo com duas linhas, cada uma contendo inteiros separados por um espaço. A estrutura inicial é construída a partir das chaves na primeira linha. Após isso, os elementos da segunda linha são inseridos um por vez. A cada modificação da estrutura, a mediana atual deve ser impressa na tela.

4.8 Laboratório 8

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab8.hs`. Por razões que ficarão claras no decorrer desse texto, esperamos receber, como resultado desse laboratório, os arquivos `Lab4.hs` e `Lab8.hs`. O conteúdo inicial de `Lab8.hs` deve ser o seguinte.

```
module Main where

import Prelude (IO, print, putStrLn,
                readFile, writeFile,
                getContents,
                (>>=))

import System.Environment (getArgs)

import Functions
import List
```

Nesse laboratório, vamos exercitar uma interação básica com o sistema operacional, que consiste em tratar argumentos da linha de comando. Vamos desenvolver uma série de programas elementares, começando com `program1`.

```
program1 :: IO ()
program1 = getArgs >>= print
```

Basicamente, `program1` toma a lista de argumentos e a imprime no terminal. Para compilar, acrescentamos.

```
main = program1
```

Não escrevemos a assinatura explícita de `main`, pois ela tem claramente a mesma assinatura de `program1`. Tente passar argumentos para o executável `Lab8`. Agora, vamos desenvolver um análogo do comando `cat`, que imprime o conteúdo de um arquivo de texto no terminal.

```
program2 :: IO ()
program2 = getArgs >>= readFile . head >>= putStrLn
```

Com o uso de `head`, é bem simples tratar um único argumento passado pela linha de comando. Faça a alteração devida em `main` e teste o novo programa. Observe que ele ignora todos, menos o primeiro argumento para ele passado. Note também que ele lança exceções caso não receba argumentos (por conta de `head`), ou caso seu primeiro argumento não seja um arquivo válido (por conta de `readFile`). Isso indica que deve haver um jeito melhor de lidar com argumentos.

Como sua primeira tarefa, escreva `program3`, que imprime o conteúdo de um arquivo de texto passado como argumento, mas antes ordena suas linhas. As funções `lines`, `sortBy`, `intercalate` e `concat` são úteis para isso. Organizá-las por composição pode ser uma boa ideia.

Agora, vamos criar um programa que copia o conteúdo de um arquivo de texto.

```
program4 :: IO ()
program4 = getArgs >>= treatArgs where
        treatArgs [from, to] = readFile from >>= writeFile to
        treatArgs _ = putStrLn "You␣must␣pass␣exactly␣two␣arguments."
```

Verifique a assinatura de `writeFile` no `ghci`, afinal é a primeira vez que a vemos. A função `treatArgs` trata os argumentos usando dois casos: no primeiro, fica claro que esperamos dois argumentos, para ler o conteúdo do arquivo `from` e escrevê-lo no arquivo `to`; o segundo caso lida com qualquer outro número de argumentos, informando o usuário sobre o uso correto do programa. Tente copiar o conteúdo de `heyjude.txt` para `teste.txt`. Após isso, tente passar diversos números de argumentos. Como `writeFile` apaga o conteúdo do arquivo antes de escrever seu segundo argumento, não é interessante que `to` seja o nome de um arquivo existente.

Vamos voltar ao `program2`. Dissemos que `program2` é análogo ao comando `cat`. No entanto, `cat` e `program2` se comportam de formas distintas quando não recebem argumentos.

Após verificar isso, sua segunda tarefa é escrever `program5`, que deve se comportar exatamente igual a `cat`, seja com zero ou um argumentos. Já devemos ter notado que `cat` lê sua entrada do teclado quando não tem argumentos. Em Haskell, `getContents` faz exatamente isso, isto é, retorna uma string "infinita" de coisas que são digitadas no teclado. Para esse programa, `treatArgs` deve tratar os casos de zero e um argumentos, e invalidar todos os demais casos.

Na nossa terceira tarefa, vamos modificar o conteúdo de `Lab4.hs`, que foi desenvolvido na Subseção 4.4. Em vez de tratar apenas o conteúdo do arquivo `heyjude.txt`, nosso programa modificado deve receber exatamente um argumento, que é o nome do arquivo que deve ser tratado.

Por fim, a quarta tarefa consiste em desenvolver `program6`, que recebe exatamente dois argumentos: uma string `str` e o nome de um arquivo `file`. Esse programa deve imprimir apenas as linhas de `file` que contém `str` como substring. A função `isInfixOf`, proposta no Exercício 2, pode ser utilizada para essa filtragem. Feito isso, acrescente um caso em que `program6` recebe exatamente um argumento, e imprime apenas as linhas digitadas no teclado que tenham `str` como substring. Temos implementado uma versão (muito) simplificada do comando `grep`. Inclusive, podemos usar `program6` em combinação com o *pipe* do sistema operacional.

4.9 Laboratório 9

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab9.hs`. O conteúdo inicial de `Lab9.hs` deve ser o seguinte.

```
module Main where

import Prelude (IO, putStrLn, getLine,
                (<$>), (<*>), pure,
                (*>), (>>=),
                Char, (==))

import Bool
import Functions
import List
```

Vamos desenvolver um programa que "cadastra" uma senha. Por motivos de verificação, o programa pede que ela seja digitada duas vezes e, enquanto houver divergência, pede novamente.

A primeira coisa de que precisamos é uma função `ask :: [Char] -> IO [Char]`, que imprime seu argumento na tela e retorna uma linha lida a partir do teclado, dentro de `IO`. Para construir `ask`, as funções `putStrLn`, `getLine` e `(*>)` são úteis.

Com `ask` em mãos, vamos construir `getPasswd :: IO ([Char], [Char])`, uma ação que retorna um par de strings em `IO`. A ideia aqui não é tão complicada: precisamos construir um par, mas dentro de `IO`. O operador `(,)` constrói pares simples, e podemos aproveitá-lo para construir um par em `IO`, já que `IO` é instância de `Functor` e `Applicative`. Sabemos que `(,) x y` constrói um par de tipo `(a, b)`, desde que `x :: a` e `y :: b`. Caso `x :: IO a` e `y :: IO b`, como poderíamos usar `(,)`, `(<$>)` e `(<*>)` para construir `IO (a, b)`? Respondida a pergunta, basta fazer com que `x` e `y` sejam duas chamadas de `ask`, com strings que avisem o usuário se é a primeira (`x`) ou segunda (`y`) vez que estão digitando a senha.

Temos implementado `getPasswd`, mas ainda precisamos validar se as duas strings digitadas são iguais. Fazemos isso em `validPasswd`, uma ação recursiva que termina apenas ao receber duas strings idênticas. Sua implementação é dada a seguir.

```
validPasswd :: IO [Char]
validPasswd = getPasswd >>= f where
    f (p1, p2) = cond (p1 == p2)
                      (pure p1)
                      (incorrect *> validPasswd)
    incorrect = putStrLn "They␣don't␣match!␣Try␣again."
```

Agora que sabemos como obter uma senha válida, implemente `main :: IO ()` e teste seu programa. Mas o laboratório ainda não acabou.

Em `Data.Char`, existem alguns predicados muito úteis, que testam se um `Char` é um certo tipo de caractere, como letra minúscula, maiúscula ou dígito. Dizemos que uma senha é forte se ela tem pelo menos oito caracteres e há, dentre eles, uma letra minúscula, uma maiúscula e um dígito. Escreva uma função `isStrong :: [Char] -> Bool`, que determina se uma senha é forte.

Uma vez implementada `isStrong`, escreva a ação `strongPasswd :: IO [Char]`, que utiliza `validPasswd` para obter senhas válidas e retorna, dentro de `IO`, a primeira que seja forte. A definição de `validPasswd` é um bom modelo para `strongPasswd`.

Adapte sua `main` e teste seu programa. Agora sim o laboratório acabou.

4.10 Laboratório 10

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab10.hs`. O conteúdo inicial de `Lab10.hs` deve ser o seguinte.

```
module Main where

import Prelude (Ord, (<=), (>=), (-), Bool,
                IO, putStrLn, (<$>), pure, (<*>))

import Bool
import Functions
import List
import Maybe
```

Neste laboratório, vamos resolver o problema da maior subsequência crescente, ou *longest increasing subsequence*. Dada uma sequência de naturais, gostaríamos de encontrar a maior subsequência crescente dessa sequência. Por exemplo, dada a sequência $10, 22, 9, 33, 21, 50, 41, 60, 80$, sua maior subsequência crescente é $10, 22, 33, 41, 60, 80$.

Primeiro, desenvolva `maximum :: Ord a => [a] -> Maybe a`, que retorna o maior elemento de uma lista, caso ela não seja vazia. Basta usar `foldl` e `(<=)`. No entanto, precisaremos fazer uso de uma versão genérica de `maximum`. Para tanto, implemente a função `maximalBy :: (a -> a -> Bool) -> [a] -> Maybe a`, de tal forma que `maximalBy (>=)` seja equivalente a `maximum`.

Agora, vamos criar uma solução ingênua para o problema em vista. Dada uma sequência de n naturais $x_0, x_1, \ldots, x_{n-1}$, criamos a seguinte notação: s_i representa a maior subsequência crescente começando com x_i, para $0 \leq i \leq n-1$.

Podemos definir s em termos de si mesma: claramente, s_{n-1} é uma sequência unitária contendo x_{n-1}; para $0 \leq i \leq n-2$, s_i começa com x_i, e pode seguir com os elementos de $s_j, i < j \leq n-1$, desde que $x_i \leq x_j$. Certamente, definimos s_i utilizando, dentre todos os s_j possíveis, aquele que represente a maior subsequência. Por fim, é fácil escolher a maior subsequência crescente dentre os $s_i, 0 \leq i \leq k-1$. Essa solução pode ser implementada da seguinte forma.

```
lis :: Ord a => [a] -> [a]
lis [] = []
lis [x] = [x]
lis xs = fromMaybe [] (maximalBy cmp (s <$> [0..n - 1] <*> pure xs)) where
        n = length xs
        cmp ps qs = length ps >= length qs
        s 0 [] = []
        s 0 [y] = [y]
        s 0 (y:ys) = y : ms where
                candidates = filter ((y<=) . head) $ s <$> [0..k - 1] <*> pure ys
                k = length ys
                ms = fromMaybe [] $ maximalBy cmp candidates
        s i ys = s 0 $ drop i ys
```

Em um primeiro momento, a função `s` parece receber mais argumentos que o necessário, afinal s é indexada apenas por $0 \leq i \leq n-1$. No entanto, receber a lista como segundo argumento nos permite tratar o caso referente a s_i, para algum $0 < i \leq n-1$, com `drop`.

Uma outra coisa interessante de se notar é `candidates`. Note que `candidates` parece estar fazendo `s` tratar sobre subsequências de outras listas além de `xs`. Na verdade não está, mas da forma como `candidates` se comporta, os índices passados para `s` podem referir-se a posições em sufixos de `xs`, e assim não podem ser interpretados de maneira absoluta, como os índices de s.

Bem, a implementação de `lis` apresentada está correta, mas é extremamente ineficiente. É preciso utilizar `memo` e `fix` para criar uma *cache* para `s`. No entanto, há um obstáculo: `memo` espera receber uma função `Int -> b`, e essa não é a assinatura de `s`.

O objetivo final desse laboratório é reescrever `lis` de forma que `s` tenha assinatura `Int -> [a]`, para em seguida criar uma *cache* para `s`.

4.11 Laboratório 11

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab11.hs`. O conteúdo inicial de `Lab11.hs` deve ser o seguinte.

```
module Main where

import Prelude (Char, Int, (==),
                IO, putStrLn, (<$>))

import Bool
import Functions
import List
```

Neste laboratório, vamos construir definições para trabalhar com autômatos finitos não-determinísticos. Para isso, vamos usar o seguinte tipo.

```
type State = Int
type Transition = (State, Char, State)

data NFA = NFA [State] State [Transition] [State]
```

Em `NFA`, o primeiro campo representa o conjunto de estados do autômato, enquanto o segundo representa seu estado corrente. Isso difere um pouco da definição usual de autômato, mas facilita a implementação. Já o terceiro campo é a lista de transições do autômato, enquanto o quarto representa seu conjunto de estados finais. Além disso, definimos uma série de funções auxiliares para esse tipo.

```
nfa :: [State] -> State -> [Transition] -> [State] -> NFA
nfa states s trs finals = NFA states s trs finals

states :: NFA -> [State]
states (NFA states _ _ _) = states

currentState :: NFA -> State
currentState (NFA _ s _ _) = s

transitions :: NFA -> [Transition]
transitions (NFA _ _ trs _) = trs

finalStates :: NFA -> [State]
finalStates (NFA _ _ _ finals) = finals
```

Com elas, podemos definir uma função que faz o autômato "consumir" um caractere. Em nossa representação desse conceito, o autômato não vai mudar de estado, e sim produzir novos autômatos, cada um deles representando o resultado de uma possível transição.

```
transition :: NFA -> Char -> [NFA]
transition aut c = f <$> nextStates where
  validTransitions = filter valid (transitions aut)
  valid (st, c', _) = and [st == (currentState aut), c' == c]
  nextStates = thd <$> validTransitions
  thd (_, _, z) = z
  f st = nfa (states aut) st (transitions aut) (finalStates aut)
```

Para prosseguir com as atividades do laboratório, faça a implementação de duas funções: `isAccepting :: NFA -> Bool` e `run :: NFA -> [Char] -> Bool`. Enquanto a primeira função deixa claro o seu propósito, a segunda deve fazer um autômato consumir uma lista de caracteres, e no fim determinar se essa foi aceita pelo autômato. Para facilitar a escrita, recomendamos a implementação de `foldM`, proposta no Exercício 45. Como a lista representa não-determinismo enquanto mônada, `run` pode ser escrita em termos de `foldM`. Observe bem a assinatura de `transition`.

Por fim, crie dois autômatos, utilizando o arcabouço aqui desenvolvido. O primeiro autômato deve aceitar representações binárias de números pares, enquanto o segundo aceita strings binárias que tenham 010 como substring. Perceba que, em ambos os casos, quaisquer outros valores de `Char` devem implicar na rejeição da string.

4.12 Laboratório 12

Desenvolva as atividades a seguir. Salve suas funções em um arquivo `Lab12.hs`. O conteúdo inicial de `Lab12.hs` deve ser o seguinte.

```
module Main where

import Prelude (Int, (+), (^),
                IO, (>>=),
                print, read)

import System.Environment (getArgs)

import Functions
import List

sum' :: [Int] -> Int
sum' = foldl' (+) 0

main :: IO ()
main = getArgs >>= \[kstr] ->
       print $ sum' (xs $ read kstr)

xs :: Int -> [Int]
xs k = zipWith (+) [1..10^k] [1..10^k]
```

Além do código que deve ser desenvolvido, é esperada a entrega de uma planilha contendo os tempos de execução do programa, de acordo com dois parâmetros: o tamanho da entrada, dado por `k`; o argumento passado para `parChunks` (o tamanho dos "pedaços" em que se divide a lista).

Compile esse programa com `ghc -O2 -threaded Lab12 -W`. Estabeleça alguns valores de `k` com os quais serão realizados testes. Inicialmente, use o comando `time ./Lab12 k +RTS -N1` para verificar o tempo (`real`) de execução serial para esses valores de `k`.

Observamos que há duas oportunidades de paralelização nesse programa. A primeira consiste em avaliar os "pedaços" de `xs` em paralelo. A segunda consiste em aplicar `sum'` a cada "pedaço", e em seguida somar os resultados parciais. Estabeleça alguns valores para o argumento de `parChunks`, com os quais serão realizados testes.

A partir de agora, utilize o comando `time ./Lab12 k +RTS -Nc` para realizar os testes. Nesse comando, `k` controla o tamanho da entrada, enquanto `c` é o número de *cores* do seu computador.

Implemente a primeira abordagem. Realize testes para as combinações de valores estabelecidos. Faça o mesmo para a segunda abordagem. Em seguida, utilize as duas abordagens simultaneamente.

Monte uma planilha com os tempos de execução `real` coletados, e calcule os *speedups* obtidos (que também devem constar na planilha). Lembre-se que o *speedup* é dado pelo tempo de execução serial sobre o tempo de execução paralelo. Não esqueça de fechar a maior parte dos programas durante os testes.

5 Trabalhos práticos

Nesta seção, descrevemos os trabalhos práticos de cada Avaliação Parcial. É esperado que haja uma subseção para cada Avaliação Parcial. Na presente versão da disciplina, deve haver três subseções.

5.1 Trabalho 1

Descrevemos alguns problemas que devem ser resolvidos e implementados.

5.1.1 Problema 1

Uma agência de crédito emite cartões de crédito, cada um identificado por uma sequência de dez dígitos: os oito primeiros são significativos e os dois últimos são verificadores. Os dois dígitos verificadores representam a soma dos oito primeiros dígitos.

Escreva uma função `addSum` que, dada uma string de oito dígitos, adiciona ao seu final os dois dígitos verificadores. Você vai precisar transformar cada dígito da string de entrada em um inteiro (pode ser interessante definir `getDigit :: Char -> Int`, com o auxílio de `read`), e em seguida somá-los e transformar o resultado em uma string (usar `show` é muito útil).

Escreva uma função `valid` que, dada uma string de dez dígitos, verifica se ela é um identificador válido de um cartão. Isso é bem simples quando se usa a função `take`.

5.1.2 Problema 2

A primeira parte desse problema, que será útil para a segunda, consiste em implementar a função `until :: (a -> Bool) -> (a -> a) -> a -> a`. A função `until` recebe duas funções: a primeira é um predicado `p`, e a segunda é uma função `f` que transforma elementos do tipo `a`. Além disso, ela recebe um elemento `x` do tipo `a`. Se `x` satisfaz o predicado `p`, então `until` retorna `x`. Caso contrário, `until` testa, recursivamente, o predicado `p` para `f x`. É bastante simples implementar `until` com o auxílio de `iterate`, proposta no Exercício 10.

Agora, vamos implementar a raiz quadrada. Você pode usar o seguinte template.

```
sqrt :: Floating a => a -> a
sqrt x = until goodEnough improve x where
        goodEnough y = undefined
        improve y = undefined
```

Para implementar `goodEnough`, determine se o valor absoluto de `y*y` está suficientemente próximo de `x`. Sugerimos uma precisão de seis casas decimais.

Para implementar `improve`, pode ser útil pesquisar por alguma fórmula baseada no método de Newton (ou no método da secante).

5.1.3 Problema 3

A definição indutiva dos números naturais diz que há um primeiro número natural, digamos zero, e que cada um dos outros é obtido aplicando-se uma certa função injetora a algum natural. Essa função é chamada de *sucessora*, e zero é sucessor de nenhum natural. Isso é suficiente para determinar que os números naturais formam um conjunto infinito.

Com base nessa definição, podemos implementar os naturais da seguinte forma.

```
data Nat = Zero | Suc Nat
```

Faça de `Nat` uma instância da classe `Eq` (basta implementar `(==)`). Faça de `Nat` uma instância da classe `Ord` (basta implementar `(<=)`). Escreva as funções `natToInt :: Nat -> Int` e `intToNat :: Int -> Nat` e verifique se os comparadores estão funcionando corretamente.

5.2 Trabalho 2

Dado um conjunto de símbolos S, uma codificação $c : S \to \{0,1\}^*$ é uma função bijetora que atribui a cada símbolo uma sequência de valores binários. Uma codificação livre de prefixos tem a seguinte propriedade: $c(s_1)$ não é prefixo de $c(s_2)$, para quaisquer $s_1, s_2 \in S$.

Vamos admitir que cada símbolo de S é formado por uma sequência de valores binários. Dado $s \in S$, $|s|$ representa o comprimento da sequência de *bits* que forma s.

Se conhecemos $c : S \to \{0,1\}^*$ tal que, para cada $s \in S$, $|c(s)| \leq |s|$, então uma sequência de símbolos $s_1 s_2 \dots s_k$ seria representada mais eficientemente como $c(s_1)c(s_2)\dots c(s_k)$. No entanto, esperamos que os símbolos de S não admitam tal codificação. Uma vez que admitissem, passaríamos a representar s por $c(s)$, para todo $s \in S$. Com isso, estamos supondo que a representação dos símbolos em S é a mais eficiente possível. Via de regra, se $|S| \leq 2^l$, precisamos de no máximo l *bits* para a representação de cada símbolo de S.

Então, dada uma sequência de símbolos $s_1 s_2 \dots s_k$ de S, concatenar as representações dos símbolos é a maneira mais eficiente de represnetar a sequência? Não necessariamente. Pode ser que, nessa sequência em particular, um símbolo $\overline{s}$ ocorra muito mais que outro $\tilde{s}$. Nesse caso, seria interessante que $\overline{s}$ tivesse uma representação menor que $\tilde{s}$.

Vamos admitir que, dado o conteúdo de um arquivo de texto, queremos criar uma codificação para suas palavras, associando às palavras mais frequentes as menores sequências binárias possíveis. Utilizando funções já escritas em laboratório, sabemos construir uma `BSTree` que associa a cada palavra sua quantidade de ocorrências.

Vamos criar o seguinte tipo, que representa uma *codificação de Huffman*.

```
data HuffmanTree a = Leaf a
                   | Branch (HuffmanTree a) (HuffmanTree a)
```

Vamos usar esse tipo para representar as palavras de um arquivo de texto como suas folhas. Cada folha tem sua codificação representada pelo seu caminho a partir da raiz da árvore: descer para a esquerda (direita) representa 0 (1).

Para construir a `HuffmanTree`, criamos, para cada palavra `w`, um par `(nw, w)`, onde `nw` representa o número de ocorrências de `w`. Isso pode ser feito com um passeio em uma `BSTree`. Em seguida, criamos uma `Heap` (mínima) onde os elementos são, para cada palavra `w`, `Leaf w` com prioridade `nw`.

Agora, o algoritmo de Huffman é bem simples: enquanto for possível retirar dois elementos da `Heap`, digamos `(n1, t1)` e `(n2, t2)`, retire-os e insira `(n1 + n2, Branch t1 t2)`. Quando houver exatamente um elemento na `Heap`, esse estará representando a codificação de Huffman das palavras de acordo com seu número de ocorrências.

Com isso, podemos substituir cada palavra do arquivo pela codificação dada pela `HuffmanTree`. Para tanto, é interessante criar uma função `encoding` que, dada uma `HuffmanTree`, retorna a codificação associada a cada uma de suas folhas. Poderíamos, por fim, usar `encoding` para criar uma `BSTree`, digamos `encodingTree`, que associa cada palavra à sua codificação, e usá-la para substituir cada palavra do arquivo pela codificação associada. Lembre-se que, como uma codificação de Huffman é livre de prefixos, não precisamos separar as codificações com espaços, e assim economizamos um pouco mais de memória.

É esperado que, nesse momento, estejamos aptos a, dado um arquivo de texto como entrada, criar um arquivo de texto codificado como saída que também contém a codificação usada. Podemos organizar o arquivo de saída da seguinte forma: a primeira linha contém o conteúdo codificado do arquivo de entrada; cada uma das linhas seguintes tem uma sequência binária e sua palavra correspondente.

Para facilitar a implementação, crie um executável capaz de codificar o conteúdo do arquivo `input.txt`, produzindo o arquivo `output.txt`. Agora, vamos tratar de recuperar o conteúdo original do arquivo.

Além de criar `encodingTree`, é interessante criar `decodingTree`, uma `BSTree` chaveada pelas codificações e valorada pelas palavras. Com ela, estamos aptos a criar `decode`, que recebe uma codificação e retorna a palavra associada. Antes de poder usar `decode` em um

conteúdo codificado (que seria uma sequência enorme de valores binários), precisamos saber separar seu conteúdo.

Vamos agora criar `isValid`, que determina se uma sequência binária é uma codificação válida, isto é, verifica se é uma chave de `decodingTree`. Com ela e `inits`, vamos poder determinar o menor prefixo de uma sequência binária que é uma codificação válida, e usamos isso para criar `encodingSplit`, que dada uma sequência binária, retorna-a particionada em codificações válidas. Por fim, temos apenas o trabalho de substituir cada codificação pela palavra correspondente, intercalar a lista resultante com espaços e concatenar as strings dessa lista.

Nesse momento, esperamos estar aptos a, dado um arquivo codificado (como o produzido acima) como entrada, decodificar o conteúdo de sua primeira linha a partir da codificação descrita pelas demais linhas, e imprimir o conteúdo decodificado no terminal. Para simplificar a implementação, crie um executável capaz de decodificar `output.txt`.

5.3 Trabalho 3

Nos trabalhos práticos dessa Avaliação Parcial, cada grupo de alunos deve escolher um, e somente um, dos temas listados abaixo, ou sugerir seu próprio tema. Cada grupo de alunos deve desenvolver um programa com características de paralelismo que atenda a proposta do seu tema.

Em cada um dos temas, é esperado que, além do desenvolvimento do programa, seja também elaborada uma análise do tempo de execução e do *speedup*, de acordo com os seguintes parâmetros: tamanho da entrada; número de *cores* utilizados (controlado com o parâmetro `-N`); primeiro argumento de `parChunks` (quando utilizada). Fica a critério do grupo de alunos decidir se tal análise será apresentada em forma de gráficos ou tabelas.

Para fins de consistência, aconselhamos que todos os programas desenvolvidos nesse contexto sejam compilados com o seguinte comando. Nele, `Program.hs` representa o nome do arquivo desenvolvido.

```
ghc -O2 -threaded Program.hs
```

Esperamos também que os testes sejam feitos com o seguinte comando. Nele, `k` representa o número de *threads* a serem utilizadas pelo programa. Lembramos que `real` é o tempo que deve ser considerado.

```
time ./Program +RTS -Nk
```

Sugerimos que, para obtenção de bons resultados, os testes concorram com o menor número possível de outros programas. Assim, é bom fechar o navegador e o editor de texto antes de realizar um teste.

5.3.1 Tema 1

Nesse tema, devem ser desenvolvidas operações elementares de Álgebra Linear, como soma, subtração e multiplicação de matrizes, assim como soma, subtração e produto interno de vetores. Tais operações oferecem muitas oportunidades de paralelismo.

Por exemplo, podemos imaginar a soma $A + B$ de duas matrizes $A, B \in \mathbb{R}^{m \times n}$ dividida em m (n) tarefas, onde cada tarefa realiza a soma de duas linhas (colunas) correspondentes de A e B. Com isso, podemos usar `parChunks` para realizar grupos dessas tarefas em paralelo.

Observamos que, para a multiplicação de matrizes, é interessante que uma das matrizes esteja representada por suas linhas, enquanto a outra esteja representada por suas colunas. Assim, faz-se necessário desenvolver funções que convertam matrizes entre essas representações.

5.3.2 Tema 2

Aqui, deve ser desenvolvida uma função que, dada uma lista `x:xs` sem elementos repetidos, retorna uma lista com todos os subconjuntos de `x:xs`. Uma vez construídos os subconjuntos da cauda `xs`, é preciso realizar as tarefas de duplicar cada um deles, e acrescentar `x` em uma das cópias. Note que essas são tarefas que podem ser paralelizadas, mas são muito baratas, e assim não conseguem justificar o custo do *runtime system*. Seria possível, mesmo assim, desenvolver uma forma de paralelizá-las que resulte em ganhos no tempo de execução?

5.3.3 Tema 3

Podemos aproximar o cálculo de uma integral $\int_a^b f(x)dx$ através de uma quadratura, como a regra do trapézio ou a regra de Simpson, cujas fórmulas são fáceis de implementar. No entanto, aplicar diretamente essas quadraturas pode resultar em uma aproximação não muito boa, principalmente se $|b - a|$ for um valor muito alto. Com a propriedade $\int_a^b f(x)dx = \int_a^c f(x)dx + \int_c^b f(x)dx$, para algum $c \in [a, b]$, podemos ver a integral em um certo intervalo $[a, b]$ como a soma das integrais em subintervalos que particionam $[a, b]$, e essas podem ser aproximadas em paralelo. Note que essas tarefas são muito baratas para, individualmente, compensar o *overhead* do *runtime system*.

Assim, deve ser desenvolvida uma função que toma quatro argumentos: um integrando `Double -> Double`, uma quadratura (que também é uma função), um intervalo de integração `(Double, Double)` e um número `k` de subintervalos. Essa função deve particionar o intervalo recebido em `k` subintervalos, e aplicar a quadratura a cada um deles, de forma paralela.

5.3.4 Tema 4

Considere a função `mergesort`, proposta na Subseção 4.3. Desenvolva duas versões paralelizadas de `mergesort`.

Na primeira versão, "quebre" a lista em sublistas, ordene-as em paralelo, e faça as operações de `merge` também em paralelo. Note que a eficiência dessa abordagem está relacionada com o tamanho das sublistas que se toma.

Na segunda versão, crie uma função auxiliar `mergesort'`, que recebe, além da lista, um segundo argumento que determina a "profundidade" da chamada recursiva. Assim, `mergesort xs = mergesort' xs 0`, e `mergesort'` incrementa seu segundo argumento para suas chamadas recursivas. Com isso, pode-se determinar, de acordo com o segundo argumento, se as chamadas recursivas serão realizadas em paralelo ou não. O ideal é que apenas as chamadas recursivas mais "rasas" sejam realizadas em paralelo.

5.3.5 Tema 5

Escolha um dos temas anteriores e o desenvolva na linguagem Scala. O tema deve ser desenvolvido, preferencialmente, evidenciando as facilidades que se obtém ao integrar programação OO com programação funcional. Assim como os outros temas, esse também deve ser desenvolvido usado paralelismo. Observamos também que algumas instruções dadas no começo dessa seção não se aplicam a esse tema.