

Programação Funcional: Resumão

Sumário

Matéria da 1ª Prova.....	2
Aula 1: Tipos e Funções.....	2
Aula 2: Condicionais, casamento de padrão e list comprehension.....	4
Aula 3: Polimorfismo e classes de tipos.....	7
Aula 4: Recursão.....	9
Aula 5: Enumeração, tipos polimórficos, sinônimos e registros.....	11
Aula 7: <i>Map</i> , <i>filter</i> , composição, <i>foldr</i> e <i>foldl</i>	15
Aula 8: Estudos de caso – Criptografia e serialização.....	21
Aula 9: Estudo de caso – Árvores Binárias.....	23
Extra: Estudo de caso – Conjuntos.....	26
Matéria da 2ª Prova.....	31
Aula 10: Classes de tipos.....	31
Aula 11: Estrutura de projetos e testes.....	35
Aula 12: Avaliação <i>lazy</i>	41
Aula 13: I/O, do-Notation e manipulação de arquivos.....	43
Aula 14: Functores.....	48
Aula 15: Estudo de caso – Biblioteca para <i>parsing</i>	55
Extra: Estudo de caso – Validador de Cliente e Manipulação de arquivos.....	68
Matéria da 3ª Prova.....	73
Aula 16: Mônadas.....	73
Aula 17: Correção de Programas.....	85
Aula 18: Correção de Programas – Listas e Árvores Binárias.....	88
Extra.....	94
Funções Úteis.....	94
FAQ (<i>Frequently Asked Questions</i> / Perguntas Frequentes).....	94

Matéria da 1ª Prova

Aula 1: Tipos e Funções

Conceitos básicos

- Prelude - biblioteca automaticamente importada por todo programa Haskell, que possui uma grande quantidade de tipos e funções pré-definidas;
- Tipos, módulos e classes de tipos se iniciam com letra maiúscula;
- Funções e variáveis se iniciam com letra minúscula;
- Regra de layout: se certificar de que o editor de texto insira espaços ao apertar TAB (ao invés do caractere de tabulação) para evitar erros.
- `--` comentário de uma linha
- `{-`
comentário de múltiplas linhas
`-}`

Comandos pertinentes

- Inicializar o interpretador de Haskell no Prompt de Comando → `stack ghci`
- Carregar um arquivo de código → `:l (nomedoarquivo).hs`
- Recarrega um arquivo após modificá-lo → `:r`
- Determina o tipo de uma expressão fornecida → `:t (expressão)`
Ex: `:t True`
`True :: Bool`
- Finaliza a execução do interpretador → `:q`

Componentes da definição de uma função em Haskell

`double :: Int -> Int`

`double x = x + x`

`double :: Int -> Int` (assinatura)

nome da função :: tipo do argumento → retorno da função

`double x = x + x` (equação)

(nome da função) (parâmetros) = expressão que define a função

Funções em Haskell

- Chamando uma função em Haskell: (nome da função) (argumento #1) (argumento #2)...

Ex: `f x y`

- Função pura: não depende de efeito colateral, nada que realize atualização de memória, entrada e saída, leitura de valor de rede, geração de número aleatório, etc.
- Sempre o mesmo resultado para o mesmo parâmetro de entrada;
- Ordem de avaliação não altera o resultado final → compilador é livre para escolher qual é a melhor ordem de execução para o código;
- Enquanto em uma linguagem imperativa, o programa descreve ordens a serem executadas pela máquina, em uma linguagem funcional, o programa define o próprio problema.

Tipos Primitivos

- Int, Integer, Double, Float, Char, String, Bool
- Listas: sequências de valores do mesmo tipo, representada por `[a]`, onde `a` é um tipo;

Exemplos:

`[1,3] :: [Int]`

`[True, False, False] :: [Bool]`

`['a','b','c'] :: [Char] == String`

- Tuplas: sequências de valores de tipos possivelmente diferentes (pode ter até 62 componentes);

Exemplos:

`(1, True, "ab") :: (Int, Bool, String)`

`(True, (1, 'a')) :: (Bool, (Int, Char))`

- Tipos de Funções: possuem o formato `T1 → T2` em que `T1` é um tipo do parâmetro e `T2` do resultado.

Exemplos:

`not :: Bool → Bool`

`even :: Int → Bool`

Aula 2: Condicionais, casamento de padrão e list comprehension

Setup Inicial de um programa Haskell

```
module (Nome do módulo) where
```

```
main :: IO ()
```

```
main = return ()
```

Condicionais

- *If* sempre tem o *else* correspondente.

Ex:

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

Guardas

- Maneira mais elegante de usar condicional;
- *otherwise* é definido como *True* e pode ser usado como caso padrão em guardas.

Ex:

```
absolute1 :: Int -> Int
absolute1 n
    | n < 0 = n * (-1)
    | otherwise = n
```

Casamento de Padrões

- Eliminar padrões desnecessários usando *wildcards* \rightarrow (*underline*).

Ex:

Versão “padrão” de *&&*

```
(&&) :: Bool -> Bool -> Bool
False && False = False
False && True = False
True && False = False
True && True = True
```

Versão simplificada de *&&*

```
(&&) :: Bool -> Bool -> Bool
False && _ = False
True && b = b
```

- A variável irá assumir o valor que é passado para o parâmetro correspondente na função.

Padrões sobre Tuplas

```
fst :: (a,b) -> a
```

```
fst (x,_) = x
```

- Nesse caso, as variáveis são de tipos polimórficos (tipos quaisquer)

Cláusula *where*

Ex: Função para retornar raízes de uma equação do 2º grau, caso existam: (Delta será considerado sempre como positivo)

```
roots :: Float -> Float -> Float -> (Float, Float)
```

```
roots a b c
```

```
    = (r1, r2)
```

```
    where
```

```
        delta = b^2 - 4*a*c
```

```
        r1 = ((-b) + sqrt(delta))/(2*a)
```

```
        r2 = ((-b) - sqrt(delta))/(2*a)
```

- A cláusula *where* permite introduzir funções ou variáveis locais à função criada. Essa cláusula não é obrigatória na linguagem, mas torna o código mais legível.

Listas

- [] → lista vazia, sem nenhum elemento;
- (x : xs) é uma lista com cabeça x e cauda xs (outra lista/restante da lista);
- A lista [1,2,3,4] é uma forma simplificada de escrever 1:(2:(3:(4:[]))) → lista encadeada;

Ex: A função *unit* testa se a lista possui um único elemento.

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

```
unit [_] = True
```

```
unit _ = False
```

List comprehensions

- Notação para construção de listas inspirada em teoria de conjuntos;
- Forma geral: [expr | x <- list]

Ex: A função *addOne* criada abaixo retorna uma nova lista de inteiros onde cada elemento da lista de inteiros recebida é acrescido de 1;

```
addOne :: [Int] -> [Int]
```

```
addOne xs = [x + 1 | x <- xs]
```

Ex: A função *cartProd* criada abaixo calcula o produto cartesiano de duas listas.

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

```
cartProd xs ys = [(x,y) | x <- xs, y <- ys]
```

- A expressão `x <- xs` é chamada de *gerador*. No list comprehension acima se tem mais de um gerador, e o comportamento dele será similar ao de comandos de repetição aninhados. Ou seja, para cada elemento da lista `xs` será percorrido todos os elementos da lista `ys`.
- O uso de condições em list comprehensions pode ser feito pela forma geral:
`[expr | x <- list, condition]`

Ex: A função *onlyEven* criada abaixo retorna uma nova lista de inteiros com apenas os valores pares da lista recebida.

```
onlyEven :: [Int] -> [Int]
```

```
onlyEven xs = [x | x <- xs, even x]
```

Casamento de padrão em list comprehension

- Forma geral: `[expr | pattern <- list]`

Ex: A função *factors* criada abaixo fatora um número inteiro.

```
factors :: Int -> [Int]
```

```
factors n = [x | x <- [1..n], n `mod` x == 0]
```

Ex: Função de QuickSort,

```
qSort :: [Int] -> [Int]
```

```
qSort [] = []
```

```
qSort (x:xs) = smaller ++ [x] ++ greater
```

```
  where
```

```
    smaller = qsort [y | y <- xs, y <= x]
```

```
    greater = qsort [z | z <- xs, z > x]
```

Aula 3: Polimorfismo e classes de tipos

Introdução

- Em linguagens funcionais, o elemento básico para execução de um programa são as expressões. Em Haskell, toda expressão possui um tipo;
- Um tipo é uma coleção de valores que suportam o mesmo conjunto de operações;
- Definimos que uma expressão possui um certo tipo como `expression :: Type`;

Ex: `True :: Bool`

`'a' :: Char`

`[1,2] :: [Int]`

`(1, True) :: (Int, Bool)`

`not :: Bool -> Bool`

- Aplicações de função também possuem tipos;

Ex: `1 + 2 :: Int`

`not True :: Bool`

- Haskell, por ser fortemente tipada, não permite a execução de código com erros de tipo;
- Type Safe: nenhuma falha em tempo de execução é decorrente de erros de tipos.

Verificação de tipos

- Regra geral: se `f :: A -> B` e `x :: A` então `f x :: B`. Usando essa regra, podemos deduzir que uma expressão não é válida;

Ex:

`not :: Bool -> Bool`

`'a' :: Char`

`not 'a'`

`-- Couldn't match expected type 'Bool' with actual type 'Char'`

- Parse Error: código não segue a sintaxe/identação da linguagem.

Funções são cidadãos de 1ª classe

- Podem ser passadas como argumentos de outras funções;
- Podem retornar outras funções como seu resultado;
- Funções podem ser armazenadas em listas e outras estruturas de dados.

Ex:

`[(+), (*), (-)] -> [Int -> Int -> Int]`

`[(&&), (||)] → [Bool -> Bool -> Bool]`

Polimorfismo

- Polimorfismo paramétrico: permite a definição de código que opera da mesma forma sobre valores de tipos diferentes;

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

`length [1, 2] --Ok, a = Int`

`length ['a', 'b'] --Ok, a = Char`

- Variáveis de tipo devem ser substituídas de maneira uniforme.

Ex:

`[1, 2] ++ [3, 4] --Ok`

`[1, 2] ++ ['a', 'b'] --Erro`

- Polimorfismo de sobrecarga: permite a definição de código que opera de maneira distinta de acordo com o tipo de valores.

Ex: `(+) :: Num a => a -> a -> a`

- O termo `Num` é uma classe de tipos e uma restrição. Neste caso, a restrição é que `a` deve ser um tipo numérico.

Inferência de tipos

- Processo no qual o compilador é capaz de deduzir o tipo de uma definição.

Classes de tipos

- Define um conjunto de operações suportadas por certos tipos ditos instâncias dessa classe;
- Diversas operações da biblioteca padrão de Haskell utilizam classes de tipos;
- A classe `Num` define uma interface para tipos numéricos;
- A classe `Eq` define uma interface para tipos que suportam testes de igualdade;
- A classe `Ord` define uma interface para tipos que suportam operações de comparação;
- A classe `Show` define uma operação que converte valores em `Strings`.

Aula 4: Recursão

Introdução

- Como em linguagens funcionais não existe a noção de laços de repetição, toda iteração necessária deve ser feita por meio de recursão;

- Necessariamente, funções recursivas devem ter um caso base;

Exemplo:

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

```
length [] = 0
```

```
length (_:xs) = 1 + length xs
```

- Uma string vazia é idêntica a uma lista vazia.

Exemplo: a função replicate abaixo irá criar uma lista contendo n vezes um valor x.

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

```
replicate n x
```

```
    | n <= 0 = []
```

```
    | otherwise = x : replicate (n - 1) x
```

“Receita de bolo” para funções recursivas

1. Defina o tipo da função;
2. Enumere os casos;
3. Defina os casos base;
4. Defina os casos recursivos.

Exemplo: a função elem abaixo dirá se um elemento pertence a uma lista ou não.

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

```
2. elem _ [] = _
```

```
    elem x (y : ys) = _
```

```
3. elem _ [] = False
```

```
4. elem x (y : ys) = x == y || elem x ys
```

Função elem completa:

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

```
elem _ [] = False
```

```
elem x (y : ys) = x == y || elem x ys
```

Invertendo uma lista com um algoritmo de complexidade O(n)

```
rev :: [a] -> [a]
```

```
rev xs = rev' xs []
```

```
    where
```

```
rev' [] acc = acc
rev' (z : zs) acc = rev' zs (z : acc)
```

Exemplo de execução:

```
rev [1,2,3] ==
rev' [1,2,3] [] ==
rev' [2,3] (1 : []) ==
rev' [3] (2 : (1 : [])) ==
rev' [] (3 : (2 : (1 : []))) ==
(3 : (2 : (1 : []))) == [3,2,1]
```

Inserindo um valor em uma lista ordenadamente

```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y : ys) =
    | x <= y = x : y : ys
    | otherwise = y : insert x ys
```

Insertion Sort

```
isort :: [Int] -> [Int]
isort [] = []
isort (x : xs) = insert x (isort xs)
```

Merge Sort

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge xs [] = xs
merge (x : xs) (y : ys)
    | x <= y = x : merge xs (y : ys)
    | otherwise = y : merge (x : xs) ys
```

```
msort :: [Int] -> [Int]
msort [] = []
```

```
msort [x] = [x]
msort xs = merge (msort ys) (msort zs)
  where
    n = length xs `div` 2
    ys = take n xs
    zs = drop n xs
```

- função `take` → dado um inteiro e uma lista, ela retorna os `n` primeiros elementos desta lista;
- função `drop` → dado um inteiro e uma lista, ela remove os `n` primeiros elementos desta lista.

Aula 5: Enumeração, tipos polimórficos, sinônimos e registros

Enumeração

- Consiste em uma lista finita de valores;
- Declaração inicia com a palavra reservada `data` seguida de um nome iniciado com letra maiúscula;
- Então, se segue um ou mais construtores de dados, também iniciados por letras maiúsculas, onde cada construtor pode ter zero ou mais argumentos;

Exemplos:

```
data Direction = North | South | East | West
data Point = Point Float Float
data Shape
  = Rectangle Point Float Float
  | Circle Point Float
  | Triangle Point Point Point
```

- Cada construtor de um tipo de dados é uma função que constrói valores do tipo em questão.

Exemplos:

```
:t North
North :: Direction -- sem argumentos
```

```
:t Point
Point :: Float -> Float -> Point -- 2 argumentos
```

- Podemos usar os tipos definidos para formar listas e tuplas

Função para converter valores do tipo criado Direction para String

```
directionName :: Direction -> String
directionName North = "N"
directionName South = "S"
directionName East = "E"
directionName West = "W"
```

Função para calcular o perímetro de formas geométricas

```
perimeter :: Shape -> Float
perimeter (Rectangle _ w h) = 2*w + 2*h
perimeter (Circle _ r) = 2 * pi * r
perimeter (Triangle a b c) = dist(a,b) + dist(a,c) + dist(b,c)
```

```
dist :: (Point,Point) -> Float
dist ((Point x1,y1),(Point x2,y2)) = sqrt (x11 * x11 + y11 * y11)
  where
    x11 = x1 - x2
    y11 = y1 - y2
```

Busca sequencial no tipo recursivo IntList

```
data IntList = INil | ICons IntList

elemIntList :: Int -> IntList -> Bool
elemIntList _ INil = False
elemIntList x (ICons y ys) = x == y | elemIntList x ys
```

Busca em uma árvore no tipo recursivo IntTree

```
data IntTree = ILeaf | INode Int IntTree IntTree

elemIntTree :: Int -> IntTree -> Bool
elemIntTree _ ILeaf = False
elemIntTree x (INode y l r)
  | x < y = elemIntTree x l
```

```

| x > y = elemIntTree x r
| otherwise = True

```

Função para concatenar IntLists

```

concatIntList :: IntList -> IntList -> IntList
concatIntList INil ys = ys
concatIntList (ICons x xs) ys
    = ICons x (concatIntList xs ys)

```

Função para converter um IntList em um IntTree

```

intTreeToList :: IntTree -> IntList
intTreeToList ILeaf = INil
intTreeToList (INode x l r)
    = ICons x (concatIntList l' r')
    where
        l' = intTreeToList l
        r' = intTreeToList r

```

Tipos polimórficos

- Tipo “Maybe a” representa um possível valor do tipo a.

```
data Maybe a = Just a | Nothing
```

```
:t Just True
```

```
(Just True) :: Maybe Bool
```

```
:t Nothing
```

```
Nothing :: Maybe a
```

- No segundo exemplo, o construtor Nothing manteve seu tipo polimórfico por não haver informação para determinar sua possível instanciação da variável a.

Exemplo: Implementação segura da função Head, que retorna a cabeça de uma lista e não incorre um erro em tempo de execução caso ela seja vazia.

```
safeHead :: [a] -> Maybe a
```

```
safeHead [] = Nothing
```

```
safeHead (x : _) = Just x
```

Exemplo: A função position abaixo retorna a posição de um elemento em uma lista, caso exista. Caso não exista, a função retorna Nothing.

```
position :: Eq a => a -> [a] -> Maybe Int
```

```
position x xs = pos x xs 0
```

```
  where
```

```
    pos _ [] _ = Nothing
```

```
    pos x (y : ys) ac
```

```
      | x == y = Just ac
```

```
      | otherwise = pos x ys (ac + 1)
```

Sinônimos

- Melhora a legibilidade do código.

```
type Name = String
```

```
type Surname = String
```

```
type Height = Float
```

```
type Client = (Name, Surname, Height) --(String, String, Float)
```

Definindo o tipo Client como Tipo Algébrico

```
data Client = Customer Name Surname Height
```

Imprimindo uma mensagem de boas-vindas para um cliente

```
greet :: Client -> String
```

```
greet (Customer n _ _) = "Welcome, " ++ n ++ "!"
```

Registros

Exemplo: Definindo o tipo Client como um registro.

```
data Client
```

```
= Customer {
```

```
  name :: Name
```

```
  , surname :: Surname
```

```
  , height :: Height }
```

- Cada campo do registro fornece funções de projeção (getters).

Exemplo:

```
name :: Client -> Name
```

```
surname :: Client -> Surname
```

```
height :: Client -> Height
```

```
ex :: Client
```

```
ex = Customer "José" "Silva" 170.2
```

```
name ex == "José"
```

Aula 7: Map, filter, composição, foldr e foldl

Função que dobra todo número presente em uma lista de inteiros

```
doubleList :: [Int] -> [Int]
```

```
doubleList [] = []
```

```
doubleList (x : xs) = 2 * x : doubleList xs
```

Função que realiza a negação de todos os elementos de uma lista de booleanos

```
notList :: [Bool] -> [Bool]
```

```
notList [] = []
```

```
notList (x : xs) = not x : notList xs
```

Função que retorna todos os caracteres minúsculos de uma string

```
lowers :: [String] -> [String]
```

```
lowers [] = []
```

```
lowers (x : xs)
```

```
    | isLower x = x : lowers xs
```

```
    | otherwise = lowers xs
```

Função que retorna todos os caracteres minúsculos de uma string

```
evens :: [Int] -> [Int]
```

```
evens [] = []
```

```
evens (x : xs)
```

```
| even x = x : evens xs
| otherwise = evens xs
```

Funções de ordem superior

- Ao nos depararmos com códigos similares, devemos utilizar refatoração, extraindo o que há de comum neles e reutilizar;
- Funções que recebem outras funções como parâmetro são chamadas de funções de ordem superior.

Ex: Ambas as funções `doubleList` e `notList` retornam uma lista vazia no caso base e são chamadas recursivamente sobre a cauda no caso recursivo, porém, aplicadas de maneira diferente.

A função `map` abaixo tem por objetivo aplicar uma função sobre a cabeça de uma lista, e inserir esse resultado a frente de se chamar recursivamente sobre a cauda da lista, desta forma aplicando uma função `f` a todos os elementos da lista.

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

Ex: Ambas as funções `lowers` e `evens` retornam uma lista vazia no caso base, e no caso recursivo, inserem a cabeça da lista a frente de se chamar recursivamente sobre a cauda no resultado, caso a cabeça da lista cumpra uma determinada condição.

A função `filter` abaixo tem por objetivo determinar se cada elemento de uma lista cumpre uma determinada condição, e caso cumpra, este elemento fará parte do resultado, desta forma filtrando a lista.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x : xs)
  | p x = x : filter p xs
  | otherwise = filter p xs
```

Redefinindo `doubleList` e `notList` usando a função `map`

```
doubleList :: [Int] -> [Int]
```



```
doubleList xs = map double xs
  where
    double x = 2 * x
```

```
notList :: [Bool] -> [Bool]
notList xs = map not xs
```

Redefinindo lowers e evens usando a função filter

```
lowers :: String -> String
lowers xs = filter isLower xs
```

```
evens :: [Int] -> [Int]
evens xs = filter even xs
```

Funções anônimas

- Haskell permite definir uma função sem atribuir a elas um nome;
- Forma geral: `\ argumento -> código`
- Útil na definição de funções mais simples que são utilizadas em funções mais complexas para deixar o código mais limpo;
- Funções anônimas podem ser utilizadas em qualquer lugar em que se espera uma função.
Ex: Definindo a função `doubleList` utilizando uma função anônima no lugar de `double`.

```
doubleList :: [Int] -> [Int]
doubleList xs = map (\ x -> 2*x) xs
```

Função que muda a ordem de aplicação dos parâmetros

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \ y x -> f x y
```

```
Ex de uso: flip map [1,2,3] (\ x -> 2*x)
           [2,4,6]
```

Seções

- Aplicações parciais de operadores;
- Considere `x` e `y` como parâmetros e `#` como o operador.

- $(x \#) = \lambda x \rightarrow x \# y$
Ex: `filter (10 >) [1 .. 20]`
`[1,2,3,4,5,6,7,8,9]`
- $(\# y) = \lambda y \rightarrow x \# y$
Ex: `filter (> 10) [1 .. 20]`
`[11,12,13,14,15,16,17,18,19,20]`

Composição

- Definição matemática de composição de funções.
- $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$
 $g . f = \lambda x \rightarrow g (f x)$

Ex: `not :: Bool -> Bool`
`even :: Int -> Bool`
`odd = not . Even`

Ex: A função `maxAverage` abaixo, dado uma lista de listas de float, irá retornar a maior média entre elas.

```
maxAverage :: [[Float]] -> Float
maxAverage = maximum . map average . filter (not . null)
  where
    average xs = sum xs / fromIntegral (length xs)
```

- `(not . Null)` significa que a lista não é vazia
- `filter (not . Null)` irá retornar todas as listas não-vazias
- `average` é o cálculo da média de uma lista
- `fromIntegral` irá converter o tamanho da lista (inteiro) em um número de ponto flutuante (float), o que é necessário para fazer a divisão;
- `map average` irá calcular a média de cada lista, no final desta operação, teremos uma lista com todas as médias;
- `maximum` irá retornar o maior valor dentre as médias calculadas.

Função foldr

- Retorna um valor padrão quando a lista passada por parâmetro é vazia;
- Caso a lista não seja vazia, o resultado será a combinação entre a cabeça e o resultado da chamada recursiva da função para a cauda;
- Forma geral:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr _ v [] = v
```

```
foldr f v (x : xs) = x `f` foldr f v xs
```

```
Ex: sum [] = 0
```

```
sum (x : xs) = x + sum xs
```

```
sum = foldr (+) 0
```

```
Ex: concat [] = []
```

```
concat (xs : xss) = xs ++ concat xss
```

```
concat = foldr (++) []
```

```
Ex: and [] = True
```

```
and (x : xs) = x && and xs
```

```
and = foldr (&&) True
```

- A função foldr segue a estrutura de listas, onde a lista vazia é substituída pelo valor padrão e o operador (:) é substituído pela função passada como parâmetro.

- Ex: foldr (+) 0 (x : (y : (z : [])))
-- (x + (y + (z + 0)))

Implementando a função map usando a função foldr

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

```
map f = foldr step []
```

```
where
    step x ac = f x : ac
```

Implementando a função filter usando a função foldr

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = foldr step []
    where
        step x ac = if p x then x : ac else ac
```

Implementando a função reverse usando a função foldr (complexidade $O(n) = n^2$)

```
reverse :: [a] -> [a]
reverse = foldr step []
    where
        step x ac = ac ++ [x]
```

Função foldl

- Forma geral:
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ v [] = v
foldl f v (x : xs) = foldl f (f v x) xs
- A função foldl também segue a estrutura de listas, mas aninha chamadas à esquerda.

Ex:

```
foldl (+) 0 (x : (y : (z : [])))
      (0 +x) (y : (z : []))
```

Implementando a função reverse usando a função foldl (complexidade $O(n) = n$)

```
reverse :: [a] -> [a]
reverse = foldl step []
    where
        step ac x = x : ac
```

Aula 8: Estudos de caso – Criptografia e serialização

Criptografia: Cifra de César

- Desloca cada caractere em n posições;
Ex: encode 2 “abc” = “cde”
- A função abaixo é responsável por converter um caractere em um número inteiro:

```
char2Int :: Char -> Int  
char2Int c = ord c - ord 'a'
```
- A função abaixo é responsável por converter um número inteiro em um caractere:

```
int2Char :: Int -> Char  
int2Char n = chr (ord 'a' + n)
```
- A função abaixo é responsável por aplicar a cifra de César em um caractere. O “mod 26” permite que números maiores que 25 possam ser fornecidos como parâmetro de entrada sem que o deslocamento “saia do alfabeto”. Observação: o algoritmo a ser implementado só irá criptografar letras minúsculas.

```
shift :: Int -> Char -> Char  
shift n c  
  | isLower c = int2Char ((char2Int c + n) `mod` 26)  
  | otherwise = c
```
- A função abaixo é responsável por criptografar uma cadeia de caracteres:

```
encrypt :: Int -> String -> String  
encrypt n s = map (shift n) s
```
- A função abaixo é responsável por descriptografar uma cadeia de caracteres:

```
decrypt :: Int -> String -> String  
decrypt n s = encrypt (-n) s
```

Serialização

- A função abaixo cria uma lista de pares, juntando elementos que estão na mesma posição das duas listas fornecidas como parâmetro de entrada. Caso uma lista seja maior que a outra, a função retorna uma lista vazia como resultado;

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

- A função abaixo cria uma lista infinita com o mesmo valor. Isso é possível graças a *lazy evaluation*: o valor de uma expressão só é calculado quando ele é necessário para o resultado final;

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

- A função abaixo cria uma lista infinita de aplicações de uma função f sobre um valor inicial;

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

- A função abaixo converte uma sequência de bits em um número inteiro correspondente.

Weights gera uma lista infinita de potências de 2: [1,2,4,8,16..]. A potência de 2 será multiplicada pelo bit b correspondente e uma lista com os produtos de pares w e b será gerada. Ao final, todos os valores dessa lista serão somados, resultando em um número inteiro;

```
type Bit = Int
bin2Int :: [Bit] -> Int
bin2Int bs = sum [w * b | (w,b) <- zip weights bs]
  where
    weights = iterate (* 2) 1
```

- A função abaixo converte um valor inteiro em uma sequência de bits;

```
int2Bin :: Int -> [Bit]
int2Bin 0 = []
int2Bin n = n `mod` 2 : int2Bin (n `div` 2)
```

- A função abaixo agrupa uma sequência de bits em bytes;

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

- A função abaixo divide uma sequência de bits em uma lista de bytes;

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

- A função abaixo codifica uma string em uma sequência de bits;

```
encode :: String -> [Bit]
encode = concat . map (make8 . int2Bin . Ord)
- ord irá gerar o código ASCII de um caractere
```

- `int2Bin` irá converter o código de caractere em uma sequência de bits
- `make8` irá converter a sequência de bits em um byte
- `map` irá aplicar o procedimento (`make8 . int2Bin . Ord`) em todos os caracteres da string
- `concat` irá concatenar todas as listas de bytes geradas em uma única lista de bits
- A função abaixo decodifica uma uma sequência de bits em uma string.
`decode :: [Bit] -> String`
`decode = map (chr . bin2Int) . chop8`
 - `chop8` irá dividir a lista de bits em uma lista de bytes
 - `bin2Int` irá converter um byte em um código de caractere (número inteiro)
 - `chr` irá converter o código de caractere no caractere correspondente
 - `map` irá aplicar o procedimento (`chr . bin2Int`) em todos os bytes

Aula 9: Estudo de caso – Árvores Binárias

Definição do tipo de dados de árvores binárias

```
data Tree a
  = Leaf
  | Node a (Tree a) (Tree a)
deriving (Eq, Ord, Show)
```

Buscar um elemento em uma árvore de busca

```
memberTree :: Ord a => a -> Tree a -> Bool
memberTree _ Leaf = False
memberTree x (Node y l r)
  = case compare x y of
      LT -> memberTree x l
      GT -> memberTree x r
      _  -> True
```

- A função `compare` pertence à classe `Ord` e compara valores, retornando o construtor `LT` caso o primeiro argumento seja menor que o segundo, `GT` caso ele seja maior e `EQ` quando ambos são iguais.

Inserir um elemento em uma árvore binária

```
insertTree :: Ord a => a -> Tree a -> Tree a
insertTree _ Leaf = Node x Leaf Leaf
insertTree x (Node y l r)
    = case compare x y of
        LT -> Node y (insertTree x l) r
        GT -> insertTree Node y l (insertTree x r)
        _ -> Node y l r
```

Convertendo uma árvore binária em uma lista ($O(n) = n^2$)

```
toList :: Tree a -> [a]
toList Leaf = []
toList (Node x l r) = toList l ++ [x] ++ toList r
```

Convertendo uma árvore binária em uma lista usando um acumulador ($O(n) = n$)

```
toList :: Tree a -> [a]
toList t = toList' t []
    where
        toList' Leaf ac = ac
        toList' (Node x l r) ac
            = toList' l (x : toList' r ac)
```

Convertendo uma lista em uma árvore binária

```
fromList :: Ord a => [a] -> Tree a
fromList = foldr insertTree Leaf
```

Ordenando uma lista através de conversões entre listas e árvores ($O(n) = n^2$)

```
treeSort :: Ord a => [a] -> [a]
treeSort = toList . FromList
```

Obter o menor valor da sub-árvore à direita e removê-lo

```
removeMin :: Ord a => Tree a -> Maybe (a, Tree a)
removeMin Leaf = Nothing
```



```

removeMin (Node x Leaf r) = just (x, r)
removeMin (Node x l r)
    = case removeMin l of
        Nothing -> Nothing
        Just (y, l') -> Just (y, Node x l' r)

```

Função auxiliar removeEq

Caso um nó não possua uma das subárvores, a função `removeEq` abaixo irá retorna a outra subárvore. Caso contrário, a função `removeEq` irá substituir o valor presente no nodo da subárvore pelo seu sucessor.

```

removeEq :: Ord a => Tree a -> Tree a -> Tree a
removeEq Leaf r = r
removeEq l Leaf = l
removeEq l r
    = case removeMin l of
        Nothing -> error "Impossible!"
        Just (x, l') -> Node x l' r

```

Remover um elemento em uma árvore binária

```

remove :: Ord a => a -> Tree a -> Tree a
remove _ Leaf = Leaf
remove v (Node x l r)
    = case compare v x of
        EQ -> removeEq l r
        LT -> Node x (remove v l) r
        GT -> Node x l (remove v r)

```

Função map para árvores binárias

```

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree _ Leaf = Leaf
mapTree f (Node x l r) = Node (f x) (mapTree f l) (mapTree f r)

```

Função fold para árvores binárias

Na função `(a -> b -> b -> b)` presente abaixo, o primeiro parâmetro (a) é o elemento presente no nodo da árvore, o segundo parâmetro (primeiro b) é o acumulador da subárvore à esquerda e o terceiro parâmetro (segundo b) é o acumulador da subárvore à direita. O terceiro b é o resultado dessa função.

```
foldTree :: (a -> b -> b -> b) -> b -> Tree a -> b
foldTree _ v Leaf = v
foldTree f v (Node x l r) = f x (foldTree f v l) (foldTree f v r)
```

Função para calcular a altura de uma árvore

```
height :: Tree a -> Int
height = foldTree (\ _ acl acr -> 1 + max acl acr) 0
```

- A função `max` irá retornar o maior valor dentre os dois valores passados como parâmetro.

Extra: Estudo de caso – Conjuntos

Importando as funções `union` e `intersect` do módulo `Data.List`

```
import Data.List (union, intersect)
```

Definição do tipo de dados de árvores binárias

`Empty` é o construtor de um conjunto vazio, `Single` de um conjunto que contém apenas um elemento, `Union` de uma união entre dois conjuntos e `Intersect` de uma interseção entre dois conjuntos.

```
data Set a
  = Empty
  | Single a
  | Union (Set a) (Set a)
  | Intersect (Set a) (Set a)
  deriving (Eq, Ord, Show)
```

Função fold para conjuntos

- Será criada uma equação para cada construtor do tipo de dado `Set`;
- O primeiro parâmetro `(a -> b)` será a função `f` aplicada no conjunto;

- O segundo parâmetro ($b \rightarrow b \rightarrow b$) será função g que irá combinar as chamadas recursivas para os subconjuntos s e s' da união;
- O terceiro parâmetro ($b \rightarrow b \rightarrow b$) será função h que irá combinar as chamadas recursivas para os subconjuntos s e s' da interseção;
- O quarto parâmetro (b) será o valor retornado ao ter um conjunto vazio (`Empty`);
- O quinto parâmetro (`Set a`) será o conjunto em que a função será aplicada;
- O sexto parâmetro (b) é o retorno da função `foldSet`.

```
foldSet :: (a -> b) -> (b -> b -> b) -> (b -> b -> b) -> b -> Set a -> b
foldSet _ _ _ v Empty = v
foldSet f _ _ _ (Single a) = f a
foldSet f g h v (Union s s') = g (foldSet f g h v s)
                                (foldSet f g h v s')
foldSet f g h v (Intersect s s') = h (foldSet f g h v s)
                                    (foldSet f g h v s')
```

Função para converter um conjunto em uma lista

```
toList :: Eq a => Set a -> [a]
toList = foldSet box union intersect []
  where
    box x = [x]
```

- `box` será aplicado em conjuntos unitários;
- `union` faz a união de duas listas e será aplicada em conjuntos `Union`;
- `intersect` faz a interseção de duas listas e será aplicada em conjuntos `Intersect`;
- Uma lista vazia será retornada em caso de conjuntos vazios.

Função para converter uma lista em um conjunto usando recursividade

```
fromList :: [a] -> Set a
fromList [] = Empty
fromList (x : xs) = Union (Single x) (fromList xs)
```

- Para uma lista vazia, será retornado um conjunto vazio;
- Para uma lista não-vazia, o resultado será a união entre um conjunto que possui somente a cabeça x da lista, com o resultado de converter a cauda xs da lista.

Função para converter uma lista em um conjunto usando fold e map

```
fromList :: [a] -> Set a
```

```
fromList = foldr Union Empty . map Single
```

- `map Single` irá transformar a lista em uma lista de conjuntos unitários;
- `foldr` irá transformar a lista de conjuntos unitários em um único conjunto, usando o construtor de união de conjuntos `Union`, e quando alcançar a lista vazia, será retornado o construtor de conjunto vazio `Empty`.

Adicionando um elemento a um conjunto

```
add :: a -> Set a -> Set a
```

```
add x s = Union (Single x) s
```

Determinando se um elemento pertence a um conjunto usando elem e toList

```
member :: Eq a => a -> Set a -> Bool
```

```
member x = elem x . toList
```

- `toList` irá transformar o conjunto em uma lista;
- `elem` irá testar se o elemento pertence a essa lista.

Determinando se um elemento pertence a um conjunto usando recursão

```
member :: Eq a => a -> Set a -> Bool
```

```
member _ Empty = False
```

```
member x (Single v) = x == v
```

```
member x (Union s s') = (member x s) || (member x s')
```

```
member x (Intersect s s') = (member x s) && (member x s')
```

Determinando se um elemento pertence a um conjunto usando foldSet

```
member :: Eq a => a -> Set a -> Bool
```

```
member x = foldSet (x ==) (||) (&&) False
```

- a função `(x ==)` é utilizada no conjunto unitário;
- a função `(||)` é utilizada na união;
- a função `(&&)` é utilizada na interseção;
- O valor `False` é retornado quando o conjunto é vazio.

Função para calcular o número de elementos de um conjunto

```
size :: Eq a => Set a -> Int
```

```
size = length . toList
```

- `toList` irá transformar o conjunto em uma lista;
- `length` irá calcular o tamanho da lista.

Função map para conjuntos usando foldSet

```
mapSet :: (a -> b) -> Set a -> Set b
```

```
mapSet f = foldSet (Single . f) Union Intersect Empty
```

- A função `f` será aplicada no construtor armazenado no construtor `Single` e depois este valor será passado novamente para o construtor `Single` para construir novamente o conjunto unitário;
- Os demais construtores são para construir novamente a estrutura recursiva do conjunto usando `foldSet`.

Função map para conjuntos usando toList e fromList

```
mapSet :: (Eq a, Eq b) => (a -> b) -> Set a -> Set b
```

```
mapSet f = fromList . map f . toList
```

- `toList` irá transformar o conjunto em uma lista;
- `map f` irá aplicar a função `f` para cada elemento da lista;
- `fromList` irá transformar a lista resultante em um conjunto.

Função map para conjuntos usando recursão

```
mapSet :: (a -> b) -> Set a -> Set b
```

```
mapSet _ Empty = Empty
```

```
mapSet f (Single x) = Single (f x)
```

```
mapSet f (Union s s') = Union (mapSet f s) (mapSet f s')
```

```
mapSet f (Intersect s s') = Intersect (mapSet f s) (mapSet f s')
```

Função filter para conjuntos usando foldSet

```
filterSet :: Eq a => (a -> Bool) -> Set a -> Set a
```

```
filterSet p = foldSet select Union Intersect Empty
```

```
  where
```

```
    select x = if p x then Single x else Empty
```

- A função `select` será utilizada para aplicar o filtro em um conjunto unitário. Se `p` for verdadeiro para o elemento `x`, então será retornado o próprio conjunto unitário, e caso contrário, será retornado um conjunto vazio;
- Os demais construtores são para construir novamente a estrutura recursiva do conjunto usando `foldSet`.

Função filter para conjuntos usando toList e fromList

```
filterSet :: Eq a => (a -> Bool) -> Set a -> Set a
```

```
filterSet p = fromList . filter p . toList
```

- `toList` irá transformar o conjunto em uma lista;
- `filter p` irá aplicar a função de filtro `p` na lista;
- `fromList` irá transformar a lista resultante em um conjunto.

Função filter para conjuntos usando recursão

```
filterSet :: Eq a => (a -> Bool) -> Set a -> Set a
```

```
filterSet _ Empty = Empty
```

```
filterSet p (Single x)
```

```
  | p x = Single x
```

```
  | otherwise = Empty
```

```
filterSet p (Union s s') = Union (filterSet p s) (filterSet p s')
```

```
filterSet p (Intersect s s') = Intersect (filterSet p s) (filterSet p s')
```

Matéria da 2ª Prova

Aula 10: Classes de tipos

Introdução

- Iniciam com letra maiúscula;
- Definem uma variável de tipo que representa o tipo a ser sobrecarregado;
- Cada classe define uma ou mais funções que devem ser implementadas por suas instâncias
- Variáveis são substituídas por tipos completos;
- Assinaturas de tipos devem ser substituídas por implementações das funções para o tipo em questão;
- Só podemos usar uma função sobrecarregada se essa possuir implementação para um certo tipo;
- A extensão InstanceSigs habilita a possibilidade de anotar tipos em instâncias.

Classe Eq

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

Instância da classe Eq para o tipo Bool

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

    x /= y = not (x == y)
```

Instância da classe Eq para um tipo de dados Point

```
data Point = Point Int Int
instance Eq Point where
    (Point x y) == (Point x' y') = (x == x') && (y == y')
    x /= y = not (x == y)
```

Instância da classe Eq sobre listas

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

    xs /= ys = not (xs == ys)
```

Superclasse e símbolo =>

- Classes podem exigir que todas as suas instâncias possuam instâncias de outras classes. Nesse caso, dizemos que a primeira é uma subclasse da segunda. Ou seja, a segunda é uma superclasse da primeira.
- O símbolo => serve para:
 - Restringir um tipo polimórfico
Ex: `elem :: Eq a => a -> [a] -> Bool`
Na função `elem`, `a` deve ser uma instância do tipo `Eq`;
 - Definir relações de subclasses
Ex: `class Eq a => Ord a`
A classe `Ord` é subclasse da classe `Eq`;
 - Definir um requisito para a instância definida
Ex: `instance Eq a => Eq [a]`
É possível existir uma instância de igualdade para listas caso ela exista para os seus elementos.

Classe Ord

```
class Eq a => Ord a where
    (<) :: a -> a -> Bool
    (>) :: a -> a -> Bool
    (<=) :: a -> a -> Bool
    ((>=) :: a -> a -> Bool
    max :: a -> a -> a
    min :: a -> a -> a
```


Definições default (padrão)

- Quando diversas instâncias de uma mesma classe de tipos possuem a mesma implementação, tais definições podem ser incluídas na declaração da classe.
- Definição default para a classe Eq:

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

    x /= y = not (x == y)
    x == y = not (x /= y)
```

Derivações automáticas

- A linguagem Haskell é capaz de produzir definições de algumas funções sobrecarregadas automaticamente, como Eq, Ord, Show e Enum.
- Derivação automática de instâncias de Eq, Ord e Show para um tipo Point2D:

```
data Point2D = Point2D Int Int
              deriving (Eq, Ord, Show)
```

Derivações automáticas

- Tipos de dados para um vetor (Vector) e uma forma geométrica (Shape):

```
data Vector = Vector Float Float
              deriving (Eq, Ord, Show)
```

```
data Shape
    = Rectangle Vector Float Float
    | Circle Vector Float
    deriving (Eq, Ord, Show)
```

- Operação para realizar *scaling* (multiplicar as dimensões por um escalar) de formas e vetores:

```
class Scale a where
    scale :: Float -> a -> a
```

- Instância para um Vector:

```
instance Scale Vector where
    scale s v@(Vector x y) = Vector(x * s') (y * s')
```

where

$$s' = s / \text{norm } v$$
$$\text{norm } (\text{Vector } a \ b) = \text{sqrt } (a^2 + b^2)$$

- `v@(Vector x y)` é um *s-pattern*, a variável `v` está representando todo o valor dentro dos parênteses;
- `norm` está calculando a norma do vetor.

- Instância para um `Shape`:

```
instance Scale Shape where
```

```
    scale s (Circle p r) = Circle p (s * r)
```

```
    scale s (Rectangle p w h) = Rectangle p (s * w) (s * h)
```

- Podemos definir funções sobre tipos de classes definidos;

Ex: a função `doubleScale` abaixo dobra o tamanho de qualquer vetor ou forma geométrica

```
doubleScale :: Scale a => a -> a
```

```
doubleScale s :: scale 2.0 s
```

- Podemos definir funções sobre coleções dos tipos de classes definidos.

Ex: definindo a classe `Scale` para uma lista de valores `Scale`

```
instance Scale a => Scale [a] where
```

```
    scale s = map (scale s)
```

Sobrecargas dentro da linguagem Haskell

- Constantes numéricas são sobrecarregadas e podem ser usadas como qualquer instância da classe `Num`;
- A função `fromInteger` é sobrecarregada. Essa função converte um valor do tipo `Integer` para o tipo `a`, que é instância da classe `Num`. A assinatura dessa função é:

```
fromInteger :: Integer a -> a
```

- A sobrecarga de valores numéricos é feita pela função `fromInteger`. Isso permite a sobrecarga de quaisquer números e expressões numéricas.
- A sintaxe de intervalos usada em listas é uma operação sobrecarregada:
 - A lista `[n..m]` é uma abreviação de `enumFromTo`, que retorna uma lista contendo o valor inicial `n`, o valor final `m` e os valores entre `n` e `m`
 - A lista `[n..]` é uma abreviação de `enumFrom`, que retorna uma lista contendo o valor inicial `n`, e os valores a partir de `n`. O resultado dessa lista pode ser uma lista infinita por causa da *lazy evaluation* presente em Haskell

- A lista `[n, p..m]` é uma abreviação de `enumFromThenTo`, que retorna uma lista contendo `n`, `p`, `m` e os valores com a mesma distância entre `n` e `p`,
Ex: `[1.0, 1.2 .. 2.0] = [1.0, 1.2, 1.4, 1.6, 1.8, 2.0]`

Aula 11: Estrutura de projetos e testes

Introdução

- Programas em Haskell são organizados em pacotes e módulos;
- Pacotes são unidades de distribuição de código (ou bibliotecas) e podem ser adicionados como dependências em um projeto;
- Cada pacote possui um conjunto de módulos;
- Módulos definem funções, classes e tipos de dados que podem ser exportados.

Arquivo `package.yaml`

- Permite escrever dependências e outras configurações;
- Dependências são especificadas na seção `dependencies`;
- Todo projeto tem como dependência o pacote `base`;
- `tasty` é um framework utilizado pra testes em Haskell;
- `tasty-hunit` é um pacote utilizado para testes de unidade;
- `tasty-quickcheck` é um pacote utilizado para testes baseados em propriedades.
- Também permite a definição do executável da aplicação, especificado na seção `main`:

Comandos pertinentes em um projeto

- Para criarmos um novo projeto, utilizamos `stack new [nome do projeto]`
- Para instalarmos as dependências e compilarmos o projeto, utilizamos `stack build`
- Para executarmos o código, depois de compilá-lo, utilizamos `stack exec [nome do projeto]`

Módulos

- Haskell permite apenas um módulo por código fonte;
- O nome do arquivo deve corresponder ao nome do módulo;
- Se o arquivo está em uma determinada estrutura de diretórios, ela vai corresponder a um prefixo no nome do módulo;

Ex: O módulo `My.Long.ModuleName` está presente no diretório `My/Long/ModuleName.hs`

- Ao definir um módulo, podemos ocultar sua implementação e construtores de tipos presentes nele quando ele for importado.

Ex: `module Name (Name, mkName, render)`

Quando o módulo `Name` acima for importado, teremos acesso apenas ao tipo de dados `Name` e as funções `mkName` e `render`, mas não a implementação.

Importações

- Uma importação do tipo `import [nome do módulo]` importa todas as funções, classes e tipos presentes nele;

Ex: `import Data.List`

- Uma importação do tipo `import [nome do módulo] (nome da definição 1, nome da definição 2, nome da definição 3 ...)` importa somente as definições presentes na lista;

Ex: `import Data.List (nub, permutations)`

- Uma importação do tipo `import [nome do módulo] hiding (nome da definição 1, nome da definição 2, nome da definição 3 ...)` importa todas as definições do módulo exceto as que estão presentes na lista;

Ex: `import Data.List hiding (nub)`

- Uma importação do tipo `import qualified [nome do módulo] as [nome do identificador]` importa todas as definições do módulo, as quais devem ser utilizados usando o qualificador no código fonte. Isso é útil para evitar colisões de nome, ou seja, diferenciar duas funções que possuam o mesmo nome.

Ex: `import qualified Data.List as L`

`L.nub` deve ser utilizado no código, em vez de `nub`

Testes de unidade

- Valida o resultado de uma função de acordo com o resultado esperado;
- A biblioteca `HUnit` pode ser utilizada para construir testes unitários em Haskell;
- Exemplo: a função `mkName` tem uma string como parâmetro de entrada, e retorna um valor do tipo de dados `Name`, onde a primeira letra será maiúscula e todas as letras subsequentes serão minúsculas.;

`mkName :: String -> Name`

`mkName [] = MkName []`

```
mkName (x : xs) = MkName (x' : xs')
  where
    x' = toUpper x
    xs' = map toLower xs
```

A função `render` tem um `Name` como parâmetro de entrada, e retorna uma `string`.

```
render :: Name -> String
render (MkName s) = s
```

Exemplo de teste de unidade:

```
mkNameTest :: TestTree
mkNameTest
  = testCase "MkName Test" (s @?= "Maria")
  where
    s = render (mkName "maria")
```

- Os testes definidos em `testCase` serão agrupados em um `TestTree`;
- Um caso de teste (`testCase`) recebe como parâmetro o nome do teste (especificado como `MkName Test`) e o teste em si, dentro dos parênteses;
- A função `@?=` serve para verificar se uma determinada expressão à esquerda é igual a um resultado esperado à direita. Neste caso, estamos criando um valor do tipo `Name`, usando a função `mkName` e a string “maria”, e convertendo este resultado para uma string utilizando a função `render`. Se a implementação estiver correta, o resultado desta operação será “Maria”;
- Para executar os testes, utilizamos o comando `stack exec [nome do executável]` no terminal. Caso o resultado de um teste seja igual ao valor esperado, a mensagem OK será impressa ao lado do nome do teste.

Testes baseados em propriedades

- Valida o resultado de uma função de acordo com uma propriedade que caracteriza um resultado correto;
- A biblioteca `QuickCheck` pode ser utilizada para construir testes baseados em propriedades em Haskell;
- As propriedades utilizadas no `QuickCheck` retornam valores booleanos;

- Geradores de casos de teste também são necessários para gerar entradas aleatórias e automatizar o processo, os quais são definidos usando uma classe de tipos;
- `QuickCheck` possui uma ampla biblioteca de funções para construir geradores de caso de teste;
- Exemplo: a função `startsWithUpper` tem uma string como parâmetro de entrada, e retorna verdadeiro caso a string seja um nome válido, ou seja, vazia, ou caso a primeira letra seja maiúscula (utilizando a função `isUpper`).

```
startsWithUpper :: String -> Bool
startsWithUpper [] = True
startsWithUpper (c : _) = isUpper c
```

A função `nameCorrect` tem um string como parâmetro de entrada, e retorna um booleano. A string `s` será convertida para um `Name` usando a função `mkName`, e convertida novamente para um string usando a função `render`. Então, essa string será validada usando a função `startsWithUpper`.

```
nameCorrect :: String -> Bool
nameCorrect s = startsWithUpper (render (mkName s))
```

Exemplo de teste baseado em propriedade, executado enquanto o interpretador está ativo:

```
quickCheck nameCorrect
```

- Caso a propriedade retorne “falso” em algum dos testes, a entrada gerada é apresentada na tela como um contraexemplo;
- Quando um contraexemplo é encontrado pelo `quickCheck`, ele irá diminuí-lo para tentar encontrar um contraexemplo mínimo, assim tornando mais fácil de entender e depurar o código;
- No exemplo anterior, a função `nameCorrect` retornará falso caso a string seja composta por números. Para resolvermos esse problema:

A função `implies` modela implicação lógica.

```
implies :: Bool -> Bool -> Bool
implies x y = not x || y
```

Modificando a implementação da função `startsWithUpper`. A função `all isLetter s` irá retornar verdadeiro se a string `s` for formada apenas por letras. A função `b` irá retornar

verdadeiro se a primeira letra da string `s'` for maiúscula. A string `s'` é o resultado de converter a string `s` em um `Name`, e depois convertê-la de volta para uma string.

```
nameCorrect :: String -> Bool
nameCorrect s = (all isLetter s) `implies` b
  where
    b = startsWithUpper s'
    s' = render (mkName s)
```

- Após a mudança realizada, `quickCheck nameCorrect` imprimirá OK na tela, seguido de quantos testes foram realizados, denotando que todas as entradas testadas retornaram verdadeiro.

Estudo de caso – Insertion Sort

- Implementação base do algoritmo Insertion Sort:

```
sort :: [Int] -> [Int]
sort [] = []
sort (x : xs) = insert x xs

insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y : ys)
  | x <= y    = x : ys
  | otherwise = y : insert x ys
```

- Utilizando um teste baseado em propriedades, devemos especificar uma propriedade satisfeita por um algoritmo de ordenação. Esta propriedade não pode ser atrelada a algum algoritmo específico de ordenação, sendo válida para todos;
- A propriedade `sortPreservesLength` verifica se o tamanho da lista permanece o mesmo após a ordenação;

```
sortPreservesLength :: [Int] -> Bool
sortPreservesLength xs = length (sort xs) == length xs
```

- Executando `quickCheck sortPreservesLength`, observamos que a lista `[0,0]` é retornada como contra-exemplo. O elemento `y` também deve ser inserido a frente da cauda `ys` da lista. Corrigindo o método `insert`:

```
insert :: Int -> [Int] -> [Int]
```

```

insert x [] = [x]
insert x (y : ys)
    | x <= y    = x : y : ys
    | otherwise = y : insert x ys

```

- A função `preserves` verifica se uma propriedade `p` é válida tanto para `x`, quanto para uma função `f` aplicada em `x`;
`preserves :: Eq b => (a -> a) -> (a -> b) -> a -> Bool`
`(f `preserves` p) x = p x == p (f x)`
- Reimplementação de `sortPreservesLength`:
`sortPreservesLength :: [Int] -> Bool`
`sortPreservesLength xs = sort `preserves` length`
- A propriedade `idPreservesLength` verifica se o tamanho da lista permanece o mesmo após a ser aplicada a função identidade;
`idPreservesLength :: [Int] -> Bool`
`idPreservesLength xs = id `preserves` length`
- A função identidade (`id`) também preserva o tamanho de listas. Deste modo, devemos pensar em uma propriedade que difere um algoritmo de ordenação de outras funções sobre listas;
- A função `sorted` retorna verdadeiro caso a lista de entrada seja vazia, possua um único elemento, ou se, em listas com mais de um elemento, se o primeiro é menor que o segundo e `sorted` é verdadeiro em aplicações recursivas do segundo elemento com a cauda da lista;
`sorted :: [Int] -> Bool`
`sorted [] = True`
`sorted [_] = True`
`sorted (x : y : ys) = x <= y && sorted (y : ys)`
- A propriedade `sortEnsuresSorted` verifica se uma lista está ordenada após ser aplicado o algoritmo de ordenação `sort`;
`sortEnsuresSorted :: [Int] -> Bool`
`sortEnsuresSorted = sorted . Sort`
- Executando `quickCheck sortEnsuresSorted`, observamos que a lista `[0,0,-1]` é retornada como contra-exemplo. A função `sort` deve ser chamada recursivamente para a cauda `xs` da lista. Corrigindo o método `sort`:
`sort :: [Int] -> [Int]`


```
sort [] = []
```

```
sort (x : xs) = insert x (sort xs)
```

- Um algoritmo de ordenação também deve considerar que a lista retornada é uma permutação da lista original, contendo os mesmos elementos;
- A função `permutes` retorna verdadeiro quando o resultado de uma função `f` aplicada a uma lista `xs` é uma permutação da lista de entrada;

```
permutes :: ([Int] -> [Int]) -> [Int] -> Bool
```

```
permutes f xs = all (flip elem xs) (f xs)
```

- A função `all` testa se uma propriedade é verdadeira para todos os elementos de uma lista;
- A função `elem` testa se um elemento pertence a uma lista;
- A função `flip` foi utilizada apenas para mudar a ordem dos parâmetros de `elem`.
- A propriedade `sortPermutes` verifica se uma lista ordenada é uma permutação da lista original;

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

```
sortPermutes xs = sort `permutes` xs
```

- Executando `quickCheck sortPermutes`, observamos que todos os casos foram aprovados com sucesso.

Aula 12: Avaliação lazy

Introdução

- Argumentos são substituídos diretamente no corpo da função, sendo avaliados somente quando necessário;
- Permite *sharing* de expressões, ou seja, o resultado de cálculos de expressões podem ser compartilhados e reutilizados, não sendo necessário recalculá-los em certas situações;
- Cada expressão aguardando avaliação é armazenada como uma estrutura de dados conhecida como *thunk*. Isso pode ocasionar em um aumento do consumo de memória;
- Permite a construção de estruturas de dados “infinitas”, as quais podem ser utilizadas posteriormente por outras funções;

Ex: `ones` é uma lista “infinita” que contém somente o número 1.

```
ones :: [Int]
```

```
ones = 1 : ones
```

Ex: `nats` é uma lista “infinita” que contém “todos” os números naturais.

```
nats :: [Integer]
nats = 0 : map (+1) nats
```

```
take 2 (ones) = [1,1]
take 3 (nats) = [0,1,2]
```

- Por padrão, nenhum argumento é avaliado. Quando ocorre um casamento de padrão, avalia-se os argumentos até determinar qual equação deve ser executada. Deste modo, os argumentos são avaliados até chegarem na *weak head normal form*.
- Uma expressão está na *weak head normal form* se:
 - É um construtor de dados com expressões não avaliadas;
 - É uma função anônima;
 - É uma função *built-in* (operações e tipos de dados primitivos da linguagem, como adição ou subtração) parcialmente aplicada.

Construindo a sequência de Fibonacci usando avaliação *lazy*

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

- 0 e 1 são os dois primeiros elementos da sequência;
- `zipWith` irá aplicar a função `(+)` em elementos da mesma posição de duas listas, sendo a primeira a própria lista `fibs`, e a segunda a cauda da lista `fibs`;
- A lista `fibs` irá começar com 0 e 1, e `tail fibs` irá descartar o primeiro elemento de `fibs`, começando com 1.

```
Ex:  0 : 1 : 1 : 2 ...      ← lista fibs
    +  1 : 1 : 2 ...      ← lista tail fibs
-----
      1 : 2 : 3 ...      ← zipWith (+) fibs (tail fibs)
```

Crivo de Eratóstenes usando avaliação *lazy*

- Algoritmo utilizado para enumerar todos os números primos, desenvolvido por Eratóstenes na antiguidade;
- Passo 1: Liste todos os números naturais iniciando com 2;
- Passo 2: Seja `p` o primeiro número dessa lista;

- Passo 3: Remova todos os múltiplos de p da lista;
- Passo 4: Retornar ao passo 2, agora com o próximo número da lista.

```
primes :: [Integer]
primes = sieve [2..]
```

```
sieve :: [Integer] -> [Integer]
sieve (p : ns) = p : sieve [n | n < - ns, n `mod` p /= 0]
```

- `primes` é uma lista “infinita” contendo números primos;
- `sieve` é o algoritmo do Crivo de Eratóstenes;
- A expressão `p : [n | n < - ns, n `mod` p /= 0]` é equivalente ao passo 3;
- Chamar recursivamente a função `sieve` sobre o resultado da lista feita no passo 3 é equivalente ao passo 4.

Função `seq`

- A função `seq` força a computação do primeiro parâmetro antes de continuar a execução do segundo. Ou seja, forçamos o cálculo do argumento antes da chamada da função.
- Um operador de avaliação estrita, que possui o mesmo objetivo de `seq`, pode ser definido como:

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

Aula 13: I/O, do-Notation e manipulação de arquivos

Introdução

- No início da computação, os programas eram feitos em *batch*: executados isoladamente e produzindo resultados sem interação com seu ambiente de execução.
- Nos dias atuais os programas são interativos: respondem aos comandos feitos pelo usuário, entradas provenientes pelo ambiente, sensores e etc;
- Haskell possui transparência referencial: sempre é possível substituir um termo por sua definição sem alterar seu significado. Desta forma, o compilador é livre para escolher a melhor forma de executar o código e o paralelismo se torna mais simples;

- A interação com o mundo externo não possui transparência referencial;
- Em Haskell, o tipo usado em entrada e saída é `IO a`;
- Tipos `IO` são cidadãos de primeira classe;
- Não podemos realizar casamento de padrão em valores do tipo `IO Char`.

Modelando o tipo `IO`

```
type IO a = World -> (a, World)
```

- `World` é o tipo que representa o mundo externo;
- O tipo `IO` recebe um valor de tipo `World` e retorna um resultado `a` e um valor possivelmente modificado de `World`;

Primitivas do tipo `IO`

- `getChar :: IO Char`: retorna o caractere lido da entrada padrão;
- `putChar :: Char -> IO ()`: imprime o caractere passado como argumento na entrada padrão. Retorna `()`, equivalente ao tipo *void* em linguagens imperativas;
- `return :: a -> IO a`: converte um valor de tipo `a` em um valor de tipo `IO a`.

Atenção: Em Haskell, `return` não muda o fluxo de execução e não tem a mesma funcionalidade de linguagens imperativas.

Combinando expressões do tipo `IO`

- A função `(>>=)`, também chamada de *bind*, já é definida na biblioteca padrão de Haskell:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

```
m >>= f = \w -> let (v, w') = m w
```

```
           in f v w'
```
- `m` é uma computação de tipo `IO a`;
- Dado a configuração atual do mundo `w`, será executada a função `m` sobre esse mundo, e retornado o par `(v, w')` onde `v` é um valor e `w'` é uma nova configuração do mundo.
- Na sequência, será executada a função `f`, passando o valor `v` de tipo `a`, e o mundo `w'` como seus argumentos.

Lendo um caractere e imprimindo-o no console

```
echo :: IO ()
```

```
echo = getChar >>= \c -> putChar c
```

- `getChar` irá ler o caractere;
- O operador de *bind* `>>=` irá compor a execução do `getChar` com uma função anônima que irá pegar esse caractere e passá-lo como argumento da função `putChar`;
- `putChar` irá imprimir o caractere na tela.

Lendo uma letra e retornando uma letra maiúscula

```
getUpper :: IO Char
```

```
getUpper = getChar >>= \c -> return (toUpper c)
```

Lendo uma string inteira

```
getLine :: IO String
```

```
getLine = getChar >>= \c ->
```

```
    case c of
```

```
        '\n' -> return []
```

```
        _   -> getLine >>= \ cs -> return (c : cs)
```

- Um casamento de padrão será realizado sobre o caractere `c`, recebido da função `getChar`;
- Se o caractere for `\n`, indicando que terminou a linha, o resultado será uma lista vazia. A função `return` está convertendo a lista vazia em um valor do tipo `IO String`;
- Caso contrário, será chamado recursivamente a função `getLine`, que irá retornar uma string `cs`.
- O resultado final será converter a lista `(c : cs)`, que contém o caractere `c` lido inicialmente, junto com a string `cs` retornada pela execução recursiva de `getLine`, no tipo `IO String` usando a função `return`.

do-Notation

- Agrupamos código de IO em blocos iniciados com a palavra reservada `do`;
- O operador `<-` dá um nome a um resultado de uma operação;
- Permite a execução sequencial de ações de entrada e saída.

Lendo uma string inteira usando do-Notation

```
getLine :: IO String
```

```
getLine = do
```

```
    c <- getChar
```

```

case c of
    '\n' -> return []
    _    -> do
        cs <- getLine
        return (c : cs)

```

- A função `getChar` será executada, e o caractere retornado por ela será nomeado de `c`;
- O casamento de padrão feito é análogo ao visto anteriormente;
- A função `getLine` será executada, e seu resultado será nomeado de `cs`.
- O resultado final também é análogo ao visto anteriormente.

Imprimindo uma string no console

```

putStr :: String -> IO ()
putStr [] = return ()
putStr (x : xs) = do
    putChar x
    putStr xs

```

Imprimindo uma string no console e gerando uma quebra de linha no final

```

putStrLn :: String -> IO ()
putStrLn s = do
    putStr s
    putChar '\n'

```

Jogo de adivinhação

```

readGuess :: IO Guess
readGuess
    = do
        c <- getLine
        if c == "g" then return Greater
        else if c == "l" then return Less
        else return Equal

```

```

guess :: Int -> Int -> IO ()

```

```

guess l u
= do
  let m = (l + u) `div` 2
  putStr ("O número é " ++ show m ++ "?")
  putStrLn "(g = maior, l = menor, c = correto)"
  k <- readGuess
  case k of
    Less -> guess l (m - 1)
    Greater -> guess (m + 1) u
    Equal -> putStrLn "Acertou!"

```

- O usuário deve pensar em um número e em um intervalo em que esse número se encontra. Logo em seguida, ele deve digitar esse intervalo;
- Os dois parâmetros de entrada, l e u, são os limites inferior e superior da busca realizada;
- m será calculado, que é a média entre o limite inferior e superior;
- Será perguntado se o número é m. Se o número em que o usuário pensou é maior que o mostrado, o usuário deve digitar g; se é menor, ele deve digitar l; e se for igual, ele deve digitar c;
- Análogo ao algoritmo de busca binária.

Função para sequenciar ações em uma lista de valores de tipo IO

```

sequence_ :: [IO a] -> IO ()
sequence_ [] = return ()
sequence_ (a : as) = do
  a
  sequence_ as

```

Função que retorna uma lista de resultados obtida através de operações de tipo IO

```

sequence :: [IO a] -> IO [a]
sequence [] = return []
sequence (a : as) = do
  x <- a
  xs <- sequence as
  return (x : xs)

```

Manipulações de Arquivos

- Tipo de dados `FilePath` usado na manipulação de arquivos:

```
type FilePath = String
```

- A função de leitura de arquivos `readFile` tem o caminho do arquivo como parâmetro de entrada e retorna uma string com o conteúdo daquele arquivo:

```
readFile :: FilePath -> IO String
```

- A função de escrita em arquivos `writeFile` tem o caminho do arquivo e o conteúdo que vai ser escrito como parâmetros de entrada e retorna o conteúdo do arquivo alterado como resultado:

```
writeFile :: FilePath -> String -> IO String
```

Função main

- O início da execução de um programa em Haskell se dá pela função `main`, que deve estar contida em um arquivo chamado `Main.hs` ou `Main.lhs`:

```
main :: IO ()
```

```
main = ...
```

Aula 14: Functores

Função fmap

- Considere as seguintes definições do tipo `Maybe` e `Tree`:

```
data Maybe a = Just a | Nothing
```

```
data Tree a
```

```
    = Leaf
```

```
    | Node a (Tree a) (Tree a)
```

```
    deriving (Eq, Ord, Show)
```

- Considere as seguintes definições da função `map` para os tipos `Maybe` e `Tree`:

```
mapMay :: (a -> b) -> Maybe a -> Maybe b
```

```
mapMay _ Nothing = Nothing
```

```
mapMay f (Just x) = Just (f x)
```

```
mapTree :: (a -> b) -> Tree a -> Tree b
```

```
mapTree _ Leaf = Leaf
```

```
mapTree f (Node x l r)
```



```
= Node (f x) (mapTree f l) (mapTree f r)
```

- Observando as assinaturas das funções `mapMay` e `mapTree`, podemos generalizar a função `map` da seguinte forma:

```
fmap :: (a -> b) -> f a -> f b
```

Kinds

- *Kinds* são “tipos” de tipos;
- Tipos simples, como `Bool`, `String` e `Int` são de *kind* ‘*’;
- Construtores de tipos como `Maybe` e `Tree` precisam de um parâmetro de tipo para serem tipos efetivamente. Deste modo, eles são funções em nível de tipo, e possuem *kind* ‘* -> *’;
- O construtor de par (`()`) “recebe como parâmetro de tipo” dois valores, para então ser definido como um par. Logo, ele possui um *kind* ‘* -> * -> *’;
- Podemos usar o comando `:k` no `stack ghci` para consultar o *kind* de um tipo.
- Exemplo: Dado a definição do tipo `GRose` abaixo, qual é o seu *kind*?

```
data GRose f a = a :> f (GRose f a)
```

- Primeiramente, `(GRose f a)` tem um *kind* ‘*’;
- O *kind* de `a` também é ‘*’;
- Como `f` é usado dentro da definição do construtor como um valor aplicado ao tipo `(GRose f a)`, que tem *kind* ‘*’, o *kind* de `f` é ‘* -> *’;
- Finalmente, como o construtor de `GRose` possui `f` e `a` como seus “parâmetros de tipo”, seu *kind* será ‘(* -> *) -> * -> *’.

Functores

- É possível construir uma instância da classe `Functor` para qualquer tipo que possua um *kind* ‘* -> *’;
- A função `fmap` pode ser utilizada como o operador (`<$>`);
- As instâncias da classe `Functor` para os tipos `Maybe` e `Tree` serão:

```
instance Functor Maybe where
```

```
  fmap _ Nothing = Nothing
```

```
  fmap f (Just x) = Just (f x)
```

```
instance Functor Tree where
```

```
  fmap _ Leaf = Leaf
```

```
fmap f (Node x l r) = Node (f x) (f <$> l) (f <$> r)
```

- Exemplo: Dada o tipo Rose definindo abaixo, defina uma instância da classe Functor:

```
data Rose a = Rose a [Rose a] deriving (Eq, Ord, Show)
```

```
instance Functor Rose where
```

```
fmap f (Rose x ts) = Rose (f x) ((fmap f) <$> ts)
```

- A função `f` será aplicada ao valor `x`;
- `ts` é uma lista de valores do tipo `Rose`;
- O operador `<$>` irá executar a função `map` para a lista `ts`;
- A função `fmap` irá aplicar a função `f` no conteúdo da lista `ts`, ou seja, para os valores do tipo `Rose`.

Estudo de caso – E-Mail

- Tipo de dado para o endereço de e-mail:

```
data Address = Address String deriving (Eq, Ord, Show)
```

- Tipos de dados para o remetente e o destinatário, respectivamente:

```
type From = Address
```

```
type To = Address
```

- Tipo de dado para o corpo do e-mail:

```
data Body = Body String deriving (Eq, Ord, Show)
```

- Tipo de dado para o e-mail completo:

```
data Email = Email From To Body deriving (Eq, Ord, Show)
```

- Função que detecta se uma string é vazia:

```
nonEmpty :: String -> Maybe String
```

```
nonEmpty [] = Nothing
```

```
nonEmpty s = Just s
```

- Função que verifica se uma string `xs` contém o caractere `x`:

```
contains :: Char -> String -> Maybe String
```

```
contains x xs
```

```
  | x `elem` xs = Just xs
```

```
  | otherwise   = Nothing
```

- Validando um endereço de e-mail:

```
mkAddress :: String -> Maybe Address
```

```
mkAddress s = Address <$> contains '@' s
```

- Quando `contains` retornar um `Just`, o construtor `Address` será aplicado ao valor associado ao construtor `Just`, criando assim um valor do tipo `Maybe Address`.
- Quando `contains` retornar um `Nothing`, a instância de `Functor` irá retornar `Nothing`.

- Validando o corpo do e-mail:

```
mkBody :: String -> Maybe Body
```

```
mkBody s = Body <$> nonEmpty s
```

De maneira análoga, o construtor `Body` será aplicado quando o resultado de `nonEmpty` for `Just`. Caso contrário, o resultado será um `Nothing`.

- Validando o e-mail completamente:

```
mkEmail :: String -> String -> String -> Maybe Email
```

```
mkEmail from to body
```

```
    = case mkAddress from of
```

```
      Nothing -> Nothing
```

```
      Just fromAddr ->
```

```
        case mkAddress to of
```

```
          Nothing -> Nothing
```

```
          Just toAddr ->
```

```
            case mkBody body of
```

```
              Nothing -> Nothing
```

```
              Just nBody ->
```

```
                Just (Email fromAddr toAddr nBody)
```

- `case mkAddress from of` está validando o e-mail do remetente;
- `case mkAddress to of` está validando o e-mail do destinatário;
- `case mkBody body of` está validando o corpo do e-mail;
- Caso todos os aspectos anteriores forem válidos, o e-mail será construído.

Funtores aplicativos

- Permitem a aplicação de uma função sobre valores contidos em uma “estrutura” de construtor de tipos;
- Um functor aplicativo pode ser utilizado com o operador `(<*>)`;
- A definição da classe de functor aplicativo é:

```
class Functor f => Applicative f where
```

```
  pure  :: a -> f a
```

```
  (<*>) :: f (a -> b) -> f a -> f b
```

- Temos uma estrutura de dados de tipo `f` que armazena funções do tipo `(a -> b)` e uma estrutura de dados `f` que armazena funções do tipo `a`. O resultado dessa função é uma estrutura de dados `f` que armazena funções do tipo `b`.
- A função `pure` serve para converter um valor de tipo `a` qualquer na estrutura de dados `f` em questão.

- Considere a instância da classe de functor aplicativo para o tipo `Maybe`:

```
instance Applicative Maybe where
```

```
  pure = Just
```

```
  Nothing <*> _ = Nothing
```

```
  _ <*> Nothing = Nothing
```

```
  (Just f) <*> (Just x) = Just (f x)
```

- Para converter um valor de tipo `a` para um valor de tipo `Maybe a` significa apenas usar o construtor `Just`;
- Caso algum dos parâmetros do construtor `<*>` seja `Nothing`, o resultado será `Nothing`;
- Caso ambos os parâmetros do construtor `<*>` forem `Just`, o resultado será um valor de tipo `Just` contendo a função `f` aplica ao valor `x`.

Redefinindo a validação de E-mail usando `<$>` e `<*>`

```
mkEmail' :: String -> String -> String -> Maybe Email
```

```
mkEmail' from to body
```

```
  = Email <$> mkAddress from <*> mkAddress to <*> mkBody body
```

- Os valores do tipo `Maybe` correspondentes a validação do remetente, destinatário e corpo do e-mail serão compostos usando o operador `<*>`;
- O resultado final é feito utilizando o operador `<$>` passando o construtor `Email`;
- Caso ocorram erros na validação do e-mail, esta definição não os obtém. Isto é importante para identificar o que causou a falha na validação do e-mail e pode ser usado para mostrar boas mensagens de erro ao usuário.

Semigrupo

- Estrutura algébrica que é formada por um conjunto e uma operação binária associativa;

- Em Haskell, é uma classe de tipos que possui uma função binária que deve ser associativa;
- Definição da classe Semigroup:

```
class Semigroup a where
    (<*) :: a -> a -> a
```

Estudo de caso – E-Mail com mensagens de erro

- Tipo de dado para validadores:

```
data Validation err a
    = Failure err
    | Success a
    deriving (Eq, Ord, Show)
```

- O construtor Failure irá armazenar uma mensagem de erro err;
- O construtor Success irá armazenar um resultado de sucesso a na validação.

- Criando uma instância de Validation para a classe Functor:

```
instance Functor (Validation err) where
    fmap _ (Failure err) = Failure err
    fmap f (Success x)   = Success (f x)
```

- Fazendo com que a variável err na instância de Applicative seja uma instância de Semigroup:

```
instance Semigroup err => Applicative (Validation err) where
    pure = Success
    Failure e1 <(*) b = Failure $ case b of
        Failure e2 -> e1 <(*) e2
        Success _  -> e1
    Success _ <(*) Failure e2 =
        Failure e2
    Success f <(*) Success a =
        Success (f a)
```

- Quando o primeiro parâmetro for uma falha, será feito um casamento de padrão no segundo parâmetro: caso ele também for uma falha, as duas mensagens de erro serão “concatenadas” usando a operação binária de semigrupo <(*)>; senão, será retornado apenas o primeiro erro;

- Quando temos um sucesso no primeiro parâmetro e uma falha no segundo, será retornado apenas esta falha;
- Quando temos sucesso em ambos os parâmetros, será retornado a aplicação da função `f` sobre o valor `a`, encapsulando este resultado no construtor `Success`.
- Tipo de dado para erro:


```
data Error
    = MustNotBeEmpty
    | MustContain String
    deriving (Eq, Ord, Show)
```

 - O construtor `MustNotBeEmpty` representa que o corpo do e-mail não pode estar vazio;
 - O construtor `MustContain` representa que um endereço de e-mail deve conter o caractere '@'.
- Validação do endereço de e-mail:


```
atString :: String -> Validation [Error] Address
atString s
    | '@' `elem` s = Success (Address s)
    | otherwise    = Failure [MustContain "@"]
```

 - Se o endereço de e-mail `s` possui um '@', a função retorna um sucesso, passando o endereço de e-mail para o construtor `Address`;
 - Caso contrário, será retornado o erro `MustContain "@"`, afirmando que o endereço de e-mail deve conter um '@'.
- Validação do corpo do e-mail:


```
nonEmptyString :: String -> Validation [Error] Body
nonEmptyString [] = Failure [MustNotBeEmpty]
nonEmptyString s  = Success (Body s)
```

 - Se o corpo do e-mail estiver vazio, será retornado o erro `MustNotBeEmpty`, afirmando que o corpo do e-mail deve conter uma mensagem.
 - Caso contrário, a função retorna um sucesso, passando o corpo `s` do e-mail para o construtor `Body`;
- Validando o e-mail completamente e obtendo as mensagens de erro:


```
email :: String -> String -> String -> Validation [Error] Email
email from to body =
    Email <$> atString from <*> atString to <*> nonEmptyString body
```

Aula 15: Estudo de caso – Biblioteca para *parsing*

Introdução

- O tipo Parser a ser definido utilizará uma abordagem conhecida como *list of sucess*, onde:
 - No caso de uma falha, será retornado uma lista vazia;
 - Quando existir apenas um resultado, será retornado uma lista unitária;
 - Quando existir vários resultados, eles serão combinados em uma lista. Caso um desses resultados ocasione uma falha, a biblioteca irá descartar esse par que ocasionou a falha, e continuar o processamento com os demais elementos da lista (*backtracking*);

Tipo de dados Parser

```
newtype Parser s a = Parser {runParser :: [s] -> [(a,[s])]}
```

- *s* representa o tipo dos elementos da entrada, normalmente por caractere;
- *a* representa o resultado produzido pelo processamento da entrada.
- *runParser* é uma função que recebe uma lista de entrada, e produz como resultado uma lista de pares, no qual o primeiro componente é um valor produzido como resultado do processamento dessa entrada, e o segundo componente é o restante da entrada a ser processado.

Definindo *parsing* para um símbolo

```
symbol :: Eq s => s -> Parser s s
```

```
symbol s = Parser (\ inp -> case inp of  
  [] -> []  
  (x : xs) -> if x == s  
    then [(x,xs)]  
    else [])
```

- Será utilizado um tipo polimórfico *s*, e será exigido que ele possua uma instância de *Eq*;
- Dado o símbolo *s* na entrada, será construído um Parser (através do construtor *Parser*), através de uma função anônima;
- Na função anônima, será recebido uma lista *inp* de valores de tipo *s*, no qual será realizado um casamento de padrão:
 - Caso a lista seja vazia, será retornado uma lista vazia, indicando um erro no processamento;

- Caso a lista não seja vazia, se a cabeça x da lista for igual a s , será retornado uma lista contendo um par, formado pela cabeça x da lista (processamento de um símbolo) e pela cauda xs da lista (restante da entrada). Como o objetivo do Parser é processar o símbolo s , caso a cabeça x da lista não for igual a s , então obtivemos uma falha de processamento, e será retornado uma lista vazia.

Exemplo de um *parsing* simples

Processando o caractere 'a' através de uma definição a .

```
a :: Parser Char Char
```

```
a = symbol 'a'
```

Executando no interpretador:

```
Main*> runParser a "a"
```

```
[('a',"")]
```

```
Main*> runParser a "abc"
```

```
[('a',"bc")]
```

```
Main*> runParser a "aaaa"
```

```
[('a',"aaa")]
```

Definindo *parsing* para uma sequência de símbolos (*token*)

```
token :: Eq s => [s] -> Parser s [s]
```

```
token s = Parser (\ inp -> if s == (take n inp)
```

```
    then [(s, drop n inp)]
```

```
    else [])
```

```
    where
```

```
        n = length s
```

- Ao passar uma lista de valores do tipo s , será processado um prefixo da entrada e verificado se esse prefixo é exatamente a string s , que, em caso verdadeiro, retornará a string s como resultado e removê-la da entrada;
- n é o tamanho da string s ;
- O Parser irá comparar a string s fornecida como parâmetro de entrada com o resultado de $(take\ n\ inp)$, ou seja, dos n primeiros caracteres da entrada inp (um prefixo). Se:

- As strings forem iguais, então o prefixo da entrada é igual a string que se deseja processar. Então, será retornado uma lista contendo um par, que possui como primeiro elemento a string *s*, e como segundo elemento o restante da entrada, representado por `(drop n inp)`, que remove os *n* primeiros caracteres da entrada *inp*.
- As strings forem diferentes, será retornado uma lista vazia, indicando um erro no processamento.

Parser SAT

```
sat :: (s -> Bool) -> Parser s s
sat p = Parser (\ inp -> case inp of
    [] -> []
    (x : xs) -> if p x
        then [(x,xs)]
        else [])
```

- O Parser SAT recebe uma função `(s -> Bool)`, que representa uma condição, e irá processar a string *s* com sucesso caso essa condição seja verdadeira para o primeiro elemento da entrada;
- Dada uma condição *p*, um Parser será criado com a entrada *inp*, a qual será realizada um casamento de padrão. Se:
 - A entrada for uma lista vazia, será retornado uma lista vazia;
 - A entrada não for uma lista vazia, se o primeiro elemento *x* da entrada satisfaz o predicado *p*, então é retornado uma lista, contendo um par que possui como primeiro elemento a cabeça *x* da lista, e como segundo elemento a cauda *xs*. Caso esse primeiro elemento *x* da entrada não satisfaça o predicado *p*, então é retornado uma lista vazia.

Digit Parser

Podemos usar o Parser SAT para construir um Parser para processar um dígito da entrada.

```
digitChar :: Parser Char Char
digitChar = sat isDigit
```

Exemplos de execução de *parsing* simples no interpretador

```
Main*> runParser (Symbol 1) [1,2,3]
```

```

[(1,[2,3])]
Main*> runParser (Symbol 1) [2,3]
[]
Main*> runParser (token [1..5]) [1..10]
[[[1,2,3,4,5],[6,7,8,9,10]]]
Main*> runParser (token "abc") "abcd"
[("abc","d")]
Main*> runParser (token "abc") "ab"
[]
Main*> runParser (sat isUpper) "Abb"
[('A',"bb")]
Main*> runParser (sat isUpper) "bb"
[]

```

Instância de Functor para Parsers

```
instance Functor (Parser s) where
```

```

  fmap f (Parser p)
    = Parser (\ inp -> [(f x, xs) | (x,xs) <- p inp])

```

- Dado uma função `f` e um `Parser p`, será criado um outro parser, que a partir da entrada `inp`, irá executar o parser `p` sobre ela. Podemos usar *list comprehension* para o processamento da lista desse resultado. Então, para cada par de resultado `(x,xs)`, será aplicada a função `f` em `x` e mantido `xs`, encapsulados em um par.

Convertendo o caractere resultante do Digit Parser em um número

```
digit :: Parser Char Int
```

```
digit = f <$> digitChar
```

```
  where
```

```
    f c = ord c - ord '0'
```

- Na função `f`, dado um caractere `c`, a função `ord` irá convertê-lo em um valor inteiro, no qual será subtraído o resultado de converter o caractere `'0'` em um número.
- Desse modo, o parser `digit` retornará um número inteiro, e não mais um caractere, como o parser `digitChar`.

Parser *succeed*

Este parser nunca falha, não processa a entrada e não “consume” nenhum símbolo da entrada.

```
succeed :: a -> Parser s a
```

```
succeed v = Parser (\ inp -> [(v,inp)])
```

Parser *failure*

Este parser sempre falha, ignorando a entrada.

```
failure :: Parser s a
```

```
failure = Parser (\ _ -> [])
```

Combinador “ou” de parsers

```
(<|>) :: Parser s a -> Parser s a -> Parser s a
```

```
(Parser p) <|> (Parser q)
```

```
= Parser (\ inp -> p inp ++ q inp)
```

- Dado um parser com uma função *p* e outro parser com função *q*, será criado um novo parser que, a partir da entrada *inp*, será executado *p* sobre a entrada e concatenado esse resultado com o resultado de executar *q* sobre a entrada.
- Se o parser *p* obtiver sucesso, o parser *q* não será executado sobre a entrada. Deste modo, o parser *q* só será executado caso o parser *p* falhe.

Modelando uma execução sequencial sobre parsers

```
instance Applicative (Parser s) where
```

```
pure = succeed
```

```
(Parser p) <*> (Parser q)
```

```
= Parser (\ inp -> [(f x, xs) | (f, ys) <- p inp  
                               , (x, xs) <- q ys])
```

- O combinador *<*>* da classe *Applicative* irá permitir que o parser *q* utilize o restante da entrada produzido pelo parser *p* para continuar o processamento.
- *pure* pode ser igualado ao parser *succeed*;
- O parser *p* será executado sobre a entrada *inp*, produzindo um par contendo uma função *f* e o restante *ys* da lista;
- Então, o parser *q* será executado sobre esse restante *ys*, produzindo um par contendo o resultado *x* e o restante *xs* da lista;

- O resultado do processamento sequencial dos dois parsers p e q será o resultado de aplicar a função f sob x e o restante xs da lista.

Exemplos de execução de *parsing* usando <|> e <*>

Definindo um parser p1 que processa o símbolo a e um parser p2 que processa o símbolo b:

```
Main*> let p1 = symbol 'a'
```

```
Main*> let p2 = symbol 'b'
```

```
Main*> runParser (p1 <|> p2) "ba"
```

```
[('b',"a")]
```

```
Main*> runParser (p1 <|> p2) "ab"
```

```
[('a',"b")]
```

```
Main*> runParser (p1 <|> p2) "c"
```

```
[]
```

```
Main*> runParser ((\ c1 c2 -> [c1,c2]) <$> p1 <*> p2) "abb"
```

```
[("ab","b")]
```

Observação: A execução sequencial formada por n *parsers* exige a aplicação de uma função de n parâmetros para construir o resultado dessa sequência de processamentos.

Parser para reconhecer parênteses balanceados

- Tipo de dados para representar palavras de parênteses balanceados:

```
data Paren = Match Paren Paren | Empty
```

O construtor Empty representa a string vazia e o construtor Match representa strings com parênteses balanceados;

- Instância da classe Show para o tipo Paren:

```
instance Show Paren where
```

```
    show Empty = ""
```

```
    show (Match p p') = "(" ++ show p ++ ")" ++ show p'
```

- A string correspondente ao construtor Empty é uma string vazia;
- A string correspondente ao construtor Match é formada por '(', seguido de converter p em uma string, seguido de ')', seguido de converter p' em uma string.

- Parser para processar um '(':

`open :: Parser Char Char`

`open = symbol '('`

- Parser para processar um ')':

`close :: Parser Char Char`

`close = symbol ')'`

- Parser para processar uma string que contenha '(' e ')':

`parens :: Parser Char Paren`

`parens = (f <$> open <*> parens <*> close <*> parens)`

`<|> succeed Empty`

`where`

`f _ p _ p' = Match p p'`

- A função `f` será aplicada em quatro parâmetros, mas o primeiro e o terceiro parâmetros são os caracteres '(' e ')' processados. Logo, esses parâmetros não são necessários para construir um valor do tipo `Paren`, sendo considerados para tal apenas o segundo (primeira chamada recursiva) e o quarto (segunda chamada recursiva) parâmetros.

Parser *option*

`option :: Parser s a -> a -> Parser s a`

`option p d = p <|> succeed d`

- O parser `option p d` reconhece a entrada aceita por `p` ou retorna o valor padrão `d` caso `p` falhe sobre a entrada.

Parsers *many*

`many :: Parser s a -> Parser s [a]`

`many p = ((:) <$> p <*> many p) <|> succeed []`

`many1 :: Parser s a -> Parser s [a]`

`many1 p = (:) <$> p <*> many p`

- Um parser *many* permite repetir sucessivamente a aplicação de um parser sobre a entrada. Este parser irá executar várias vezes o parser `p` fornecido como parâmetro e combinar os resultados dessas execuções em uma lista de resultados.
- Dado um parser `p`, será executado sequencialmente o parser `p` (que retorna um valor do tipo `a`), seguido da chamada recursiva pra `many p` (que retorna um valor do tipo `[a]`). O

resultado desses dois parses serão combinados usando o construtor de listas (:). Caso a execução do parser `p` falhe, o resultado será dado pelo parser `succeed` retornando uma lista vazia de resultados.

- O parser `many1` vai reconhecer pelo menos uma vez o parser `p`, ou seja, ele não dá a possibilidade de retornar uma lista vazia de resultados.

Exemplos de execução do parser *many*

Definindo um parser `p3` que irá reconhecer 'a' ou 'b' várias vezes:

```
Main*> let p3 = many (p1 <|> p2)
```

```
Main*> runParser p3 "abba"
```

```
[("abba", ""), ("abb", "a"), ("ab", "ba"), ("a", "bba"), ("", "abba")]
```

Observação: O parser `many` gera todos os possíveis resultados intermediários.

```
Main*> runParser p3 "c"
```

```
[("", "c")]
```

```
Main*> runParser p3 "abc"
```

```
[("ab", "c"), ("a", "bc"), ("", "abc")]
```

```
Main*> runParser ((\s c -> s ++ [c]) <$> p3 <*> symbol 'c') "abc"
```

```
[("abc", "")]
```

Parser que processa um número natural

```
natural :: Parser Char Int
```

```
natural = foldl f 0 <$> many digit
```

```
  where
```

```
    f ac d = ac * 10 + d
```

- `many digit` irá produzir uma lista de números inteiros correspondentes aos dígitos processados da entrada;
- A função `foldl` irá multiplicar os dígitos dessa lista pela potência de 10 correspondente à posição do dígito.

Função *first*

```
first :: Parser s a -> Parser s a
```

```
first (Parser p) = Parser (\ inp -> let r = p inp
```

```
in if null r then []  
    else [head r])
```

- A função `first` descarta resultados intermediários;
- O parser `p` fornecido como argumento será executado sobre a entrada, e a partir do resultado dessa execução, será retornado o primeiro elemento dessa lista de resultados, denotado por `[head r]`.

Parsers *greedy*

```
greedy :: Parser s a -> Parser s [a]  
greedy = first . many
```

```
greedy1 :: Parser s a -> Parser s [a]  
greedy1 = first . Many1
```

- Os parsers *greedy* possuem a mesma funcionalidade dos parsers *many*, mas irão descartar os resultados intermediários.

Parser que reconhece identificadores de uma linguagem de programação

```
identifier :: Parser Char String  
identifier = (:) <$> letter <*> greedy (sat isAlphaNum)  
    where  
        letter = sat isLetter
```

- Normalmente, linguagens de programação utilizam a seguinte regra para identificadores: devem começar com uma letra, seguido de uma quantidade qualquer de letras ou números (caracteres alfanuméricos);
- `letter`, ou `(sat isLetter)` irá processar se o primeiro caractere é uma letra;
- `greedy (sat isAlphaNum)` irá processar se os demais caracteres são alfanuméricos, retornando apenas o resultado final;
- `<*>` é responsável por executar sequencialmente o que foi descrito nos dois tópicos anteriores;
- o construtor de lista `(:)` irá combinar o resultado de `(sat isLetter)` com `greedy (sat isAlphaNum)` em uma string.

Parser *listOf*

`listOf :: Parser s a -> Parser s b -> Parser s [a]`

`listOf p sep = (:) <$> p <*> many ((\ x y -> y) <$> sep <*> p)`

- O parser `listOf` recebe um parser `p` e um parser `sep` para o separador, e processa uma lista de elementos que são separados pelo separador `sep`;
- Primeiramente, será executado o parser `p`;
- Sequencialmente, será executado várias vezes o parser `sep` seguido do parser `p`.

Parser *pack*

`pack :: Parser s a -> Parser s b -> Parser s c -> Parser s b`

`pack p q r = (\ _ x _ -> x) <$> p <*> q <*> r`

- O parser `pack` é similar ao `listOf`, utilizado em separadores que possuem uma estrutura no início e outra no final, como os parênteses.
- O parser `pack` recebe um parser `p` que representa o primeiro separador, o parser `q` do resultado e o parser `r` do último separador e retorna um parser no resultado;
- Os três parsers serão executados em sequência por meio do operador `<*>`, mas os resultados dos parsers para os separadores serão descartados.

Parser que processa conteúdo entre parênteses

`parenthesized :: Parser Char a -> Parser Char a`

`parenthesized p = pack (symbol '(') p (symbol ')')`

- Um parser `p` é fornecido como parâmetro de entrada;
- Será utilizado o parser `pack`, passando um parser para processar o caractere '(', o parser `p` e um parser para processar o caractere ')';
- Então o parser `pack` irá processar a entrada usando o parser `p` e irá descartar os resultados referentes ao processamento dos caracteres '(' e ')

Parser *endBy*

`endBy :: Parser s a -> Parser s b -> Parser s [a]`

`endBy p sep = greedy ((\ x _ -> x) <$> p <*> sep)`

- O parser `endBy` é similar ao parser `listOf`, mas considera que o separador vem no final da sequência.

- Será executado várias vezes o processamento pelo parser `p`, seguido pelo parser `sep` do separador. Porém, a função anônima irá descartar o resultado do parser `sep` do separador.

Processando arquivos CSV

- Arquivos CSV são usados para a representação textual de dados em tabelas (planilhas).
 - Os dados são representados como strings;
 - Separadores são usados para dividir colunas;
 - As linhas no arquivo denotam linhas na tabela.

- Tipos de dados usados para representar um arquivo CSV:

```
type CSV = [Line]
```

```
type Line = [Cell]
```

```
type Cell = String
```

- Parser para a célula:

```
cellParser :: Parser Char Cell
```

```
cellParser = greedy valid
```

```
  where
```

```
    valid = sat (\ c -> notElem c ",\n")
```

Este parser irá determinar quando um caractere pode ser considerado o conteúdo de uma célula, o qual será qualquer caractere que não seja `,` ou `\n`;

- Parser para a linha:

```
lineParser :: Parser Char Line
```

```
lineParser = listOf cellParser (symbol ',')
```

Será usado o parser `listOf`, utilizando `,` para separar colunas;

- Parser para o arquivo CSV:

```
csvParser :: Parser Char CSV
```

```
csvParser = endBy lineParser eol
```

```
  where
```

```
    eol = symbol '\n'
```

O parser `endBy` irá executar diversas vezes o parser de linha `lineParser`, as quais estarão separadas pelo caractere `\n`, processados pelo parser `eol` ou `(symbol '\n')`.

Parser para um operador binário associativo à direita

```
chainr :: Parser s a -> -- expressão
```

```
Parser s (a -> a -> a) -> -- operador
```

```
Parser s a
```

```
chainr pe po
```

```
= h <$> many (j <$> pe <*> po) <*> pe
```

```
where
```

```
  j x op = op x
```

```
  h fs x = foldr ($) x fs
```

- O parser `pe` é um parser para uma expressão, cujo resultado é um valor do tipo `a`;
- O parser `po` é um parser para o operador, cujo resultado é uma função binária do tipo `(a -> a -> a)`;
- Em `many (pe <*> po)`, será executado várias vezes o parser `pe` seguido do parser `po`, que montará uma lista na qual teremos “uma expressão seguida de um operador seguida de uma expressão...” Já que o operador é binário, para concluir a expressão, adicionamos mais um parser de expressão `pe` ao final, utilizando o operador `<*>`;
- A função `h` irá aplicar o valor `x` retornado pelo *parsing* da última expressão `pe` à lista de funções `fs` retornada durante a execução do parser `many`;

Parser para um operador binário associativo à esquerda

```
chainl :: Parser s a -> -- expressão
```

```
Parser s (a -> a -> a) -> -- operador
```

```
Parser s a
```

```
chainl pe po
```

```
= h <$> pe <*> many (j <$> po <*> pe)
```

```
where
```

```
  j op x = \ y -> op y x
```

```
  h x fs = foldl (flip ($)) x fs
```

- O `chainl` é bastante similar ao `chainr`, mas agora a expressão adicional `pe` se encontra à esquerda do parser `many` (o qual também possui a ordem de execução sequencial dos parsers trocadas). Também será utilizada a função `foldl` ao invés da função `foldr`.

Processando expressões aritméticas

- Nesse estudo de caso, a multiplicação terá precedência sobre a adição;
- Tipo de dados para representar a sintaxe de expressões:

```
data Exp
```

```
  = Const Int
```

```
  | Exp :+: Exp
```

```
  | Exp *: Exp
```

```
  deriving (Eq, Ord, Show)
```

- O construtor `Const` representa uma constante;
- O construtor `Exp :+: Exp` representa a soma de duas expressões;
- O construtor `Exp *: Exp` representa a multiplicação de duas expressões.

- Parser para um fator:

```
factorParser :: Parser Char Exp
```

```
factorParser
```

```
  = parenthesized (expParser) <|>
```

```
    (Const <$> natural)
```

- Deste modo, um fator é uma expressão entre parênteses ou um número. O parser de um número será a execução do parser `natural`, que retorna um número natural, aplicado ao construtor de constante `Const`.

- Parser para a operação de multiplicação:

```
termParser :: Parser Char Exp
```

```
termParser
```

```
  = chainr factorParser pmult
```

```
    where
```

```
      pmult = const (:*:) <$> symbol '*'
```

- Será executada a expressão, que é um fator, através de `factorParser`, seguido do parser `pmult` para o símbolo de multiplicação;
- Em `pmult`, será executado o parser `symbol '*'` e será retornado como resultado o construtor `(:*)` do tipo expressão relativo à operação de multiplicação;
- Deste modo, o `chainr` irá gerar uma expressão considerando como separador o símbolo `'*'` que será substituído pelo construtor `:*:`.

- Parser para a operação de adição:

```
expParser :: Parser Char Exp
```

```
expParser
```

```
  = chainr termParser pplus
```

```
    where
```

```
pplus = const (:+:) <$> symbol '+'
```

- Será executada a expressão, que é um termo, através de `termParser`, seguido do parser `pplus` para o símbolo de adição;
- Em `pplus`, será executado o parser `symbol '+'` e será retornado como resultado o construtor `(:+:)` do tipo expressão relativo à operação de adição;
- Deste modo, o `chainr` irá gerar uma expressão considerando como separador o símbolo '+' que será substituído pelo construtor `:+:`.

Extra: Estudo de caso – Validador de Cliente e Manipulação de arquivos

Estudo de caso – Validador de Cliente

- Para um cliente ser válido:
 - Seu nome deve ser formado apenas por espaços ou letras e o tamanho mínimo é de 3 caracteres;
 - Seu CPF deve ser uma string formada por 11 dígitos;
 - Sua idade deve ser um número natural.

- Definindo sinônimos para o nome, CPF e idade:

```
type Name = String
```

```
type CPF = String
```

```
type Age = Int
```

- Definindo o tipo de dados para a representação do cliente:

```
data Client = Client Name CPF Age deriving (Eq, Ord, Show)
```

- Definindo o tipo de dados para a representação de erros provenientes dos validadores:

```
data Error
```

```
  = InvalidAge
```

```
  | InvalidCPF
```

```
  | InvalidName
```

```
  deriving (Eq, Ord, Show)
```

- `repeatParser` irá repetir o processamento de um parser `p`, `n` vezes:

```
repeatParser :: Int -> Parser s a -> Parser s [a]
```

```
repeatParser n p
```

```
  | n <= 0 = succeed []
```

| otherwise = (:) <\$> p <*> repeatParser (n - 1) p

- Caso n seja menor ou igual à 0, o parser retornará uma lista vazia;
- Caso contrário, o parser p será executado, e na sequência, repeatParser será chamado recursivamente, decrementando o valor de n em 1.

- Parser responsável por processar o nome:

parseName :: Parser Char Name

parseName

= (++) <\$> repeatParser 3 chrP <*> greedy chrP

where

chrP = sat (\ c -> isSpace c || isLetter c)

- chrP é um parser que processa espaços ou letras. Este parser será executado 3 vezes;
- Em sequência, greedy irá executar o parser chrP várias vezes, descartando resultados intermediários;
- Os resultados retornados por estes dois processamentos serão concatenados pelo construtor (++).

- Parser responsável por processar o CPF (apenas dígitos):

parseCPF :: Parser Char CPF

parseCPF = repeatParser 11 digitP

where

digitP = sat isDigit

- digitP é um parser que processa dígitos. Ele será executado 11 vezes para processar todos os dígitos presentes no CPF.

- Parser responsável por processar o CPF (no formato: XXX.XXX.XXX-XX):

parseFormatCPF :: Parser Char CPF

parseFormatCPF

= f <\$> three <*> dot <*> three <*> dot <*> three <*> dash <*> two

where

f v1 _ v2 _ v3 _ v4 = concat [v1,v2,v3,v4]

three = repeatParser 3 digitChar

two = repeatParser 2 digitChar

dot = symbol '.'

dash = symbol '-'

- three é um parser que processa três dígitos em sequência;

- `two` é um parser que processa dois dígitos em sequência;
- `dot` é um parser que processa o caractere `'.'`;
- `dash` é um parser que processa o caractere `'-'`;
- Os parsers citados acima serão executados sequencialmente de forma a processar o formato `XXX.XXX.XXX-XX`;
- A função `f`, aplicada no resultado desses parsers, irá descartar o resultado dos parsers `dot` e `dash` e combinar o resultado dos parsers `two` e `three`, concatenando-os.

- Parser responsável por processar a idade:

```
parseAge :: Parser Char Age
```

```
parseAge = natural
```

- `natural` é um parser que processa números naturais.

- Função que lê uma mensagem e a imprime na tela:

```
readData :: String -> IO String
```

```
readData msg
```

```
    = do
```

```
        putStr (msg ++ ":")
```

```
        getLine
```

- Definindo o tipo de dados para a representação de um leitor (*reader*):

```
type Reader a = IO (Validation [Error] a)
```

- Função que valida um campo (nome, CPF ou idade) do cliente:

```
reader :: String -> Error -> Parser Char a -> Reader a
```

```
reader msg err p
```

```
    = do
```

```
        s <- readData msg
```

```
        case runParser p s of
```

```
            [] -> return (Failure [err])
```

```
            ((x,_) : _) -> return (Success x)
```

- `reader` irá receber como parâmetros: a mensagem `msg` que irá imprimir na tela, o tipo de erro `err` que irá ocorrer caso o processamento do parser `p` falhe, e o parser `p`, responsável por processar um campo do cliente;
- `msg` será impresso na tela, produzindo o resultado `s`;
- O parser `p` será executado, processando `s`;

- Caso o processamento de `p` falhe e uma lista vazia seja retornada, o resultado será uma falha, contendo o erro `err`;
- Caso contrário, o resultado será o resultado `x` do processamento efetuado por `p`.
- Leitor responsável por validar o nome:


```
readName :: Reader Name
readName = reader "Name" InvalidName parseName
```

 - A mensagem "Name" será impressa na tela;
 - O erro causado por um nome inválido é denominado `InvalidName`;
 - O parser responsável por processar o nome é o `parseName`.
- Leitor responsável por validar o CPF:


```
readCPF :: Reader CPF
readCPF = reader "CPF" InvalidCPF parseCPF
```

 - A mensagem "CPF" será impressa na tela;
 - O erro causado por um CPF inválido é denominado `InvalidCPF`;
 - O parser responsável por processar o CPF é o `parseCPF`.
- Leitor responsável por validar a idade:


```
readAge :: Reader Age
readAge = reader "Age" InvalidAge parseAge
```

 - A mensagem "Age" será impressa na tela;
 - O erro causado por uma idade inválida é denominado `InvalidAge`;
 - O parser responsável por processar a idade é o `parseAge`.
- Leitor responsável por validar um cliente:


```
readClient :: Reader Client
readClient
  = do
    nm <- readName
    cpf <- readCPF
    ag <- readAge
    return (Client <$> nm <*> cpf <*> ag)
```

 - As validações do nome, CPF e idade do cliente serão executadas sequencialmente e conterão os erros de validação caso eles ocorram;
 - O construtor `Client` irá criar um cliente com os resultados retornados.

Programa que retorna o número de linhas e de palavras de um arquivo

```
main :: IO ()
```

```
main
```

```
  = do
```

```
    args <- getArgs
```

```
    case args of
```

```
      [] -> putStrLn "Erro!\nInforme um arquivo"
```

```
      (f : _) -> do
```

```
          s <- readFile f
```

```
          let ls = lines s
```

```
              ws = concatMap words ls
```

```
          putStrLn $ "Linhas:" ++ (show $ length ls)
```

```
          putStrLn $ "Palavras:" ++ (show $ length ws)
```

- A função `getArgs` retorna uma lista contendo os argumentos da linha de comando fornecidos na execução do programa no terminal;
- Caso a lista retornada seja vazia, nenhum arquivo foi passado como argumento no terminal, e uma mensagem de erro será mostrada na tela;
- Caso um arquivo `f` seja passado como argumento no terminal, a função `readFile` irá ler o conteúdo deste arquivo e armazená-lo em `s`;
- A função `lines` irá retornar uma lista contendo as linhas de `s` e armazenar em `ls`;
- A função `words` irá retornar uma lista de listas contendo as palavras de cada linha de `ls`;
- A função `concatMap` irá concatenar todas as listas de palavras de cada linha em uma lista de palavras e armazenar esse resultado em `ws`;
- As funções `putStrLn` irão imprimir na tela as mensagens e o tamanho das listas `ls` (número de linhas presente em `ls`) e `ws` (número de palavras presente em `ws`).

Matéria da 3ª Prova

Aula 16: Mônadas

Introdução

- Considere o tipo de dados `Exp` abaixo que representa uma expressão matemática, e a função que avalia uma expressão matemática e realiza as devidas operações `eval`:

```
data Exp
  = Const Int
  | Exp :+: Exp
  | Exp **: Exp
  | Exp :/: Exp
  deriving (Eq, Ord, Show)

eval :: Exp -> Maybe Int
eval (Const n)
  = Just n
eval (e :+: e')
  = case eval e of
      Just n ->
        case eval e' of
          Just m -> Just (n + m)
          Nothing -> Nothing
      Nothing -> Nothing
eval (e **: e')
  = case eval e of
      Just n ->
        case eval e' of
          Just m -> Just (n * m)
          Nothing -> Nothing
      Nothing -> Nothing
eval (e :/: e')
  = case eval e of
```

```

Just n ->
  case eval e' of
    Just m -> if m == 0 then Nothing
              else Just (n `div` m)
    Nothing -> Nothing
Nothing -> Nothing

```

- A definição apresentada acima possui repetição no casamento de padrão sobre o tipo `Maybe`.

Então, podemos criar uma função para abstrair esse casamento:

```
(>>?) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
(Just v) >>? f = f v
```

```
Nothing >>? _ = Nothing
```

Podemos entender `(a -> Maybe b)` como a continuação da computação;

- Usando a função `(>>?)`, podemos simplificar a definição de `eval`:

```
eval :: Exp -> Maybe Int
```

```
eval (Const n) = Just n
```

```
eval (e :+: e') =
```

```
  eval e >>? (\ n ->
```

```
    eval e' >>? \m -> Just (n + m))
```

```
eval (e :*: e') =
```

```
  eval e >>? (\ n ->
```

```
    eval e' >>? \m -> Just (n * m))
```

```
eval (e :/: e') =
```

```
  eval e >>? (\ n ->
```

```
    eval e' >>? \m -> if m == 0
```

```
      then Nothing
```

```
      else Just (n `div` m))
```

As funções anônimas encontradas após a função `>>?` representam a continuação do resultado de avaliar a expressão anterior `e`. `n` será o possível resultado da avaliação de `e`, e `m` será o possível resultado da avaliação de `e'`;

- A função `>>?` impõe uma ordem de execução sobre as ações a serem feitas por uma função;

Mônada

- Uma mônada é uma classe de tipos e sub-classe de `Applicative`. Sua definição é:

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

- Esta classe define duas funções:
 - `return` irá, a partir de um valor do tipo `a`, o transformar em um valor monádico do tipo `m a`;
 - `(>>=)`, que possui a mesma funcionalidade da função `(>>?)` vista anteriormente, mas não restrito ao tipo `Maybe`;

- A instanciiação de uma mônada para o tipo `Maybe` será:

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

- A *do-Notation* pode ser utilizada para qualquer instância da classe `Monad`;
- Operações de IO são exemplos de mônada;
- A mônada `Maybe` costuma ser utilizada para lidar com situações de erro e exceções;

Implementando `eval` utilizando mônada

```
eval :: Monad m => Exp -> m Int
eval (Const n) = return n
eval (e :+: e')
    = eval e >>= \ n ->
      eval e' >>= \ m ->
      return (n + m)
eval (e :+: e')
    = eval e >>= \ n ->
      eval e' >>= \ m ->
      return (n * m)
eval (e :/: e')
    = eval e >>= \ n ->
      eval e' >>= \ m ->
      if m == 0 then fail "Division by zero"
      else return (n `div` m)
```

- O `return` será responsável por encapsular o resultado das operações matemáticas (tipo `Int`) no tipo `m Int`;
- O `fail` irá imprimir uma mensagem de erro na tela caso a condição para que ele seja chamado seja verdadeira.

Implementando `eval` utilizando mônada e *do-Notation*

```
eval :: Monad m => Exp -> m Int
```

```
eval (Const n) = return n
```

```
eval (e :+: e')
= do
    n <- eval e
    m <- eval e'
    return (n + m)
```

```
eval (e :*: e')
= do
    n <- eval e
    m <- eval e'
    return (n * m)
```

```
eval (e :/: e')
= do
    n <- eval e
    m <- eval e'
    if m == 0 then fail "Division by zero"
    else return (n `div` m)
```

Mônada de listas (*List monads*)

- Utilizada para representar a possibilidade de não-determinismo, ou seja, quando uma função pode retornar mais de um resultado;
- A instanciação de uma mônada para listas será:

```
instance Monad [] where
    return x = [x]
    xs >>= f = concatMap f xs
```

O operador `>>=` irá aplicar a função `f` em cada um dos elementos de `xs` e concatená-los ao mesmo tempo utilizando a função `concatMap`, produzindo assim uma lista contendo os resultados.

Função *guard*

A função `guard` abaixo irá receber um booleano e irá falhar a computação da mônada caso ele for falso, e irá “retornar um *void*” caso ele seja verdadeiro:

```
guard :: Monad m => Bool -> m ()
guard True  = return ()
guard False = fail "failure"
```

Exemplo: Triplas pitagóricas

```
triples :: Int -> [(Int,Int,Int)]
triples n
  = do
    x <- [1..n]
    y <- [1..n]
    z <- [1..n]
    guard (x^2 == y^2 + z^2)
    return (x,y,z)
```

- A função `triples` tem por objetivo criar triplas pitagóricas que estão no intervalo $[1,n]$, onde n é fornecido como parâmetro;
- As listas `x`, `y` e `z` serão construídas com os valores de 1 até n ;
- Se $x^2 = y^2 + z^2$, então `guard` irá retornar `void` e o `return` será executado, incluindo esta tripla na lista de resultados finais. Quando o teste for falso ($x^2 \neq y^2 + z^2$), o resultado será uma lista vazia e nenhum elemento será passado para a execução do `return`. Este trecho de código está realizando uma busca exaustiva.

Exemplo: Modificação de uma árvore binária

- Considere o tipo de dados `Tree` abaixo que representa uma árvore binária:

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

deriving (Eq, Ord, Show)

- Nesta definição de Tree, os valores são armazenados nas folhas;
- Considere o seguinte problema: para cada nó da árvore, atribua um número inteiro único.

Exemplo:

Dada a árvore t1 a seguir:

```
t1 :: Tree Char
t1 = Node (Node (Leaf 'a') (Leaf 'b'))
          (Leaf 'c')
```

fazer uma função que a transforme na árvore t2 a seguir:

```
t2 :: Tree (Char, Int)
t2 = Node (Node (Leaf ('a',0)) (Leaf ('b',1)))
          (Leaf ('c',2))
```

- Este problema pode ser resolvido pela função label abaixo:

```
label :: Tree a -> Tree (a,Int)
label = fst . flip labelAcc 0
  where
    labelAcc (Leaf x) n
      = (Leaf (x,n) , n + 1)
    labelAcc (Node t1 tr) n
      = (Node t1' tr' , n2)
      where
        (t1' , n1) = labelAcc t1 n
        (tr' , n2) = labelAcc tr n1
```

- labelAcc irá receber uma árvore e um número inteiro n como entrada e:
 - Quando a árvore for formada apenas por uma folha, será criada uma nova folha, e será retornado como resultado um par, contendo a árvore alterada e o contador n acrescido de 1;
 - Quando a árvore for um nó interno com duas subárvores, a função labelAcc será chamada recursivamente para a subárvore à esquerda t1 com o contador n, a qual retornará uma nova subárvore à esquerda t1' e um novo contador n1. Em seguida, a função labelAcc será chamada recursivamente para a subárvore à direita tr com o contador n1, a qual retornará uma nova subárvore à direita tr' e um novo contador n2.

Deste modo, o resultado final deste nó será dado por um nó que utiliza as subárvores alteradas tl' e tr' e o último contador obtido $n2$.

- A função `labelAcc` tem como assinatura:

```
labelAcc :: Tree a -> Int -> (Tree a, Int)
```

Esta função recebe um estado (o número inteiro), e retorna o resultado da computação (`Tree a`, modificada) e um novo estado, que pode ser o próprio estado anterior ou um estado atualizado.

Mônada de estado (*State*)

- A definição de uma mônada de estado, ou *State*, é:

```
newtype State s a
    = State { runState :: s -> (a, s) }
```

- Uma Mônada de estados é um tipo de dados que armazena uma função que, a partir de um estado s , retorna um par contendo o resultado da computação a e um novo estado s ;

- A função `put` abaixo modifica o valor de um estado:

```
put :: s -> State s ()
put s = State (\ _ -> ((), s))
```

- A função `get` abaixo obtém o valor atual de um estado:

```
get :: State s s
get = State (\ s -> (s, s))
```

- A instancição de um `Functor` para uma mônada de estado é:

```
instance Functor (State s) where
    fmap f (State g)
        = State (\ s ->
            let (v, s') = g s
            in (f v, s'))
```

- g é uma função que, dado um estado s , irá retornar um par (v, s') onde v é o resultado da computação e s' é o novo estado;
- A função f será aplicada sobre os resultados da computação v dessa mônada de estado;

- A instancição de um `Applicative` para uma mônada de estado é:

```
instance Applicative (State s) where
    pure v = State (\ s -> (v, s))
    (State f) <*> (State g)
```

```

= State (\s -> let (h, s1) = f s
                (v, s2) = g s1
                in (h v, s2))

```

- Para `pure`, que irá converter um valor de tipo `a` em um valor de tipo `State s a`, o resultado da computação será o par contendo o valor `v` e o estado `s` sem modificações;
- Para a execução sequencial de duas computações de estado `f` e `g`, dado um estado inicial `s`, será executado a computação `f` nesse estado `s`, gerando um par contendo uma função `h` e um novo estado `s1`; e em seguida, a computação `g` receberá o estado `s1`, gerando um par contendo um valor `v` e um novo estado `s2`. Finalmente, o resultado final será um par, contendo o resultado de aplicar a função `h` sobre o valor `v`, e o estado `s2`.

- A instanciação de um `Monad` para uma mônada de estado é:

```

instance Monad (State s) where
    return = pure
    (State m) >>= f
        = State (\ s -> let (v, s') = m s
                        in runState (f v) s')

```

- `return` será a própria definição de `pure` da classe `Applicative`;
- Em `>>=`, dado um estado `s`, será executada a função `m` sobre `s`, resultando em um par contendo um valor `v` e um novo estado `s'`, e então, será aplicada a função `f` sobre `v`, resultando em um `State`, o qual será executado pela função `runState` e aplicado o novo estado `s'` sobre ele.
- A definição da falha para uma mônada de estado é:

```

instance MonadFail (State s) where
    fail s = error s

```

Exemplo: Reimplementando a modificação de uma árvore binária utilizando *State*

- A função `fresh` abaixo irá obter o inteiro atual e incrementá-lo no estado:

```

fresh :: State Int Int
fresh
    = do
        n <- get
        put (n + 1)
        return n

```


- Reimplementando label:

```
label :: Tree a -> Tree (a, Int)
label t = fst (runState (mk t) 0)
  where
    mk (Leaf v)
      = do
        n <- fresh
        return (Leaf (v, n))
    mk (Node tl tr)
      = do
        tl' <- mk tl
        tr' <- mk tr
        return (Node tl' tr')
```

- A computação (mk t) será iniciada com o estado inicial 0;
- A função fresh irá gerar um novo número inteiro n;
- Para uma folha, uma nova folha será criada, armazenando o valor v que já estava previamente armazenado em conjunto com o novo inteiro n;
- Para um nó, a função mk será chamada recursivamente sobre cada uma das subárvores tl e tr, e construindo o nó final contendo tl' e tr'.

Exemplo: Interpretador de uma máquina de pilha (simplificação da máquina virtual de Java)

- Considere os tipos de dados Var e Instr abaixo, os quais modelam uma variável e uma instrução, respectivamente:

```
type Var = String
```

```
data Instr
  = Push Int | Set Var | Get Var
  | Add | Jump Int | JumpZero Int
  deriving (Eq, Ord, Show)
```

- Push é uma instrução para empilhar valores;
- Set é uma instrução que associa o valor do topo da pilha a uma variável;
- Get é uma instrução que vai obter o valor de uma variável e empilhá-lo;
- Add é uma instrução que vai somar os dois valores mais ao topo da pilha;

- `JumpInt` é uma instrução para realizar um pulo incondicional;
- `JumpZero` é uma instrução para realizar um pulo condicional, caso o topo da pilha for igual a zero;
- Considere o tipo de dados `Stack` abaixo, que modela a pilha:
`type Stack = [Int]`
- Considere o tipo de dados `PC` abaixo, que modela o contador de instruções:
`type PC = Int`
- Considere o tipo de dados `Mem` abaixo, que modela a memória, associando variáveis a números inteiros:
`type Mem = [(Var,Int)]`
- Considere o tipo de dados `Conf` abaixo, que modela a configuração de uma máquina virtual, sendo esta uma tripla formada pelo contador de instruções, pelo estado atual da pilha e pela memória:
`type Conf = (PC, Stack, Mem)`
- Considere o sinônimo de tipos `VM` abaixo, que representa computações que envolvem como estado a configuração da máquina e que irão retornar como resultado um valor do tipo `a`:
`type VM a = State Conf a`
- A função `addPC` irá incrementar o contador de instruções em um determinado valor:
`addPC :: Int -> VM ()`
`addPC n`
`= do`
`(pc,st,m) <- get`
`put (pc + n, st, m)`
 - A função `get` irá obter a configuração inicial da máquina;
 - A função `put` irá atualizar essa configuração, incrementando o contador de instruções em `n`.
- A função `push` irá receber um número inteiro e inseri-lo na pilha de execução:
`push :: Int -> VM ()`
`push n`
`= do`
`(pc,st,m) <- get`
`put (pc, n : st, m)`
- A função `set` irá alterar um valor na memória:

```
set :: Var -> VM ()
```

```
set v
```

```
  = do
```

```
    (pc,n:st,m) <- get
```

```
    put (pc, st, (v,n) : m)
```

- Esta instrução irá considerar que a pilha não está vazia;
- O elemento *n*, presente no topo da pilha, será desempilhado e inserido em um par, juntamente com a variável *v*, na memória.

- A função *look* irá procurar um valor na memória:

```
look :: Var -> VM ()
```

```
look v
```

```
  = do
```

```
    (pc,st,m) <- get
```

```
    let st' = maybe 0 id (lookup v m) : st
```

```
    put (pc, st', m)
```

- A função *lookup* possui a assinatura abaixo e, dado um valor de tipo *a*, esse valor será procurado no primeiro elemento da lista de pares (*a*, *b*). Caso ele seja achado, será retornado *Just b*, e caso não seja, será retornado *Nothing*;

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

- Deste modo, a variável *v* será procurado na memória através da função *lookup*. Caso *v* seja encontrado, a função *id* será aplicada sobre esse resultado através da função *maybe*. Caso contrário, o valor 0 será retornado.
- O resultado de *maybe* será inserido à frente da pilha atual, produzindo uma nova pilha *st'*, utilizada na nova configuração da máquina.

- A função *add* irá somar os dois valores mais ao topo da pilha:

```
add :: VM ()
```

```
add = do
```

```
    (pc,n:p:st,m) <- get
```

```
    put (pc, (n + p) : st, m)
```

- Para que uma instrução seja executada, será efetuado um casamento de padrão sobre a estrutura da instrução. Após a execução de uma instrução, o valor do contador de instruções irá ser incrementado em 1.
- A função *instr* abaixo é responsável por executar uma instrução:

```

instr :: Instr -> VM ()
instr (Push n)
    = do
        push n
        addPC 1
instr (Set v)
    = do
        set v
        addPC 1
instr (Get v)
    = do
        look v
        addPC 1
instr Add
    = do
        add
        addPC 1
instr (Jump n)
    = addPC n
instr (JumpZero n)
    = do
        (pc,m:st,p) <- get
        let pc' = if m == 0 then pc + n else pc
        put (pc',st,p)

```

- Para o pulo condicional `JumpZero`, a configuração atual da máquina será recuperada pela função `get`, e caso o topo da pilha for igual à 0, o contador de instrução será incrementado em `n`. Caso o topo da pilha não seja igual à 0, o contador de instrução permanecerá igual.
- A função `execM` irá executar um programa completo que contenha as instruções definidas anteriormente:

```

execM :: [Instr] -> VM ()
execM = mapM_ instr

```

- A função `mapM_`, que possui a assinatura abaixo, irá descartar o resultado `b` produzido ao executar a computação:

$$\text{mapM_} :: \text{Monad } m \Rightarrow (a \rightarrow m \ b) \rightarrow [a] \rightarrow m \ ()$$
- Deste modo, a função `instr` será executada sobre cada uma das instruções presentes no programa.
- Para obter a configuração final, será executado a função `execM` sobre as instruções `is` e uma configuração inicial `initConf` (formada pelo contador de instruções inicialmente em 0, uma pilha vazia e uma memória vazia), o que retornará um par `((), Conf)`. Deste modo, para obter a configuração final, basta utilizar a função `snd`, que irá retornar o segundo elemento presente no par visto:

$$\text{exec} :: [\text{Instr}] \rightarrow \text{Conf}$$

$$\text{exec } is$$

$$= \text{snd } (\text{runState } (\text{execM } is) \text{ initConf})$$

$$\text{where}$$

$$\text{initConf} = (0, [], [])$$

Aula 17: Correção de Programas

Introdução

- Funções matemáticas não dependem de valores “ocultos” ou que possam ser alterados. Isso facilita a demonstração de propriedades ou teoremas sobre essas funções;
 Ex: $2 + 3 = 5$ tanto em $4 * (2 + 3)$ quanto em $(2 + 3) * (2 + 3)$.
- Teoremas podem ajudar na performance: substituir implementações ineficientes por equivalentes mais eficientes;
- Teoremas são a forma de mostrar que um código atende aos requisitos corretamente;
- Como Haskell possui transparência referencial, podemos provar propriedades sobre programas usando raciocínio baseado em equações, como na matemática.

Análise de caso: Concatenação e Função `reverse`

- Considere as seguintes definições para as funções de concatenação e inversão ineficiente de listas:

$$(++) :: [a] \rightarrow [a] \rightarrow [a]$$

$$[] ++ ys = ys$$

$(x:xs) ++ ys = x : (xs ++ ys)$

`reverse :: [a] -> [a]`

`reverse [] = []`

`reverse (x : xs) = reverse xs ++ [x]`

- Prove que, para todo x, o inverso de uma lista contendo um único elemento x é a própria lista contendo o elemento x:

`reverse [x] = -- teorema a ser provado`

`reverse (x : []) = -- definição de reverse`

`reverse [] ++ [x] = -- definição de reverse`

`[] ++ [x] = -- definição de concatenação`
`[x]`

Análise de caso: Função not

- Considere a seguinte definição para a função not:

`not :: Bool -> Bool`

`not False = True`

`not True = False`

- Prove que a função not é involutiva, ou seja, que aplicá-la duas vezes em um valor x resultará no próprio valor x:

- Caso x for falso:

`not (not False) = -- definição de not`

`not True = -- definição de not`

`False`

- Caso x for verdadeiro:

`not (not True) = -- definição de not`

`not False = -- definição de not`

`True`

Análise de caso: Números Naturais

- Considere a seguinte representação para números naturais, que utiliza a notação de Peano, onde um número natural é zero ou o sucessor de outro número natural:

`data Nat = Zero | Succ Nat deriving (Eq, Ord, Show)`

Ex: Representação do número 2 utilizando a definição de Nat:

two :: Nat

two = Succ (Succ Zero)

- Considere a seguinte definição de soma para números naturais:

(.+.) :: Nat -> Nat -> Nat

Zero .+. m = m -- Equação 1

(Succ n') .+. m = Succ (n' .+. m) -- Equação 2

Ex: Soma dos números 2 e 1

(Succ (Succ Zero)) .+. (Succ Zero) = -- Pela equação 2

Succ ((Succ Zero) .+. (Succ Zero)) = -- Pela equação 2

Succ (Succ (Zero .+. (Succ Zero))) = -- Pela equação 1

Succ (Succ (Succ Zero))

- Provas envolvendo funções recursivas são realizadas por indução. Casos base correspondem aos construtores do tipo de dados que não envolvem recursão enquanto o passo indutivo corresponde aos construtores do tipo de dados envolvendo recursão;
- Prove que, para qualquer número nat n: $n .+. Zero = n$

Caso base:

P(Zero) é dado por $Zero .+. Zero = Zero$

$Zero .+. Zero =$ -- Pela equação 1

Zero

Passo indutivo:

Para todo $n \in \mathbb{N}$: P(n) implica em P(Succ n)

$P(n) = n .+. Zero = n$ -- Hipótese de Indução

$P(Succ n) = (Succ n) .+. Zero = (Succ n)$

Considerando $n = Succ n'$

$(Succ n') .+. Zero =$ -- Pela equação 2

$Succ (n' .+. Zero) =$ -- Pela Hipótese de Indução

$Succ n'$

- Prove que, para quaisquer números nat n e m: $Succ (n .+. m) = n .+. (Succ m)$

Caso base:

Para P(Zero), suponha $n = Zero$ e m como um nat arbitrário

$Succ (Zero .+. m) =$ -- Pela equação 1

$\text{Succ } m = \text{--}$ Pela equação 1

$\text{Zero} .+. \text{Succ } m$

Passo indutivo:

Considerando $n = \text{Succ } n'$ e supondo m como um nat arbitrário

$\text{Succ } (n' .+. m) = n' .+. (\text{Succ } m)$ -- Hipótese de Indução

$\text{Succ } (\text{Succ } n') .+. m = \text{--}$ Pela equação 2

$\text{Succ } (\text{Succ } (n' .+. m)) = \text{--}$ Pela Hipótese de Indução

$\text{Succ } (n' .+. (\text{Succ } m)) = \text{--}$ Pela equação 2

$(\text{Succ } n') .+. (\text{Succ } m)$

Aula 18: Correção de Programas – Listas e Árvores Binárias

Provar uma propriedade para listas

- Sintaxe:
 $\text{forall } xs :: [a] . P(xs)$
- Devemos provar:
 - $P([])$, o caso base será provar a propriedade P para uma lista vazia;
 - $\text{forall } x \ xs. P(xs) \rightarrow P(x : xs)$, o passo indutivo consiste em supor que a propriedade P é verdadeira para uma lista xs (hipótese de indução), e então, provar que P continua verdadeiro caso seja inserido um elemento x na lista xs .

Exemplo de propriedade sobre listas #1 – Tamanho e Concatenação

- Sintaxe do problema:
 $\text{forall } xs \ ys. \text{length } (xs ++ ys) = \text{length } xs + \text{length } ys$
Ou seja, provar que o tamanho de duas listas xs e ys concatenadas é igual a soma do tamanho de xs com o tamanho de ys ;
- Considere as seguintes definições de concatenação e length (tamanho de uma lista):
 $(++) :: [a] \rightarrow [a] \rightarrow [a]$
 $[] ++ ys = ys$
 $(x : xs) ++ ys = x : (xs ++ ys)$

- ```
length :: [a] -> Int
length [] = 0
length (_ : xs) = 1 + length xs
```
- Prova do caso base:  $xs = []$ . Suponha  $ys :: [a]$  arbitrário ( $ys$  é uma lista arbitrária).
 

```
length ([] ++ ys) = -- definição de concatenação
length ys = -- somar o valor 0 a algo não altera o valor
0 + length ys = -- definição de length
length [] + length ys
```
  - Prova do passo indutivo:  $xs = z : zs$ . Suponha  $z :: a$ ,  $zs, ys :: [a]$  arbitrários ( $z$  é um valor do tipo  $a$ ;  $zs$  e  $ys$  são listas de valores do tipo  $a$ ) e que  $length (zs ++ ys) = length zs + length ys$ .
 

```
length ((z : zs) ++ ys) = -- definição de concatenação
length (z : (zs ++ ys)) = -- definição de length
1 + length (zs ++ ys) = -- hipótese de indução
1 + (length zs + length ys) = -- associatividade da soma
(1 + length zs) + length ys = -- definição de length
length (z : zs) + length ys
```

### Exemplo de propriedade sobre listas #2 – Map e Id

- Sintaxe do problema:
 

```
forall xs :: [a]. map id xs = xs
```

 Ou seja, provar que para toda lista  $xs$ , aplicar a função `id` na lista  $xs$  (utilizando a função `map`) é igual à própria lista  $xs$ ;
- Considere as seguintes definições de `map` e `id` (função identidade):
 

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

id :: a -> a
id x = x
```
- Prova do caso base:  $xs = []$ .
 

```
map id [] = -- definição de map
[]
```

- Prova do passo indutivo:  $xs = y : ys$ . Suponha  $y :: a$ ,  $ys :: [a]$  arbitrários e que  
 $map\ id\ ys = ys$ .  
 $map\ id\ (y : ys) = -- \text{definição de map}$   
 $id\ y : map\ id\ ys = -- \text{hipótese de indução}$   
 $id\ y : ys = -- \text{definição de id}$   
 $y : ys$

### Exemplo de propriedade sobre listas #3 – *Map Fusion*

- Sintaxe do problema:  
 $forall\ xs :: [a], f :: a \rightarrow b, g :: b \rightarrow c.$   
 $(map\ g \ .\ map\ f)\ xs = map\ (g \ .\ f)\ xs$   
 Ou seja, provar que a composição de dois *maps*  $f$  e  $g$  sobre uma mesma lista  $xs$  é igual a utilizar o *map* na composição de  $f$  e  $g$  sobre a lista  $xs$ ;
- O teorema de *Map Fusion* transforma dois caminhamentos sobre um único caminhamento, assim melhorando consideravelmente a eficiência do código;
- Considere a seguinte definição de composição de funções:  
 $(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$   
 $g \ .\ f = \lambda x \rightarrow g\ (f\ x)$
- Prova do caso base:  $xs = []$ .  
 $(map\ g \ .\ map\ f)\ [] = -- \text{definição de composição}$   
 $map\ g\ (map\ f\ []) = -- \text{definição de map}$   
 $map\ g\ [] = -- \text{definição de map}$   
 $[] = -- \text{definição de map}$   
 $map\ (g \ .\ f)\ []$
- Prova do passo indutivo:  $xs = y : ys$ . Suponha  $y :: a$ ,  $ys :: [a]$  arbitrários e que  
 $(map\ g \ .\ map\ f)\ ys = map\ (g \ .\ f)\ ys$ .  
 $(map\ g \ .\ map\ f)\ (y : ys) = -- \text{definição de composição}$   
 $map\ g\ (map\ f\ (y : ys)) = -- \text{definição de map}$   
 $map\ g\ (f\ y : map\ f\ ys) = -- \text{definição de map}$   
 $g\ (f\ y) : (map\ g\ (map\ f\ ys)) = -- \text{definição de composição}$   
 $((g \ .\ f)\ y) : (map\ g\ (map\ f\ ys)) = -- \text{definição de composição}$   
 $((g \ .\ f)\ y) : ((map\ g \ .\ map\ f)\ ys) = -- \text{hipótese de indução}$   
 $((g \ .\ f)\ y) : map\ (g \ .\ f)\ ys = -- \text{definição de map}$

```
map (g . f) (y : ys)
```

#### Exemplo de propriedade sobre listas #4 – Reverse

- Sintaxe do problema:

```
forall xs ys. reverse (xs ++ ys) = reverse ys ++ reverse xs
```

Ou seja, provar que reverter a concatenação de duas listas *xs* e *ys* é igual a concatenar o reverso da lista *ys* com o reverso da lista *xs*;

- Considere a seguinte definição ineficiente de *reverse* (reverter uma lista):

```
reverse :: [a] -> [a]
```

```
reverse [] = []
```

```
reverse (x : xs) = reverse xs ++ [x]
```

- Prova do caso base: *xs* = [].

```
reverse ([] ++ ys) = -- definição de concatenação
```

```
reverse ys = -- Propriedade de "forall ys. ys ++ [] = ys"
```

```
reverse ys ++ [] = -- definição de reverse
```

```
reverse ys ++ reverse []
```

- Prova do passo indutivo: *xs* = *z* : *zs*. Suponha *z* :: *a*, *zs* *ys* :: [*a*] arbitrários e que *reverse* (*zs* ++ *ys*) = *reverse* *ys* ++ *reverse* *zs*.

```
reverse ((z : zs) ++ ys) = -- definição de ++
```

```
reverse (z : (zs ++ ys)) = -- definição de reverse
```

```
reverse (zs ++ ys) ++ [z] = -- hipótese de indução
```

```
(reverse ys ++ reverse zs) ++ [z] = -- associatividade da concatenação
```

```
reverse ys ++ (reverse zs ++ [z]) = -- definição de reverse
```

```
reverse ys ++ (reverse (z : zs))
```

#### Exemplo de propriedade sobre listas #5 – Fold/Map Fusion

- Sintaxe do problema:

```
forall xs f g v. (foldr g v . map f) xs = foldr (g . f) v xs
```

Ou seja, provar que é possível substituir a composição de um *foldr* e um *map* sobre uma lista *xs* por um único *foldr* sobre uma lista *xs*;

- Considere a seguinte definição de *foldr*:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr _ v [] = v
```

`foldr f v (x : xs) = x `f` foldr f v xs`

- Prova do caso base: `xs = []`. Suponha `f`, `g` e `v` arbitrários.

`(foldr g v . map f) [] = -- definição de composição`

`foldr g v (map f []) = -- definição de map`

`foldr g v [] = -- definição de foldr`

`v = -- definição de foldr`

`foldr (g . f) v []`

- Prova do passo indutivo: `xs = y : ys`. Suponha `f`, `g`, `v`, `y` e `ys` arbitrários e que `(foldr g v . map f) ys = foldr (g . f) v ys`.

`(foldr g v . map f) (y : ys) = -- definição de composição`

`foldr g v (map f (y : ys)) = -- definição de map`

`foldr g v (f y : map f ys) = -- definição de foldr`

`g (f y) (foldr g v (map f ys)) = -- definição de composição`

`(g . f) y ((foldr g v . map f) ys) = -- hipótese de indução`

`(g . f) y (foldr (g . f) v ys) = -- definição de foldr`

`foldr (g . f) v (y : ys)`

### Provar uma propriedade para árvores binárias

Devemos provar:

- `P(Leaf)`, o caso base será provar a propriedade `P` para uma folha;
- `forall l r x. P(l) -> P(r) -> P(Node x l r)`, o passo indutivo consiste em supor que a propriedade `P` é verdadeira para as subárvores à esquerda (`l`) e à direita (`r`) (hipótese de indução), e então, provar que `P` continua verdadeira para o nó formado pelas duas subárvores `l` e `r` e por um valor inserido `x`.

### Exemplo de propriedade sobre árvores binárias – Height e Size

- Sintaxe do problema:

`forall t. height t <= size t`

Ou seja, provar que para toda árvore binária `t`, sua altura é sempre menor ou igual ao número de seus elementos;

- Considere as seguintes definições para o tipo de dados `Tree` que representa uma árvore binária, `size` (número de elementos de uma `Tree`) e `height` (altura de uma `Tree`):

`data Tree a`

```

= Leaf
| Node a (Tree a) (Tree a)
 deriving (Eq, Ord, Show)

```

```

size :: Tree a -> Int
size Leaf = 0
size (Node _ l r) = 1 + size l + size r

```

```

height :: Tree a -> Int
height Leaf = 0
height (Node _ l r) = 1 + max (height l) (height r)

```

- Prova do caso base:  $t = \text{Leaf}$ .

```

height Leaf = -- definição de height
0 <= -- aritmética
0 = -- definição de size
size Leaf

```

- Prova do passo indutivo:  $t = \text{Node } x \ l \ r$ . Suponha  $\text{height } l \leq \text{size } l$  e que  $\text{height } r \leq \text{size } r$ .

```

height (Node x l r) = -- definição de height
1 + max (height l) (height r) <= -- hipótese de indução
1 + max (size l) (size r) <= -- aritmética
1 + size l + size r = -- definição de size
size (Node x l r)

```

# Extra

## Funções Úteis

### Função Print















- A função `print` imprime na tela um valor qualquer a que possua uma instância `Show`:  
`print :: Show a => a -> IO ()`  
`print x = PutStrLn (show x)`

## FAQ (Frequently Asked Questions / Perguntas Frequentes)

### Como passo um arquivo como argumento de uma função?

- O arquivo em questão deve estar na mesma pasta em que o projeto a ser compilado se encontra:

ProgramacaoFuncional-Haskell > Aula13

| Nome                                                                                                | Data de modificação | Tipo               | Tamanho |
|-----------------------------------------------------------------------------------------------------|---------------------|--------------------|---------|
|  .stack-work     | 15/03/2021 16:30    | Pasta de arquivos  |         |
|  app             | 15/03/2021 15:25    | Pasta de arquivos  |         |
|  src             | 15/03/2021 14:09    | Pasta de arquivos  |         |
|  test            | 15/03/2021 14:09    | Pasta de arquivos  |         |
|  .gitignore      | 15/03/2021 14:09    | Documento de Te... | 1 KB    |
|  Aula13.cabal    | 15/03/2021 14:09    | Arquivo CABAL      | 2 KB    |
|  ChangeLog.md    | 15/03/2021 14:09    | Arquivo MD         | 1 KB    |
|  LICENSE         | 15/03/2021 14:09    | Arquivo            | 2 KB    |
|  package.yaml    | 15/03/2021 14:09    | Arquivo YAML       | 2 KB    |
|  README.md       | 15/03/2021 14:09    | Arquivo MD         | 1 KB    |
|  Setup           | 15/03/2021 14:09    | Arquivo HS         | 1 KB    |
|  stack.yaml      | 15/03/2021 14:10    | Arquivo YAML       | 3 KB    |
|  stack.yaml.lock | 15/03/2021 14:10    | Arquivo LOCK       | 1 KB    |
|  teste           | 15/03/2021 15:10    | Documento de Te... | 1 KB    |

- Basta digitar o nome do arquivo seguido de sua extensão, entre aspas duplas, para passá-lo como argumento:

```
*Aula13 Lib Paths_Aula13> estatisticas "teste.txt"
Numero de Linhas: 4
Numero de Palavras: 7
```