

Sumário

1	Aula 2	3
1.1	Tipos	3
1.2	Algebraic Data Types	4
1.3	Tuplas	4
1.4	Exercícios	5
2	Aula 3	9
2.1	Algebraic Data Types - Continuação	9
2.2	Record syntax	10
2.3	Exercícios	11
3	Aula 4	13
3.1	Currying	13
3.2	Map/Fold/Filter	14
3.3	Exercícios	17
4	Aula 5	19
4.1	Sintaxe em funções	19
4.2	Recursão	20
4.3	Type parameter	21
4.4	Exercícios	22
5	Aula 6	25
5.1	Typeclasses	25
5.2	Monóides	27
5.3	Exercícios	28
6	Aula 10	31
6.1	Monads e Applicatives	31

1 Aula 2

1.1 Tipos

A linguagem Haskell é uma linguagem de tipagem forte e estática, ou seja, toda expressão possui um tipo definido em tempo de execução. Qualquer problema com tipos, por exemplo, um parâmetro do tipo inteiro é esperado e um tipo char é passado será pego em tempo de compilação. O comando `:t` inspeciona e mostra o tipo.

```
Prelude> :t '2'  
'2' :: Char  
Prelude> :t 3.4  
3.4 :: Double
```

Funções também possuem tipo e podem ser declaradas explicitamente. É considerado uma boa prática a tipagem de funções.

```
dobroLista :: [Int] → [Int]  
dobroLista xs = [2*x | x<-xs]
```

A função acima possui como parâmetro a lista de inteiros `x` e ela devolve uma lista de inteiros contendo o dobro de cada elemento `x` contido em `xs`. Tendo em mente a descrição do função acima é possível enxergar o uso da sintaxe `[Int] → [Int]` que representa que a função possui um parâmetro do tipo lista de inteiros e esta retorna uma lista de inteiros. O tipo definido após a última `→` representa o retorno da função.

1.2 Algebraic Data Types

É possível criar novos tipos em Haskell utilizando a palavra `data`. Considere o seguinte exemplo

```
data Dia = Segunda | Terca | Quarta | Quinta | Sexta | Sabado | Domingo
```

Lê-se que o novo tipo chamará `Dia` e possuirá *value constructors* `Segunda` ou `Terca` ou `Quarta` ou `Quinta` ou `Sexta` ou `Sabado` ou `Domingo`. Nota-se que `|` representa ou para a leitura da linha. Se usarmos `:t` é possível enxergar o tipo de cada *value constructor*

```
Prelude> :t Segunda
Segunda :: Dia
Prelude> :t Quinta
Quinta :: Dia
```

Pode-se agora criar funções com o novo tipo criado

```
agenda :: Dia → String
agenda Domingo = "TV..."
agenda Sabado = "Festa"
agenda _ = "Trabalho"
```

Este exemplo toma como parâmetro um `Dia` e retorna uma `String` que representa a tarefa determinada pela agenda. Note que nesta função foi aplicada o conceito de *Pattern Matching* que é uma importante parte da sintaxe de funções do Haskell. Neste exemplo, se o `Dia` passado for `Domingo` o retorno será `"TV..."`, se for `Sabado` `"Festa!"` e qualquer outro dia que não os dois retornará `"Trabalho!"`.

1.3 Tuplas

Tuplas possuem um outro jeito de carregar muitos elementos de tipos diferentes. Diferentemente das listas, não é possível usar o operador `cons` (`:`) nem concatenar (`++`) nada a elas. Tuplas são imutáveis. O número de elementos

em uma tupla é fixo e cada local ao qual um elemento reside é chamado de coordenada.

```
Prelude> :t ('A',"ALO")
('A',"ALO") :: (Char, [Char])
```

É possível fazer funções usando tupla

```
foo :: Char → Int → (Int, String)
agenda x y = (y+9, x:[x])
```

As funções `fst` e `snd` projetam a primeira e a segunda coordenada de uma tupla respectivamente.

```
Prelude> fst ('A',"ALO")
'A'
Prelude> snd ('A',"ALO")
"ALO"
```

1.4 Exercícios

Exercício 1.1 Crie o tipo *Pergunta* com os *value constructors* *Sim* ou *Nao*. Faça as funções abaixo determinando seus tipos explicitamente

1. *pergNum*: recebe via parâmetro uma *Pergunta* e retorne 0 para *Nao* e 1 para *Sim*;
2. *listPergs*: recebe via parâmetro uma lista de *Perguntas* e retorna 0's e 1's correspondentes aos constructores contidos na lista;
3. *and'*: recebe duas *Perguntas* como parâmetro e retorna a tabela verdade do *and* lógico usando *Sim* como verdadeiro e *Nao* como falso.
4. *or'*: Idem acima, porém, deve ser usado o *ou* lógico.
5. *not'*: Idem aos anteriores, porém, usando o *not* lógico.

Exercício 1.2 Gere as listas `[1,11,121,1331,14641,161051,1771561]` e `[1,2,3,5,6,7,9,10,11,13,14,15,17,18,19,21,22,23,25,26,27,29,30,31,33,34,35,37,38,39]` usando *list comprehension*.

Exercício 1.3 *Faça o tipo `Temperatura` que pode ter valores Celsius, Farenheit ou Kelvin. Implemente as funções:*

1. *`converterCelsius`: recebe um valor `double` e uma temperatura e faz a conversão para Celsius.*
2. *`converterKelvin`: recebe um valor `double` e uma temperatura e faz a conversão para Kelvin.*
3. *`converterFarenheit`: recebe um valor `double` e uma temperatura e faz a conversão para Farenheit.*

Exercício 1.4 *Faça uma função que simule o vencedor de uma partida de pedra, papel e tesoura usando tipos criados (você não poderá usar qualquer outro tipo que não seja criado usando o `data`). Casos de empate devem ser considerados em seu tipo.*

Exercício 1.5 *Faça uma função que retorne uma string com todas as vogais maiúsculas e minúsculas eliminadas de uma string passada por parâmetro usando `list comprehension`.*

Dica: procure informações sobre a função `elem`.

Exercício 1.6 *Gere as listas*

1. `["AaBB", "AbBB", "AcBB", "AdBB", "AeBB", "AfBB", "AgBB"]`
2. `[5, 8, 11, 17, 20, 26, 29, 32, 38, 41]`
3. `[1.0, 0.5, 0.25, 0.125, 0.0625, 0.03125]`

usando `list comprehension`.

Exercício 1.7 *Sabe-se que as unidades imperiais de comprimento podem ser `Inch`, `Yard` ou `Foot` (há outras ignoradas aqui). Sabe-se que $1\text{in} = 0.0254\text{m}$, $1\text{yd} = 0.9144\text{m}$, $1\text{ft} = 0.3048$. Faça a função `converterMetros` que recebe a unidade imperial e o valor correspondente nesta unidade e retorna o valor em metros. Faça a função `converterImperial` que recebe um valor em metros e a unidade de conversão e retorna o valor convertido para a unidade desejada.*

Exercício 1.8 *Faça um novo tipo chamado `Mes` que possui como valores todos os meses do ano. Implemente*

1. a função *checaFim* que retorna o número de dias que cada mês possui (considere Fevereiro tendo 28 dias).
2. a função *prox* que recebe um mês atual e retorna o próximo mês.
3. a função *estacao* que retorna a estacao do ano de acordo com o mes e com o hemisfério. Use apenas tipos criados pela palavra *data* aqui.

Exercício 1.9 Faça uma função que receba uma *String* e retorne *True* se esta for um palíndromo, *False* caso contrário.

Exercício 1.10 Faça uma função que elimine todos os números pares, todos os ímpares múltiplos de 7 e negativos de uma lista de inteiros passado via parâmetro. Você deve retorna esta lista em ordem reversa em comparação a do parâmetro.

Exercício 1.11 Faça uma função que recebe três strings *x*, *y* e *z* como parâmetro. A função retorna uma tupla com três coordenadas contendo a ordem reversa em cada. A primeira coordenada deve conter string reversa do primeiro parâmetro e assim por diante.

Exercício 1.12 Gere as listas abaixo usando *list comprehensions*

- [1, 10, 19, 28, 37, 46, 55, 64]
- [2, 4, 8, 10, 12, 16, 18, 22, 24, 28, 30]
- ['@', 'A', 'C', 'D', 'E', 'G', 'J', 'L']

Exercício 1.13 Faça uma função, chamada *revNum*, que receba uma *String* *s* e um *Int* *n*. Esta deverá retornar as *n* primeiras letras em ordem reversa e o restante em sua ordem normal.

Exemplo:

$$\text{revNum } 4 \text{ "FATEC"} = \text{"ET AFC"} \quad (1.1)$$

Exercício 1.14 Crie o tipo de dado *Binario* que pode ser *Zero* ou *Um*. Crie também o tipo de dado *Funcao* que pode ser *Soma2*, *Maior*, *Menor* e *Mult2*. Faça a função *aplicar* que recebe uma *Funcao* e dois *Binarios* seu retorno consiste em executar a operação desejada.

Exemplo:

$$\text{aplicar Soma2 Um Um} = \text{Zero} \quad (1.2)$$

Exercício 1.15 *Faça uma função, chamada `binList`, usando `list comprehension` que recebe uma lista de Binarios (ver exercício acima) e retorna outra lista com elemento somado Um e convertido para Int.*

$$\text{binList} [\text{Um}, \text{Zero}, \text{Zero}, \text{Um}, \text{Zero}] = [0, 1, 1, 0, 1] \quad (1.3)$$

2 Aula 3

2.1 Algebraic Data Types - Continuação

Todo *value constructor* pode possuir campos.

```
data Pessoa = Fisica String Int | Juridica String
```

No exemplo acima, o tipo Pessoa possui dois *value constructors*, Fisica que possui dois campos e Juridica que possui apenas um. Inspeccionando o novo tipo no GHCi é possível notar que todo *value constructor* são na realidade funções que um valor do tipo Pessoa e os campos são parâmetros.

```
Aula3> :t Fisica
Fisica ::String → Int → Pessoa
Aula3> :t Juridica
Juridica ::String → Pessoa
```

Com a ajuda do *pattern matching* vamos criar uma função teste que recebe uma Pessoa como parâmetro e esta retornará uma tupla contendo duas Strings informando nome e idade no caso de Pessoa do tipo Fisica. Para o tipo Juridica será mostrado o nome na primeira coordenada e uma mensagem informando que não há o campo idade na segunda.

```
teste ::Pessoa → (String, String)
teste (Fisica x y) = ("Nome: "++ x, "Idade: "++ show y)
teste (Juridica x) = ("Nome: "++ x, "Não há idade")
```

A primeira parte da função usa o *pattern matching* para encaixar os parâmetros x e y dentro do padrão encontrado no tipo Pessoa String Int, logo, x possui o tipo String e y Int. A segunda parte procede da mesma forma, porém, o *value constructor* Juridica possui apenas um parâmetro String, logo, o *pattern matching* faz com que x seja String.

2.2 Record syntax

Outra forma de declarar novos tipos com parâmetros é o *record syntax* que permite escrever tais tipos usando nomes para cada parâmetro.

```
data Ponto = Ponto {xval,yval :: Double}
```

Uma observação é que o *type constructor* `Ponto` pode ter o mesmo nome do seu *value constructor*, isto não causa ambiguidade nenhuma ao compilador.

```
Aula3> Ponto 1.1 2  
Ponto {xval = 1.1, yval = 2.0}
```

Para fazer uma função que calcula a distancia do ponto a origem podemos proceder de três formas.

```
distOrig ::Ponto → Double  
distOrig (Ponto x y) = sqrt(x**2 + y**2)
```

```
distOrig ::Ponto → Double  
distOrig (Ponto {xval=x, yval=y}) = sqrt(x**2 + y**2)
```

```
distOrig ::Ponto → Double  
distOrig p = sqrt(xval p**2 + yval p**2)
```

As duas primeiras maneiras usam o *pattern matching* para encaixar os parâmetros `x` e `y` no padrão que possui o tipo `Ponto`. A última usa `xval` e `yval` que são funções de projeção também. Pode-se inspecionar o tipo de `xval` no GHCi.

```
Aula3> :t xval  
xval :: Ponto → Double
```

De maneira análoga, o tipo de `yval` pode ser inspecionado para comprovar que possuem o mesmo tipo. Em suma, o *record syntax* provê além de um nome aos campos de um *value constructor*, funções de projeção para obter o valor do campo correspondente (algo análogo aos gets usados no paradigma orientado a objetos).

2.3 Exercícios

Exercício 2.1 Faça um novo tipo chamado *Metros*, que possui um *value constructor* de mesmo nome cujos parâmetros são um *Int* que representa a dimensão e um *Double* que representa o valor da medida e outro chamado *MetragemInvalida*. Implemente as funções

1. *areaQuadrado* :: *Metros* → *Metros*: Calcula a área de um quadrado
2. *areaRet* :: *Metros* → *Metros* → *Metros*: Calcula a área de um retângulo
3. *areaCubo* :: *Metros* → *Metros*: Calcula a área de um cubo

Exemplo,

$$\text{areaQuadrado}(\text{Metros } 1 \text{ } 2.0) = \text{Metros } 2 \text{ } 4.0. \quad (2.1)$$

Use o *pattern matching* para ignorar as metragens erradas (cáclular a área de um quadrado com um lado de dimensão 4 não é válido).

Exercício 2.2 (Validação de nomes) Faça o novo tipo *Valido* que possui dois *value constructor* *Sim* e *Nao*. O *value constructor* *Sim* possui um parâmetro (campo) *String*. Implemente uma função *isNomeValido* que recebe um nome e retorna *Nao* caso a *String* seja vazia e *Sim* caso contrário.

Exercício 2.3 Refaça o exercício 3 do capítulo anterior usando *record syntax* e tipos com parâmetro (siga o exemplo da conversão de medidas SI para imperial).

Exercício 2.4 Faça o tipo *Numero*, que possui um *value constructor* *Ok* com um campo *double* e outro *value constructor* *Erro* com um campo *String*. Faça a função *dividir* que divida dois números e caso o segundo número seja 0 emita um erro (use o *pattern matching*). Exemplo,

$$\text{dividir}(\text{Numero } 6) (\text{Numero } 5) = \text{Numero } 1.2. \quad (2.2)$$

Exercício 2.5 Faça o tipo *Cripto* que possua dois *value constructors* *Mensagem* e *Cifrado* ambos com um campo *String* e um *value constructor* *Erro*. Faça as funções *encriptar* e *decriptar* seguindo cada exemplo a seguir

$$\text{encriptar}(\text{Mensagem } "FATEC") = \text{Cifrado } "GBUFD" \quad (2.3)$$

$$\text{decriptar}(\text{Cifrado } "DBTB") = \text{Mensagem } "CASA". \quad (2.4)$$

OBS: a encriptação deve empurrar cada letra a frente e a decriptação, faz o inverso, empurrando uma letra para trás. Use as funções `succ` e `pred` e também `list comprehensions`. Não é possível encriptar mensagens cifradas e decriptar mensagens.

Exercício 2.6 Faça uma função `encriptarTodos` que encripta (ou dá erro) todos os elementos de um vetor de Cripto.

Exercício 2.7 Tendo como base o exercício de conversão de medidas dado em aula, crie uma função que faça conversão de câmbio. Você deve criar o tipo `Cambio` contendo os `value constructors` `Euro`, `Real` e `Dollar`. Crie também o tipo `Moeda` que possui os campos `(val :: Double)` e `(cur :: Cambio)`. Use `record syntax` e as taxas de conversão do dia ao qual você fez o exercício (especifique o dia por comentário).

Exercício 2.8 Crie a função `converterTodosReal` que recebe uma lista de `Moedas` e retorna outra lista de `Moedas` com todos os seus elementos convertidos para `Real`. Use `list comprehension`.

Exercício 2.9 Crie a função `maxMoeda` que recebe uma lista de `Moedas` e retorna o valor máximo absoluto(sem conversão alguma) dentre os campos `val` desta lista. Exemplo,

$$\text{maxMoeda } [\text{Moeda } 3 \text{ Real}, \text{Moeda } 7 \text{ Dollar}, \text{Moeda } 2 \text{ Euro}] = 7. \quad (2.5)$$

OBS: Use a função `maximum`.

3 Aula 4

3.1 Currying

Currying é uma técnica que consiste em transformar a chamada de uma função (retorno valorado) que recebe múltiplos argumentos em uma avaliação de uma sequência de funções. Você pode fixar uma quantidade de argumentos e deixar o restante variável.

```
somarTresNum :: Int → Int → Int → Int
somarTresNum x y z = x+y+z

somarCurr = somarTresNum 4 5
```

A função `somarCurr` possui fixo os parâmetros `x` e `y` da função `somarTresNum` deixando livre para variar `z`. Portanto, podemos inspecionar o tipo de `somarCurr`

```
Aula4> :t somarCurr
somarCurr :: Int → Int.
```

Como esperado o tipo de `somarCurr` é `Int → Int`, pois, apenas uma variável foi mantida livre na definição de `somarCurr`. É interessante no tipo de `somarTresNum` podemos agrupar as últimas duas flechas usando parênteses

```
somarTresNum :: Int → Int → (Int → Int)
```

Fazendo isso, fica claro que a partir da função `somarTresNum`, se passarmos dois argumentos a ela o retorno será uma função de um parâmetro `Int` e seu retorno `Int`. Sempre que houver parênteses encobrindo flechas e tipos, consideraremos que o parâmetro ou o retorno é uma função. Em suma, sempre que uma função tiver algum parâmetro suprimido o retorno será uma função.

Esta função possuirá como parâmetro todas as variáveis livres e sua avaliação levará em conta todos os parâmetros que foram deixados fixos e os que ficaram livres.

```
Aula4> somarCurr 1
10
```

3.2 Map/Fold/Filter

A função `map` tem como objetivo aplicar uma função `f`, recebida via parâmetro, a todos os elementos de uma lista `l` também recebida via parâmetro. O retorno desta função é uma nova lista ao qual seus elementos são as saídas da função `f`.

```
Aula4> :t map
map :: (a → b) → [a] → [b]
```

O tipo de entrada da lista deve ser o mesmo tipo da entrada da função e o tipo da saída será o mesmo da saída da função. Usando o conceito de *currying* e *high-order functions* pode-se enxergar o `map` como

```
Aula4> :t map
map :: (a → b) → ([a] → [b])
```

Ao suprimir a lista, de tipo `[a]`, como parâmetro da função `map`, o retorno obtido é uma função do tipo `[a] → [b]`. Ou seja, sua função foi levantada de `a → b` para `[a] → [b]`, isto é, a função que inicialmente trabalhava com estruturas simples passa agora a trabalhar com listas.

```
Aula4> map (+2) [1..5]
[3,4,5,6,7]
```

Neste exemplo, a função `map` recebe via parâmetro a função `(+2)`, que é um *currying* da função `(+)` tendo fixo o valor 2 em seu segundo argumento, e distribui esta função a todos os elementos da lista `[1, 2, 3, 4, 5]`. Assim é efetuado, $1 + 2$, $2 + 2$, $3 + 2$, $4 + 2$ e $5 + 2$. Note que o uso do `map` se assemelha ao *list comprehension* visto no primeiro capítulo.

Na função `foldl`, uma função `f` com dois parâmetros deve ser passado para `foldl` e também um valor inicial. A função `foldl` dobra a lista começando da esquerda, isto é, a função `f` é aplicada ao valor inicial e ao primeiro elemento da lista o retorno desta mais o segundo elemento devem ser os parâmetros para a nova aplicação de `f` até acabarem os elementos da lista. Você pode pensar que a função `foldl` se comporta como um acumulador se comparada ao paradigma imperativo.

```
Aula4> :t foldl
foldl :: (b -> a -> b) -> b -> [a] -> b
```

O primeiro parâmetro (de `foldl`) deve ser uma função ao qual sua primeira entrada deve o mesmo tipo `b` que o valor inicial (segundo parâmetro de `foldl`) e sua segunda entrada com o mesmo tipo `a` dos elementos contidos na lista do terceiro parâmetro (de `foldl`).

```
Aula4> foldl (+) 0 [1..4]
10
```

A função soma `(+)` terá como parâmetro o valor inicial `0` e o primeiro valor da lista `1`. A função soma será aplicada a estes dois parâmetros resultando em `1`. O valor `1` do acúmulo `(0+1)` será um novo parâmetro junto com o valor `2` da lista e ambos sofrerão a aplicação da soma novamente resultando em `3`. Esse valor `3` juntamente com o valor `3` da lista sofrerão a aplicação de `(+)` novamente resultando em `6`. O valor `6` e o elemento `4` da lista sofrerão pela última vez a aplicação de `(+)` resultando em `10`.

```
(+) 0 1 [2,3,4]
(+) 1 2 [3,4]
(+) 3 3 [4]
(+) 6 4 []
10
```

Note que, em Haskell, `0+1` é a versão infixa de `(+) 0 1` e ambas produzem o mesmo valor `1`.

```
Aula4> foldl ( \xs x -> x:xs ) [] "FATEC"
"CETAF"
```

```
'F':[] "ATEC"
'A':F':[] "TEC"
'T':A':F':[] "EC"
'E':T':A':F':[] "C"
'C':E':T':A':F':[] ""
"CETAF"
```

Observe que $(\lambda xs\ x \rightarrow x : xs)$ é um *lambda*, que nada mais é que uma função sem nome. Esta função recebe uma lista xs e um valor x e retorna o elemento x inserido à frente da lista xs .

O `filter` é uma função que recebe uma outra função f de retorno booleano e uma lista de elementos, esta função retorna uma outra lista contendo os elementos que foram argumentos de f e que tiveram `True` como retorno.

```
Aula4> filter (>0) [-4..4]
[1,2,3,4]
```

Neste exemplo é filtrado todos os elementos maiores que zero. A função recebida (>0) se equivale ao *lambda* $\lambda x \rightarrow x > 0$ que recebe um valor x e este retorna `True` caso seja maior que zero e `False` caso contrário. Os elementos $[1, 2, 3, 4]$ são todos que satisfazem a condição da função (>0) e no caso são retornados pela função `filter`.

```
Aula4> :t filter
filter :: (a → Bool) → [a] → [a]
```

O tipo da função `filter` nos diz que a função do primeiro parâmetro precisa retornar uma expressão booleana. A entrada da função do primeiro parâmetro deve ter um tipo a que deve ser o mesmo tipos dos elementos do segundo parâmetro, de tipo $[a]$.

3.3 Exercícios

Exercício 3.1 *Faça uma função que retorne a média de um [Double] usando foldl.*

Exercício 3.2 *Faça uma função que retorne o desvio padrão de um [Double] usando foldl. O desvio padrão de um vetor é dado por $\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}}$, onde \bar{x} é a média deste vetor.*

Exercício 3.3 *Ref faça o exercício 1.15 usando map.*

Exercício 3.4 *Faça uma função que receba um [String] e retorne todos elementos palíndromos. Ver exercício 1.9.*

Exercício 3.5 *Ref faça o exercício 2.8 usando map.*

Exercício 3.6 *Usando o exercício 2.7 como base, faça uma função que retorna todas as moedas com o campo val valendo Euro.*

4 Aula 5

4.1 Sintaxe em funções

Guards são uma maneira de testar varias condições em uma função de maneira similar a um if encadeado. Por exemplo, se quisermos calcular o IMC de uma pessoa e a partir deste valor, mostrar uma mensagem na tela indicando se este está acima do peso ou não, usando *guards* é possível escrever as condições de uma maneira limpa

```
imc p a
  | p/(a*a) <= 18.5 = "Abaixo do peso"
  | p/(a*a) < 25.0 = "Peso ideal"
  | p/(a*a) <= 30 = "Acima do peso"
  | otherwise = "Obesidade"
```

Um possível tipo da função acima é $\text{Double} \rightarrow \text{Double} \rightarrow \text{Double}$. Os parâmetros p e a representam peso e altura. A expressão $p / (a * a)$ representa o cálculo do IMC. A partir deste cálculo as condições são verificadas em ordem até que alguma seja `True` e o retorno da função (mensagem) será executada. Caso a condição seja `False` a próxima condição será verificada até chegar ao `otherwise` que sempre será `True`. A cláusula *where* pode ajudar a facilitar a escrita da função de `imc`.

```
imc p a
  | valorImc <= 18.5 = "Abaixo do peso"
  | valorImc < 25.0 = "Peso ideal"
  | valorImc <= 30 = "Acima do peso"
  | otherwise = "Obesidade"
where
  valorImc = p/(a*a)
```

O uso do *where* ajuda a não escrever a expressão $p/(a * a)$ em toda condição. Note que para cada padrão do *pattern matching* é possível ter *guards* próprios.

4.2 Recursão

Recursão é um método de resolução de problemas que consiste na solução de pequenas instâncias do problema até achar a solução global. A recursão precisa de uma condição de base (de parada ou inicial) para que não se caia em loops infinitos. Em Haskell, é uma técnica fundamental para resolver problemas, pois, não há instruções de loop como `repeat` ou `for`.

```
fat n
  | n <= 1 = 1
  | otherwise = n * fat(n-1)
```

(4.1)

A expressão acima realiza o cálculo de um fatorial que é um problema recursivo. A condição base é quando temos `fat 1 = 1`. Como `fat 0 = 1` e não queremos que a função exploda com valores negativos, a condição `n <= 1` foi escolhida (por enquanto é a melhor opção). Um possível tipo para função acima é $Int \rightarrow Int$. Para se reveter a ordem de uma `String` é possível proceder de forma recursiva, como mostra o trecho de código abaixo.

```
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

Note que o tipo da função é $String \rightarrow String$. Se a função `reverse'` for chamada com um argumento `"Fatec"` obteremos `"Cetaf"`. O padrão $(x : xs)$ quebra a `String` `"Fatec"` em `'F' : "atec"`, a expressão da função `reverse'` concatena a chamada da função novamente com o argumento `"atec"` concatenando a `String` `['F']` (equivalente a `"F"`) ao final, trocando assim, a ordem da palavra. A função procede recursivamente até achar a condição de parada `reverse' [] = []`, que quer dizer que se a `String` vazia for encontrada retorne a `String` vazia, não modificando a palavra a ser revertida e encerrando o processo recursivo.

4.3 Type parameter

Type parameters é o equivalente aos generics do Java aqui no Haskell. É uma forma de se implementar o conceito de polimorfismo paramétrico, ou seja, uma função ou tipo que possui um *type parameter* poderá se comportar de maneiras diferentes dependendo do tipo que for passado via parâmetro.

```
data Coisa a = UmaCoisa a | DuasCoisas a a | ZeroCoisa
```

O tipo acima possui um *type parameter* `a` que significa que qualquer tipo poderá ser passado ao tipo `Coisa a`. O *value constructor* `DuasCoisas` carrega dois campos do tipo `a`, enquanto o *value constructor* `UmaCoisa` carrega um e o `ZeroCoisa` é apenas um *value constructor* sem campos.

```
:t (DuasCoisas "OLAMundo")
(DuasCoisas "OLAMundo") :: Coisa String
```

Se neste tentarmos inspecionar o tipo de `(DuasCoisas 7 'A')` obteremos erro de compilação, pois, `DuasCoisas` possui dois campos genéricos de mesmo tipo. Um novo tipo pode carregar vários *type parameters*. Usando a recursão, podemos também criar tipos recursivos como listas ligadas e árvores.

```
data Arvore a = Nulo |
              Leaf a |
              Branch a (Arvore a) (Arvore a) deriving Show
```

(4.2)

O tipo árvore possui três *value constructors* `Nulo` que não possui nenhum campo, `Leaf a` que possui um campo de tipo `a` e `Branch a (Arvore a) (Arvore a)` que possui um campo `a` um campo para representar o nó filho esquerdo de tipo `Arvore a` (que pode ser novamente `Nulo`, `Leaf` ou `Branch`) e um campo para representar o nó filho direito de tipo `Arvore a`. Os campos de tipo `a` representam o elemento `a` ser guardado na árvore, enquanto dos campos de tipo `Arvore a` fazem a continuação da estrutura, tanto para esquerda tanto para a direita.

A árvore da Figura 4.1 pode ser representada pela expressão

```
Branch 50 (Branch 30 (Leaf 20) (Leaf 40)) (Branch 90 Nulo (Leaf 100))
```

O percurso em ordem, por exemplo, pode ser escrito com ajuda da *recursão* e do *pattern matching*.

```
emOrdem :: Arvore a → [a]
emOrdem (Branch x l r) = emOrdem l ++ [x] ++ emOrdem r
emOrdem (Leaf x) = x
emOrdem Nulo = []
```

Note que a condição de parada da função acima se baseia no fato do *pattern matching* encontrar um Leaf ou Nulo interrompendo o processo recursivo dos Branches.

4.4 Exercícios

Exercício 4.1 Usando a estrutura de árvore, monte uma função *mapA*, que jogue uma função passada por parâmetro para todos os elementos de uma árvore. Deixe explícito o tipo desta função.

Exercício 4.2 Usando o exercício acima, some 5 a cada elemento de uma árvore de inteiros.

Exercício 4.3 Uma lista ordenada é uma lista cujos elementos são inseridos de forma ordenada (crescente). Usando o tipo

```
ListaOrd a = a ::> (ListaOrd a) | Nulo deriving Show
```

crie as funções

- *inserir* :: (Ord a) ⇒ a → ListaOrd a → ListaOrd a
- *remover* :: (Eq a) ⇒ a → ListaOrd a → ListaOrd a
- *tamanho* :: ListaOrd a → Int

A função *remover* deve buscar um elemento, se não achar a lista deve se manter intacta.

Exercício 4.4 Usando a estrutura de árvore dada em aula, faça uma função que some todos os elementos de uma árvore de números.

Exercício 4.5 Implemente os percursos pós-ordem e pré-ordem. Via comentário, faça os "testes de mesa" para os dois percursos usando as árvores:

- Da Figura 4.1.
- Raiz 15 (Raiz 11 (Folha 6) (Raiz 12 (Folha 10) Nula)) (Raiz 20 Nula (Raiz 22 (Folha 21) Nula))

Exercício 4.6 Faça uma função para inserir um elemento em uma árvore de busca binária (use a mesma estrutura dada em aula).

Exercício 4.7 Faça uma função que a partir de uma lista de elementos de tipo *a* insira, usando o exercício anterior, todos os elementos desta lista na árvore e retorne-a.

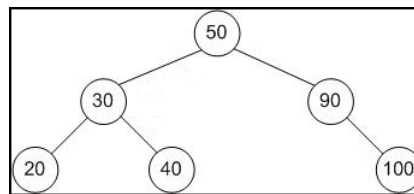


Figura 4.1: Árvore Binária

5 Aula 6

5.1 Typeclasses

Typeclass é uma estrutura do Haskell que habilita um operador(es) ou função(ões) a ser(em) usado(s) de forma(s) diferente(s) dependendo de um *type parameter*. Para cada tipo, uma instância deverá ser definida e com ela, a definição do operador ou função que o *typeclass* provê. Fazendo uma analogia com o paradigma orientado a objetos, um *typeclass* se assemelha a uma interface e o operador/função inerente a ele, se assemelha a um método abstrato. Se usarmos o tipo

```
data Coisa a = Nada | UmaCoisa a | DuasCoisas a a
```

não poderíamos mostra-lo na tela nem comparar (usando a função `(==)`). Para mostrar qualquer tipo na tela, é necessário o uso do *typeclass* `Show` e para comparar usamos o `Eq`. Reescrevendo

```
data Coisa a = Nada | UmaCoisa a | DuasCoisas a a deriving (Show, Eq)
```

temos agora que o tipo `Coisa a` pode ser mostrado na tela e é comparável. O Haskell provê para o tipo `Coisa a` a seguinte regra de igualdade: Dois valores do tipo `Coisa a` são iguais, se e somente se, seus *value constructors* são iguais e os campos também.

```
DuasCoisas 5 5 == DuasCoisas 5 5
True
```

(5.1)

```
DuasCoisas 7 3 == DuasCoisas 3 7
False
```

(5.2)

Esta regra de igualdade para o tipo em questão vem de forma gratuita apenas usando `deriving Eq` na definição. Se a regra de igualdade para este tipo for criada pelo leitor, a instância de `Eq` para o tipo `Coisa` a se faz necessária.

```
instance (Eq a) => Eq (Coisa a) where
  (DuasCoisas x1 y1) == (DuasCoisas x2 y2) = x1 == y2
  (UmaCoisa x) == (UmaCoisa y) = x==y
  Nada == Nada = True
  _ == _ = False
```

(5.3)

Note que o tipo `a` dentro da estrutura `Coisa` deve ser comparável (instância de `Eq`) como mostra o trecho `(Eq a) =>`. Se o tipo `a` não for instância de `Eq` não há como comparar os elementos de `Coisa`, logo, o tipo `Coisa a` não pode ser comparável (instância de `Eq`). Ainda no exemplo acima, mudou-se a regra de igualdade apenas para o *value constructor* `DuasCoisas` e com isso

```
DuasCoisas 7 3 == DuasCoisas 3 7
True
```

(5.4)

A operação binária diferente (`/=`) vem de graça com a negação da igualdade (`==`). É possível também criar nossa própria *typeclass*.

```
class SimNao a where
  simnao :: a -> Bool
```

(5.5)

O *typeclass* acima provê uma função `simnao` que deve ser definida para todo `a` que desejarmos criar uma instância. Pode-se criar instâncias do *typeclass* `SimNao` a para os seguintes tipos, por exemplo

```
instance SimNao Int where
  simnao 0 = False
  simnao 1 = False
  simnao _ = True
```

(5.6)

```
instance SimNao [a] where
  simnao [] = False
  simnao _ = True
```

(5.7)

que apenas diz se uma estrutura é verdadeira ou falsa com o uso do *pattern matching*.

```
simnao (1::Int)
False
simnao [1]
True
```

(5.8)

5.2 Monóides

Definição 5.1 Um Monóide é um conjunto M equipado com a operação $<>$ binária ($<>: M \times M \rightarrow M$) que satisfazem os seguintes axiomas:

1. $(a <> b) <> c = a <> (b <> c) \forall a, b, c \in M$
2. $(\exists e \in M) a <> e = a \forall a \in M$

O axioma 1 é chamado de *associatividade* e o axioma 2 de *elemento neutro*. No Haskell o *elemento neutro* é chamado de *mempty* e a operação binária ($<>$) de *mappend*. Um *Monóide* em Haskell é uma *typeclass* que deve ser importado do módulo `Data.Monoid`. Toda lista é um Monóide e este possui a instância

```
import Data.Monoid
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

(5.9)

Esta instância já está implementada no Haskell. Um `Int` (qualquer instância do *typeclass* `Num`) não possui instância de Monóide para não causar ambiguidades, pois, há duas Monóides possíveis

```
instance Monoid Int where
  mempty = 0
  mappend = (+)
```

(5.10)

```
instance Monoid Int where
  mempty = 1
  mappend = (*)
```

(5.11)

Os *typeclasses* $(Numa) \Rightarrow Suma$ e $(Numa) \Rightarrow Producta$ representam as duas Monóides acima respectivamente. As demonstrações de que listas e inteiros são *Monóides* não são escopo deste material. O operador $(< >)$ vem de graça, pois, é o mesmo que `mappend`. O uso prático de um Monóide é a criação de estruturas de dados que possuem operações binárias combináveis (como se fossem somas ou multiplicações).

5.3 Exercícios

Exercício 5.1 Usando o tipo

```
data Coisa a = Nada | UmaCoisa a | DuasCoisas a a | TresCoisas a a a
```

com sua devida instância do `Show` (não coube no retângulo azul, favor inserir o `deriving Show`), deixe-o combinável usando as regras:

1. *UmaCoisa* com *DuasCoisas* devem se combinar em *TresCoisas*;
2. *DuasCoisas* com *DuasCoisas* se combinam no mesmo *value constructor* e seus elementos se combinam em ordem (primeiro campo com primeiro campo, etc...);
3. *TresCoisas* com *TresCoisas* idem acima;
4. *UmaCoisa* com *UmaCoisa* se combina em *DuasCoisas*;
5. Controle as operações com o `mempty`;
6. Todos os outros casos serão escolha sua.

Exercício 5.2 Mostre, via comentário, 15 casos de teste do exercício acima e suas devidas respostas.

Exercício 5.3 *Faça a função*
$$\text{combinarTudo} :: [\text{Coisa } a] \rightarrow \text{Coisa } a$$

Que combina todos os elementos de uma lista de Coisas ($[\text{Coisa } a]$) em um elemento único combinado do tipo $\text{Coisa } a$

6 Aula 10

6.1 Monads e Applicatives

Exercício 6.1 Faça um programa que receba um número inteiro e mostre na tela se este é par ou ímpar usando a Monad IO.

Exercício 6.2 Faça uma instância de Functor para o tipo Coisa definido no exercício 5.1. A função deve ser aplicada em todas as coordenadas de Coisa.

Exercício 6.3 Aproveitando o exercício anterior, faça uma instância de Applicative Functor para o tipo Coisa definido no exercício 5.1. Para definir um Applicative Functor é necessário definir seu elemento neutro `pure(TresCoisas)` e seu operador `<*>`. A regra para `<*>` deve ser a distribuição das funções dos campos para o campo do argumento de forma ordenada. Por exemplo,

```
DuasCoisas (+4) (+5) <*> DuasCoisas 2 1 =  
ghci> DuasCoisas 6 6
```

 (6.1)

Cada `value constructor` deve se combinar com o mesmo, caso contrário o valor retornado é sempre Nada.

Referência: <http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

Exercício 6.4 Crie uma instância de Monad para o tipo Coisa definido no exercício 5.1. Seu return deve ser o `value constructor` UmaCoisa. Referência: <http://learnyouahaskell.com/a-fistful-of-monads>

Exercício 6.5 Crie uma função

```
mult234 :: Double → Coisa Double
```

que multiplica por 2 a primeira coordenada, por 3 a segunda e por 4 a terceira o parâmetro `x` recebido.

Exercício 6.6 *Determine o valor das expressões (caso seja possível) abaixo:*

1. *TresCoisas 1 2 3 »= mult234*
2. *(*2) <\$> DuasCoisas 2 4*
3. *:kind Coisa*
4. *DuasCoisas (*2) (*3) <*> DuasCoisas 3 4*
5. *(*2) <\$> DuasCoisas 7 9*
6. *DuasCoisas (*2) (*3) <*> UmaCoisa 5*
7. *pure (*2) <*> DuasCoisas 4 5*
8. *pure (+) <*> DuasCoisas 1 2 <*> DuasCoisas 2 1*
9. *(*2) <\$> TresCoisas 1 2 3 <*> TresCoisas 1 2 3*
10. *(*2) <\$> (TresCoisas 1 2 3 »= mult234) <*> (TresCoisas 1 2 3 »= mult234)*

Exercício 6.7 *Faça um exemplo, usando a notação do, de um trecho qualquer de código usando sua Monad Coisa.*

Exercício 6.8 *Escreva a função do exercício 6.5 em termos dos operadores Applicative.*

Exercício 6.9 *Escreva uma instância para Functor e Applicative Functor para o tipo (Arvore a) visto na Aula 4, quadro 4.2. A regra para estas instâncias são análogas (a menos de recursão).*

Exercício 6.10 *Crie 5 expressões usando o Applicative Functor de seu tipo Tree.*