Programação Funcional com a Linguagem Haskell

©André Rauber Du Bois
dubois@macs.hw.ac.uk

Índice

CAPÍTULO 1 – Programação em Haskell

1.1 Expressões e Funções

A idéia principal da linguagem Haskell é baseada na avaliação de expressões. A implementação da linguagem avalia (simplifica) a expressão passada pelo programador até sua forma normal. Por exemplo:

```
Haskell >"Alô Mundo!!"
"Alô Mundo!!"
```

Neste exemplo foi passada para o interpretador Haskell a *string* (seqüência de caracteres) "Alô Mundo!!". O sistema respondeu com a mesma seqüência de caracteres, pois esta expressão não pode mais ser avaliada, já encontra-se normalizada. Pode-se utilizar comandos mais complexos:

```
Haskell> 4 + 3
7
```

ou

```
Haskell> ((9*6)+(59/3)) *27
1989.0
```

Um comando em Haskell é uma fórmula escrita na sintaxe da linguagem.

Em Haskell existem várias funções pré-definidas que podem ser usadas para a construção de expressões:

```
Haskell> reverse "Alô Mundo!!"
"!!odnuM ôlA"
```

A função reverse inverte a ordem dos caracteres em uma *string*.

Apesar de existirem várias funções pré-definidas, a grande idéia da programação funcional é que o usuário defina as suas próprias funções. As funções do usuário são definidas em *scripts*. Um script contém definições de funções associando nomes com valores e tipos. Em scripts da linguagem Haskell também existem comentários que facilitam uma leitura posterior. Tudo o que for escrito depois de dois travessões (--) é considerado comentário e não é interpretado. Segue um exemplo de script:

```
--
--      exemplo.hs
--      Neste script apresentam-se algumas definições simples
--

idade :: Int                    -- Um valor inteiro constante
idade = 17

maiorDeIdade :: Bool            -- Usa a definição de idade
maiorDeIdade = (idade>=18)

quadrado :: Int -> Int          -- função que eleva um número ao quadrado
quadrado x = x * x

mini :: Int -> Int -> Int       -- função que mostra o menor valor entre dois inteiros
mini a b
        | a <= b        = a
        | otherwise     = b
```

A primeira linha de uma definição é a declaração de tipo. A notação '::' pode ser lida como '*possui tipo*'. Então idade tem tipo Int, que em Haskell é o tipo dos números inteiros. A linha seguinte atribui o valor 17 para idade.

Na definição seguinte é introduzido o tipo *Booleano*, que pode ter dois valores, True ou False. No caso, maiorDeIdade tem o valor False pois 17 (valor de idade) é menor do que 18. Na definição de maiorDeIdade foi utilizada a definição de idade. Em Haskell uma definição pode usar qualquer outra definição do mesmo script.

Em scripts encontra-se também definições de funções. A função quadrado no exemplo, é uma função que vai do tipo Int para o tipo Int. A função através de seu argumento calcula uma resposta utilizando uma equação (x * x) que está no lado direito da definição. Por exemplo, se passarmos para o interpretador a função quadrado e como argumento utilizarmos o valor 2 teremos:

```
Haskell> quadrado 2
4
```

A função mini devolve o menor valor entre os seus dois argumentos, que são valores do tipo Int. Para se obter a resposta, testam-se os valores para se decidir qual é o menor. Para isso são usados *guards* que são expressões booleanas iniciadas por uma barra |. No exemplo, se o valor de a é menor ou igual que b a resposta é a, senão passa-se para o próximo *guard*. Temos então a expressão otherwise, que sempre possui a resposta se todos os outros *guards* falharem. Ex:

```
Haskell > mini 2 3
2
```

Outros detalhes sobre scripts, serão apresentados no decorrer do texto.

1.2. Inteiros

O tipo Int é o tipo dos números inteiros em Haskell. Este tipo possui alguns operadores e funções:

+, *	Soma e multiplicação de inteiros
^	Potência: 2^4 é 16
-	Serve para mudar o sinal de um inteiro ou para fazer a subtração

Tabela 1. Operadores do Tipo Int

div	Divisão de números inteiros; div 10 3 é 3
mod	O resto de uma divisão de inteiros; mod 10 3 é 1
abs	Valor absoluto de um inteiro (remove o sinal).
negate	Muda o sinal de um inteiro.

Tabela 2. Funções do Tipo Int

Qualquer operador pode ser usado como função, e qualquer função pode ser usada como um operador, basta incluir o operador entre parênteses (), e a função entre crases ``.

Ex:

```
Haskell> (+) 2 3
5

Haskell> 10 `mod` 3
1
```

O programador pode definir os seus próprios operadores em scripts:

```
-- script do meu primeiro operador
(&&&) :: Int -> Int -> Int

a &&& b
        | a < b         = a
        | otherwise     = b
```

Ex:

```
Haskell> 10 &&& 3
3
```

Pode-se trabalhar com ordenação e igualdade com os números inteiros, assim como com todos os tipos básicos. As funções de ordenação e igualdade tem como argumento dois números inteiros e devolvem um valor do tipo Bool:

>	Maior que
>=	Maior ou igual
==	Igual
/=	Diferente
<=	Menor ou igual
<	Menor

Tabela 3. Ordenação e Igualdade

Ex:

```
Haskell> 29 > 15
True
```

1.3 Booleanos

O tipo Bool é o tipo dos valores booleanos True (Verdadeiro) ou False (Falso). Os operadores booleanos são:

&&	e
||	ou
not	negação

Tabela 4. Operadores Booleanos

Exemplo de definição utilizando Booleanos:

```
-- ou exclusivo
ouEx :: Bool -> Bool -> Bool
ouEx x y = (x || y) && not (x && y)
```

O *ou exclusivo* poderia ser definido utilizando *patterns* ao invés de uma fórmula:

```
ouEx True x       = not x
ouEx False x      = x
```

Este tipo de definição utiliza mais de uma equação. No exemplo, na primeira linha da definição, se for passado um valor True e um outro valor qualquer, a resposta será a negação deste valor. Se não ocorrer este caso, passa-se para a segunda linha em que se passa como argumento um valor False e um outro valor qualquer, que será a resposta.

1.4 Caracteres e Strings

O tipo Char é o tipo composto de caracteres, dígitos e caracteres especiais, como nova linha, tabulação, etc. Caracteres individuais são escritos entre aspas simples: ‘a’ é o caracter *a* e ‘7’ é o caracter sete.

Alguns caracteres especiais são representados da seguinte maneira:

‘\t’	Tabulação
‘\n’	Nova linha
‘\’ ’	Aspas simples (‘)
‘\” ’	Aspas duplas (“)
‘\\’	Barra (\)

Tabela 5. Caracteres Especiais

Os caracteres são ordenados internamente pela tabela ASCII. Por isso:

```
Haskell> 'a' < 'z'
True

Haskell> 'A'< 'a'
True
```

Pode-se utilizar a barra para representar o caracter por seu número:

```
Haskell > '\65'
'A'
```

Existem funções que transformam um número em caracter, e um caracter em número inteiro, baseando-se na tabela ASCII. Respectivamente:

```
chr :: Int -> Char
ord :: Char -> Int
```

Listas de caracteres pertencem ao tipo String, e podem ser representados entre aspas duplas:

```
"Alô Mundo!!"
"Haskell"
```

Ex:

```
Haskell> "Haskell é\nLegal !!"
"Haskell é
Legal"
```

Listas podem ser concatenadas usando o operador (++). Ex:

```
Haskell > “Preciso de” ++ “\nfrases “ ++ “melhores”
“Preciso de
frases melhores”
```

A linguagem Haskell permite que se de *sinônimos* aos nomes de tipos. Exemplo:

```
type String = [Char]
```

Isto quer dizer que o tipo String é um sinônimo de uma lista de caracteres. Ex:

```
Haskell> “Haskell” == [‘H’, ‘a’, ‘s’, ‘k’, ‘e’, ‘l’, ‘l’]
True
```

O assunto listas será analisado mais profundamente no decorrer do texto.

1.5 Números em Ponto Flutuante

Existe em Haskell o tipo Float, que trabalha com números fracionários que são representados em ponto flutuante.

Pode-se escrever os números com casas decimais ou utilizando notação científica; 231.6e-2 que significa 231.61×10^{-2}, ou simplesmente 2.3161. O tipo Float além de aceitar os operadores (+, - , *, ^, = =, /=, <=,>=, <, >) vistos anteriormente, possui algumas funções próprias:

/	Float -> Float -> Float	Divisão
**	Float -> Float -> Float	Exponenciação, x ** x = x^y
Cos, sin, tan	Float -> Float	Coseno, seno e tangente
log	Float -> Float	Logaritmo base *e*
logBase	Float -> Float -> Float	Logaritmo em qualquer base (primeiro argumento é a base)
read	String -> Float	Converte uma string representando um real, em seu valor
show	Float -> String	Converte um número para uma string
sqrt	Float -> Float	Raiz quadrada
fromInt	Int -> Float	Converte um Int para um Float
pi	Float	Constante Pi

Tabela 6. Funções do tipo Float

1.6 Tuplas

Uma tupla em Haskell é uma agregação de um ou mais componentes. Estes componentes podem ser de tipos diferentes. As tuplas são representadas em scripts por listas de componentes separados por vírgula, entre parênteses. O tipo de uma tupla parece uma tupla, mas possui tipos como componentes.

Ex:

```
-- script com tuplas

Type Nome = String          -- Sinônimo para String (Nome)

Type Idade  = Int           -- Sinônimo para Int (Idade)


verIdade :: (Nome, Idade) -> Idade        -- Função que se passa uma tupla
verIdade (a,b) = b                        -- (Nome, Idade), e devolve a idade
```

Então:

```
Haskell > verIdade (“André”, 21)
21
```

1.7 Funções Recursivas

Uma função recursiva é uma função que chama a ela mesma. Grande parte das definições em Haskell serão recursivas, principalmente as que necessitam de algum tipo de repetição. Uma definição recursiva clássica é a do fatorial de um número inteiro positivo:

O fatorial de um número inteiro positivo pode ser dividido em dois casos:

- O fatorial de 0 será sempre 1;
- E o fatorial de um número n>0, será 1 * 2 *...* (n-1) * n

Então:

```
fatorial :: Int -> Int
fatorial 0      = 1                     (regra 1)
fatorial n      = n * fatorial (n-1)    (regra 2)
```

Exemplo de avaliação:

```
fatorial 3
= 3 * (fatorial 2)              (2)
= 3 * 2 * (fatorial 1)          (2)
= 3 * 2 * 1 * (fatorial 0)      (2)
= 3 * 2 * 1 * 1                 (1)
= 6                             Multiplicação
```

Introduz-se agora um exemplo mais prático de definição recursiva. Seja a função aluno :: Int -> Float, que possui como argumento o número da chamada de um aluno (que pode variar de 1 até n), e fornece a nota do aluno na última prova como resultado.

Como se calcularia a média de notas da turma?

Para se resolver este problema, o ideal é dividi-lo em partes menores. Poderíamos primeiro pensar em uma função soma :: Int -> Float, que soma a nota de todos os alunos. Esta função teria dois casos:

- soma 1 seria a nota do aluno 1, ou simplesmente (aluno 1);
- soma n seria

aluno 1 + aluno 2 + ... + aluno (n-1) + aluno n

Tem-se então:

```
soma :: Int -> Float
soma 1        = aluno 1
soma n        = aluno n + soma (n-1)
```

Definida a função soma, pode-se definir a função média de maneira simples:

```
media :: Int -> Float
media n = (soma n) / (fromInt n)
```

Na segunda linha da definição tem-se que usar a função fromInt para transformar o valor n que tem tipo Int, em Float, pois o tipo do operador de divisão é (/) :: Float -> Float -> Float.

1.8 Exemplos

Nesta parte do texto analisa-se um exemplo mais extenso, usando as funções aluno e media explicadas anteriormente. O objetivo é criar uma função que gere uma tabela mostrando o número de todos os alunos e suas respectivas notas. No final da tabela deve aparecer a média das notas. Exemplo:

```
Haskell > tabela 4

Aluno       Nota
1           7.5
2           10
3           9
4           6.3

Média da Turma:  8.2
```

Pode–se resolver o problema utilizando uma abordagem *top-down*. A tabela pode ser pensada como sendo uma grande string. Então

```
tabela :: Int -> String
```

A função tabela tem como argumento um número inteiro (número de alunos), e devolve uma string (a tabela). A definição dessa função seria:

```
tabela n = cabeçalho ++ imprimeAlunos n ++ imprimeMedia n
```

A função cabeçalho tem uma definição direta:

```
cabeçalho :: String
cabeçalho = “Aluno       Nota\n”
```

Para se imprimir as notas, deve–se imprimir um aluno por linha. Isto pode ser definido recursivamente utilizando uma outra função

```
imprimeAluno :: Int -> String
```

Dessa maneira teremos:

```
imprimeAlunos :: Int -> String

imprimeAlunos 1     = imprimeAluno 1
imprimeAlunos n     = imprimeAlunos (n-1) ++ imprimeAluno n
```

Para a definição das funções imprimeAluno e imprimeMedia é necessário o uso da função pré-definida show, que transforma um número de qualquer tipo em string:

```
imprimeAluno :: Int -> String
imprimeAluno n = show n ++ “              “ ++ show (aluno n)  ++ “\n”

imprimeMedia :: Int -> String
imprimeMedia n = “\n” ++ “Média da Turma: “ ++ show (media n)
```

Foram usadas as funções aluno e media definidas anteriormente.

Agora apresenta-se o script completo para a função tabela:

```
-- script tabela

-- banco de dados das notas:
aluno :: Int -> Float

aluno 1 = 7.5
aluno 2 = 10
aluno 3 = 9
aluno 4 = 6.3
-- (...)
```

```
tabela :: Int -> String
tabela n = cabeçalho ++ imprimeAlunos n ++ imprimeMedia n

cabeçalho :: String
cabeçalho = “Aluno          Nota\n”


imprimeAlunos :: Int -> String
imprimeAlunos 1     = imprimeAluno 1
imprimeAlunos n     = imprimeAlunos (n-1) ++ imprimeAluno n



imprimeAluno :: Int -> String
imprimeAluno n = show n ++ “              “ ++ show (aluno n) ++ “\n”

imprimeMedia :: Int -> String
imprimeMedia n = “\n” ++ “Média da Turma: “ ++ show (media n)

soma :: Int -> Float
soma 1        = aluno 1
soma n        = aluno n + soma (n-1)

media :: Int -> Float
media n = (soma n) / (fromInt n)
```

A ordem em que as definições aparecem em um script não é importante.

É importante ressaltar que os nomes das funções sempre começam com letras minúsculas, e os tipos com letras maiúsculas.

CAPÍTULO 2 – Listas em Haskell

2.1 Listas

Em Haskell pode-se trabalhar com listas de vários tipos diferentes. Para qualquer tipo t, pode-se criar uma lista com elementos do tipo t, que será do tipo [t]. Exemplo:

```
[1, 2, 3, 4]                    :: [Int]
['H', 'a', 's', 'k', 'e', 'l', 'l']   :: [Char]
[False, True, True]             :: [Bool]
```

Pode-se trabalhar também com listas de listas, listas de tuplas e listas de funções (desde que as funções tenham o mesmo tipo):

```
[[1,2,3], [2,3], [3,4,5,6]]     :: [[Int]]
[(1,'a'), (2, 'b') , (3, 'c')]  :: [(Int, Char)]
[(/), (**), logBase]            :: [Float -> Float -> Float]
```

Um outro caso são as listas vazias, [], que não possuem elementos e podem ser de qualquer tipo:

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

A ordem e o número de ocorrência dos elementos é significante. Uma lista [3,4] é diferente de uma lista [4,3], e uma lista [1] é diferente de uma lista [1,1].

Existem outras maneiras de descrever listas:

- [a .. b] é a lista [a, a+1, ..., b]. Ex:

```
Haskell > [1 .. 6]
```

[1, 2, 3, 4, 5, 6]

Haskell > [4 .. 2]
[]

- [a, b .. c] é a lista de elementos de a até c passo b – a. Ex:

Haskell > [2,4 .. 10]
[2, 4, 6, 8, 10]

Haskell > [1,3 .. 10]
[1, 3, 5, 7, 9]

O último elemento da lista é o maior da seqüência e deve ser menor ou igual a c.

2.2 Operadores

O operador (:) é o operador de construção de listas. Toda a lista é construída através deste operador, de elementos e de uma lista.

```
[1]             = 1 : []
[1, 2, 3, 4]    = 1 : 2 : 3 : 4 : []
```

Este operador serve para todo o tipo de listas:

```
(:) :: Int -> [Int] -> [Int]
(:) :: Char -> [Char] -> [Char]
(:) :: Bool -> [Bool] -> [Bool]
(...)
```

O que se observa é que este operador trabalha com um elemento e uma lista que devem ser do mesmo tipo. Na verdade este é um operador *polimórfico* e seu tipo é:

```
(:) :: t -> [t] -> [t]
```

Onde t é uma *variável de tipo* que pode ser substituída por qualquer tipo (Int, Char, etc...). O conceito de polimorfismo será esclarecido em maior profundidade no decorrer do texto.

Outro operador para listas é o de concatenação (++):

```
Haskell> [1, 2] ++ [3, 4] ++ [5, 6]
[1, 2, 3, 4, 5, 6]
```

Apenas listas de mesmo tipo podem ser concatenadas, por isso:

```
(++) :: [t] -> [t] -> [t]
```

Aqui se usa a letra t como variável de tipo. Porém pode-se usar qualquer letra minúscula.

2.3 Funções sobre Listas

Na maioria das definições sobre listas irá se usar a recursão para se percorrer todos os elementos. Uma função simples seria a função para somar todos os elementos de uma lista de números inteiros:

```
somaLista :: [Int] -> Int
```

Para esta função existem dois casos:

- Caso Básico: Somar os elementos de uma lista vazia [] que irá resultar em 0, e

- Passo Indutivo: Somar os elementos de uma lista não vazia. Em uma lista não vazia existe sempre o elemento *head* (o primeiro elemento), e o *tail* da lista, que é a lista que sobra sem o elemento *head*. Por exemplo, a lista [1, 2, 3] tem *head* 1 e *tail* [2,3]. Uma lista com *head* a e *tail* x é escrita (a:x). Então a soma dos elementos de uma lista não vazia (a:x) é dada somando a à soma dos elementos de x.

A definição da função seria:

```
somaLista []        = 0                      (1)
somaLista (a:x)     = a + somaLista x        (2)
```

Ex:

```
Haskell> somaLista [1, 2, 3 ,4 ,5]
15
```

O comando é avaliado da seguinte maneira:

```
somaLista [1, 2, 3, 4, 5]
= 1 + somaLista [2, 3, 4, 5]                          (2)
= 1 + ( 2 + somaLista [3, 4, 5])                      (2)
= 1 + (2 + ( 3 + somaLista [4, 5]))                   (2)
= 1 + (2 + ( 3 + ( 4 + somaLista [5])))               (2)
= 1 + (2 + ( 3 + ( 4 + ( 5 + somaLista []))))         (2)
= 1 + (2 + ( 3 + ( 4 + ( 5 + 0))))                    (1)
= 15                                                  (+)
```

Uma função que teria uma definição muito parecida com a definição de somaLista, seria a função para determinar a lista cujos elementos são o dobro dos elementos de uma lista:

```
dobraLista :: [Int] -> [Int]
dobraLista []          = []
dobraLista (a:x)       = 2*a : dobraLista x
```

Quais são os dois casos desta função?

O caso básico é determinar a lista cujos elementos são o dobro dos elementos de uma lista vazia. A resposta seria [].

O passo indutivo consiste em considerar uma lista não vazia. Como se faz isso? Calcula-se o dobro do head e coloca-se este elemento como o primeiro da lista cujos elementos são o dobro dos elementos do tail.

Ex:

```
Haskell > dobraLista [1, 2, 3]
[2, 4, 6]
```

As funções apresentadas até o momento trabalham apenas com listas de um tipo específico. Porém existem funções polimórficas que trabalham com listas de qualquer tipo. Um exemplo seria a função length, pré-definida da linguagem, que dá o número de elementos de uma lista:

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

length []       = 0
length (a:x)    = 1 + length x
```

A lista vazia tem tamanho 0. A lista não vazia, possui sempre um elemento a mais que o seu tail.

Esta definição serve para qualquer tipo de listas, tanto para números quanto para caracteres, etc, por isso usa-se a variável de tipo t na declaração da função.

Um exemplo interessante que envolve recursão é o de uma função de ordenação de uma lista. O objetivo do algoritmo utilizado é inserir o primeiro elemento da lista a ser ordenada no tail da lista ordenado:

```
ordenacao :: [Int] -> [Int]

ordenacao []        = []
ordenacao (a:x)     = insere a (ordenacao x)
```

Utiliza-se para a função ordenacao uma abordagem *top-down*. Define-se a função ordenacao utilizando-se a função

```
insere :: Int -> [Int] -> [Int].
```

Inserir um elemento em uma lista vazia é simples:

```
insere e []     = [e]
```

Para se inserir um elemento no lugar certo em uma lista ordenada tem-se dois casos:

- Se o elemento a ser inserido é menor ou igual ao head da lista, coloca-se este elemento como o primeiro
- Caso contrário, insere-se o elemento no tail da lista e o head é concatenado na resposta:

```
insere e (a:x)
        | e <=a         = e:(a:x)
        | otherwise     = a : insere e x
```

Ex:

```
Haskell > ordenacao [3, 1, 2]
[1, 2, 3]
```

2.4 List Comprehensions

A *List Comprehension* é uma maneira de se descrever uma lista inspirada na notação de conjuntos. Por exemplo, se a lista list é [1, 7, 3], pode-se duplicar o valor dos elementos desta lista da seguinte maneira:

```
[ 2 * a | a < - list ]
```

que terá valor:

```
[2, 14, 6]
```

ou

```
Haskell > [2* a | a<- [1, 7, 3]]
[2, 14, 6]
```

Na *list Comprehension* o a <-list é chamado de *generator (gerador)*, pois ele gera os dados em que os resultados são construídos. Os geradores podem ser combinados com predicados (*predicates*) que são funções que devolvem valores booleanos (a->Bool).

Ex:

```
Haskell > [ a | a<-list, even a]
[]
```

No exemplo a função even devolve o valor booleano True se o seu argumento for um número par. Então esta *list comprehention* devolve apenas os valores pares da lista list. Nos geradores pode-se trabalhar com qualquer tipo de *pattern*.

Ex:

```
somaPares :: [(Int, Int)] -> [Int]
somaPares lista = [ a+b | (a,b) <- lista]
```

Ex:

```
Haskell > somaPares [(2,3), (3,7), (4,5)]
[5, 10, 9]
```

Quando se trabalha com mais de um gerador, o primeiro valor da primeira lista é gerado e mantido enquanto se avalia os valores da lista seguinte, Ex:

```
pares :: [t] -> [u] -> [(t,u)]
pares n m = [(a,b) | a <- n , b <-m]
```

Então:

```
Haskell > pares [1,2,3] [4,5]
[(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]
```

Por exemplo, utilizando *list Comprehensions* e um predicado, pode-se criar um filtro para strings:

```
remove :: Char -> [Char] -> [Char]
remove  carac str = [c | c <-str , c/= carac]
```

Exemplo:

```
Haskell > remove ‘ ‘ “Este exemplo remove os espaços em branco!”
“Esteexemploremoveosespaçosembranco!”
```

Um exemplo mais prático:

Tendo uma lista de tuplas, em que cada tupla tem-se o número do aluno, o nome do aluno e sua nota:

```
baseDeDados :: [(Int, String, Float)]
baseDeDados = [ (1, “André”, 10.0), (2, “Carlos”, 6.8), (3,”Maurício”, 7.0)]
```

Pode-se transformar esta lista em uma lista de nomes de alunos:

```
nomes :: [(Int, String, Float)] -> [String]

nomes list = [pegaNome a | a <-list]
        where
        pegaNome (a,b,c) = b
```

Na função nomes, foi usada a função pegaNome, que foi definida localmente através da palavra reservada where. Esta definição não serve para nenhuma outra função no script. Ela só funciona na função nomes.

A função nomes poderia ter sido definida de uma maneira mais simples:

```
nomes list = [b | (a,b,c) <-list]
```

2.5 Definições

A maioria das definições sobre listas se encaixam em três casos: *folding*, que é a colocação de um operador entre os elementos de uma lista, *filtering*, que significa filtrar alguns elementos e *mapping* que é a aplicação de funções a todos os elementos da lista. Os outros casos são combinações destes três, ou recursões primitivas.

Existem funções pré-definidas em Haskell que servem para resolver estes casos. São as funções foldr1, map, e filter. Estas funções são todas polimórficas, ou seja, servem para listas de qualquer tipo e são *high order functions*. As *high order functions* são funções que recebem outras funções como argumento.

- **foldr1**

Esta função coloca um operador entre os elementos de uma lista:

foldr1 (⊕) [x_1, x_2, ..., x_n] = $x_1 \oplus x_2 \oplus \ldots \oplus x_n$

A definição em Haskell é:

```
foldr1 :: (t -> t -> t) -> [t] -> t

foldr1 f  [a]            = a
foldr1 f  (a:b:x)        = f  a  (foldr1 f  (b:x))
```

A função tem como argumento um operador (ou melhor, uma função com dois argumentos), e uma lista. Ex:

```
Haskell > foldr1 (&&) [True, False, True]
False

Haskell > foldr1 (++) [“Concatenar “,“uma “,“lista “,“de ”,“strings “,“em “,“uma “,“só.”]
“Concatenar uma lista de strings em uma só.”

Haskell > foldr1 (+) [1,2,3]
6
```

Existe a função foldr que tem um argumento a mais, que seria o que deve devolver como resposta caso seja passada uma lista vazia como argumento:

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

Ex:

```
Haskell > foldr (*) 1 []
1
```

- **map**

A função map aplica uma função a todos os elementos de uma lista.

Para se aplicar f em uma lista (a:x), o head será f aplicado à a, e o tail será dado por mapear f na lista x.

A definição:

```
map :: (t -> u) -> [t] -> [u]

map f []        = []
map f (a:x)     = f a : map f x
```

Uma outra definição utilizando *list comprehension* seria:

```
map f list = [f a | a < -list]
```

Exemplo:

```
Haskell > map length [“Haskell”, “Hugs”, ”GHC”]
[7, 4, 3]
```

```
Haskell > map (2*) [1, 2, 3]
[2, 4, 6]
```

- **filter**

A função filter filtra a lista através de um predicado ou propriedade. Um predicado é uma função que tem tipo t -> Bool, como por exemplo:

```
par :: Int -> Bool
par n = (n `mod` 2 == 0)
```

A função filter é definida:

```
filter :: (t ->Bool) -> [t] -> [t]

filter p [] = []
filter p (a:x)
        | p a           = a: filter p x
        | otherwise     = filter p x
```

Exemplo:

```
Haskell > filter par [2, 4, 5, 6, 10, 11]
[2, 4, 6, 10]
```

Uma definição alternativa através de list comprehension seria:

```
filter p x = [ a | a <- x, p a]
```

Utilizando a baseDeDados definida anteriormente pode-se fazer uma função que de os nomes dos alunos com nota maior que 7.

```
alunos :: [(Int, String, Float)] -> [String]

alunos base = map pegaNome (filter nota base)
        where
        nota (a,b,c) = c>7
        pegaNome (a,b,c) = b
```

Então:

```
Haskell > alunos baseDeDados
["André"]
```

2.6 Outras Funções Úteis sobre Listas

Muitas funções de manipulação de listas necessitam tomar, ou retirar, alguns elementos de uma lista a partir do início. Para isto, a linguagem Haskell possui as seguintes funções:

```
        take :: Int -> [t] -> [t]
        drop :: Int -> [t] -> [t]
```

A função take n gera uma lista com os n primeiros elementos da lista parâmetro:

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

Então:

```
Haskell > take 3 [1, 2, 3, 4, 5, 6]
[1, 2, 3]

Haskell > take 0 [2, 4, 6, 8, 10]
[]
```

A função drop n gera uma lista sem os n primeiros elementos da lista parâmetro, sua definição é parecida com a da função take:

```
drop 0 list        = list
drop _ []          = []
drop n (a:x)       = drop (n-1) x
```

Exemplo:

```
Haskell > drop 3 [2, 4, 6, 8, 10]
[8, 10]

Haskell > drop 10 [2, 4, 6, 8, 10]
[]
```

Outras funções interessantes, que seguem o mesmo princípio, são as funções takeWhile e dropWhile, que tem como parâmetro, ao invés de um número, uma função de tipo (t -> Bool).

```
Haskell > takeWhile par [2,4,5,7, 2]
[2, 4]

Haskell > dropWhile par [2,4,5,7, 2]
[5, 7, 2]
```

Definição da função takeWhile:

```
takeWhile :: (t -> Bool) -> [t] -> [t]

takeWhile p []            = []
takeWhile p (a:x)
        | p a             = a: takeWhile p x
        | otherwise       = []
```

A dropWhile é definida de maneira semelhante.

Outra função muito utilizada é a função zip, que transforma duas listas em uma lista de tuplas.

```
zip (a:x) (b:y) = (a,b) : zip x y
zip _ _         = []

Haskell > zip [1, 3, 5] [2, 4, 6]
[(1,2), (3, 4), (5, 6)]

Haskell > zip [1, 3, 5, 7, 9, 11] [2, 4, 6]
[(1,2), (3, 4), (5, 6)]
```

A lista gerada pela função zip sempre terá o mesmo número de elementos da menor lista passada como argumento. Existe uma função pré-definida da linguagem derivada da função zip, é a função zipWith:

```
ZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Ela funciona da seguinte maneira:

ZipWith **op [**x_1, x_2, x_3, ...**] [**y_1, y_2, y_3, ...**]** = **[op** x_1 y_1, **op** x_2, y_2, **op** x_3 y_3, ...**]**

2.7 Listas Infinitas

A linguagem Haskell assim como todas as linguagens funcionais puras, são chamadas de *non-strict languages*, elas trabalham com a *lazy evaluation*, ou seja, os argumentos de funções são avaliados somente quando necessário. Se temos por exemplo uma função

$$f\,(x) = 7$$

e passamos o seguinte argumento:

$$f((21 + 33) * 8) = 7$$

é realmente necessário perder-se tempo computacional avaliando-se a expressão *(21+33)*8*, se a avaliação da função sempre gera o valor 7, independente do argumento?

Em linguagens imperativas como C e Pascal, os argumentos sempre são avaliados antes de serem passados para as funções.

A *lazy evaluation* nos permite trabalhar com estruturas infinitas. Estas estruturas necessitariam de um tempo infinito de processamento, mas na *lazy evaluation* apenas partes de uma estrutura de dados precisam ser avaliadas.

Uma das principais estruturas infinitas utilizadas são as listas.

Um exemplo simples de lista infinita seria

```
uns = 1 : uns
```

Se passarmos esta estrutura para um interpretador:

```
Haskell > uns
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ^C {Interrupted}
```

ele nos avaliaria a lista até que uma tecla de interrupção fosse usada.

Pode-se usar funções com listas infinitas:

```
somaOsDoisPrimeiros :: [Int] -> Int
somaOsDoisPrimeiros (a:b:x)      = a+b
```

Temos:

```
Haskell > somaOsDoisPrimeiros uns
2
```

A estrutura uns não precisa ser gerada por completo para que a função somaOsDoisPrimeiros seja avaliada.

Um exemplo interessante de lista infinita é a gerada pela função pré-definida iterate:

```
iterate :: (t -> t) -> t -> [t]

iterate f  x = [ x ] ++ iterate f (f x)
```

Esta função constrói uma seqüência em que o próximo elemento é o valor gerado aplicando-se uma função ao elemento anterior. Ex:

```
Haskell > iterate (+1) 1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ^C {Interrupted}
```

Pode-se definir uma função que pegue todos os valores até a posição n em uma iteração.

```
valorEmUmaIteração :: (v -> v) -> v -> v ->v
```

```
valorEmUmaIteração func inic valor = take valor (iterate func ini)
```

Então:

```
Haskell > valorEmUmaIteração (*2) 1 3
[1, 2, 4]
```

Existem outras maneiras de se descrever listas infinitas:

```
 [3 .. ] = [3, 4, 5, 6 ...

[2, 4 .. ] = [2, 4, 6, 8 ...
```

Pode–se definir então uma função que ache todas as potências de um número inteiro:

```
pot :: Int -> [Int]

pot n = [ n^y | y <- [0 ..] ]
```

Tem–se então

```
Haskell > pot 2
[ 1, 2, 4, 8 ^C {Interrupted}
```

2.8 Erros

Se for passado para a função take um valor negativo, ela ira devolver uma mensagem de erro:

```
Haskell > take (-1) [1,2]

Program error:  negative argument.
```

Existe uma função em Haskell chamada

```
error :: String -> a
```

que pára a avaliação de uma expressão caso ocorra um valor não desejado (⊥). A definição final da função take seria:

```
take                :: Int -> [a] -> [a]
take 0 _                  = []
take _ []                 = []
take n (x:xs) | n>0       = x : take (n-1) xs
take _ _                  = error "negative argument"
```

CAPÍTULO 3 – Conceitos Avançados

3.1 Currying

Em Haskell uma função de dois ou mais argumentos, pode aceitá-los um de cada vez. Isto se chama *currying*. Por exemplo:

```
soma :: Int -> Int -> Int

soma x y      =       x + y
```

Esta função pega dois números inteiros como argumento e os soma:

```
Haskell > soma 2 8
10
```

Se aplicarmos a função soma a apenas um argumento (soma 1), teremos uma função que aplicada a um argumento b, incrementa este valor (1+b).

Pode-se definir então uma função incrementa da seguinte maneira:

```
incrementa :: Int -> Int

incrementa = soma 1
```

Ex:

```
Haskell > incremeta 4
5
```

Observa-se então que uma função de dois ou mais argumentos pode ser aplicada parcialmente (*partial aplication*) formando como resultado funções:

```
soma            :: Int -> Int -> Int
soma 2          :: Int -> Int
soma 2 3        :: Int
```

Pode–se fazer definições do tipo:

```
incrementaLista :: [Int] -> Int

incrementaLista = map (soma 1)
```

Neste exemplo existem duas aplicações parciais de funções. A função soma 1 incrementa um número inteiro, e a função map (soma 1), que tem como argumento uma lista, incrementa todos os valores de uma lista de inteiros.

A aplicação parcial pode ser analisada no cálculo lambda. Considere como exemplo, uma expressão:

$$\lambda y.\ (\lambda x.\ x + y)$$

Aplicando-se um argumento:

$$\lambda y.\ (\lambda x.\ x + y)\ 3$$

obtem-se uma função com apenas um argumento:

$$\lambda x.\ x + 3$$

Os operadores da linguagem podem também ser parcialmente aplicados, o que gera os *operator sections*.

(+1)	Função que incrementa.
(1+)	Função que incrementa.
(<=100)	Função que devolve um valor booleano, True se o argumento for menor ou igual a 100 , False caso contrário.
(“Haskell“ ++)	Função que concatena a string “Haskell“ no início de outra string
(++ “Haskell”)	Função que concatena a string “Haskell” no final de uma string

Tabela 7. Operator Sections

Sendo op um operador, x e y argumentos, a regra é a seguinte:

```
(op x) y      =  y op x
(x op) y      = x op y
```

A função incrementaLista poderia ter sido definida da seguinte maneira:

```
incrementaLista = map (+1)
```

A própria função soma poderia ter sido definida simplesmente:

```
soma = (+)
```

3.2 Composição de Funções

A composição de funções é utilizada para aplicação de funções sobre funções. Isso proporciona uma maneira de dividir um problema em várias partes menores.

Por exemplo, através da função remove, definida anteriormente, pode-se definir uma função que remove pontuação (,.!) em uma string:

```
removePontuacao :: String -> String

removePontuacao str = remove '!' (remove '.' ( remove ',' str) ) )
```

Então:

```
Haskell > removePontuacao "Haskell. É muito bom, para manipular strings !!!"
"Haskell É muito bom para manipular strings "
```

Existe um operador de composição de funções em Haskell (.), que ajuda a evitar o uso de vários parênteses nas definições. A definição de removePontuacao ficaria da seguinte maneira:

```
removePontuacao  = remove '!'  .  remove '.'  .   remove ','
```

O operador de composição funciona como esta equação:

$$f\,(g\,x) = (f\,.\,g)\;\;x$$

e seu tipo é

```
(.) :: (u -> v) -> (t -> u) -> (t -> v)
```

Um exemplo interessante é a definição de iteracao, que tem como parâmetro uma função e o número de vezes que esta deve ser composta com ela mesma:

```
iteracao :: (a->a) -> Int -> (a->a)

iteracao f 1 = f
iteracao f n = iteracao f (n-1) . f
```

Tem –se :

```
Haskell> iteracao (+1) 5 1
6
```

3.3 Expressões Lambda

Ao invés de usar equações para definir funções, pode-se utilizar uma notação lambda, em que a função não precisa ter um nome. Por exemplo a função

```
sucessor :: Int -> Int

sucessor x = x+1
```

poderia ser definida como

$$\lambda x.\ x+1$$

na notação lambda, ou

```
\x -> x + 1
```

em Haskell. Temos então

```
Haskell > (\x -> x + 1) 10
11
```

Da mesma maneira a função `soma` poderia ter sido definida da seguinte maneira:

```
soma = \ x y -> x + y
```

O operador de composição de funções é definido utilizando a sintaxe lambda:

```
(.)      :: (u -> v) -> (t -> u) -> (t -> v)

f . g    = \x -> f (g x)
```

CAPÍTULO 4 – Classes de Tipo

4.1 Classes de Tipo

Observando o operador (==), nota-se que ele tem tipo:

```
(==) :: t -> t-> Bool
```

ou seja, ele é uma função polimórfica. Porém este polimorfismo é diferente do da função length. Analisando-se a definição da função length, observa-se que a mesma definição vale *para qualquer tipo de listas*:

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

length []      = 0
length (a:x)   = 1 + length x
```

Já o operador (==) tem uma definição diferente para cada tipo, pois não é o mesmo algoritmo que calcula a igualdade entre listas, caracteres ou números. Este operador também não funciona para todos os tipos, por exemplo, não existe um algoritmo que diga se uma função é igual a outra.

Em Haskell chama-se *classe* o conjunto de tipos sobre os quais uma função é definida. Por exemplo a *equality class*, ou classe Eq, é o conjunto de tipos em que o operador (==) é definido.

A classe Eq é definida da seguinte maneira:

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

Na verdade ela possui os operadores de igualdade e de diferença.

Definindo-se um tipo Endereco (a definição de tipos Algébricos será explicada detalhadamente no próximo capítulo):

```
data Endereco = Rua String Residencia

data Residencia = Casa Int | Apto Int Int
```

e avaliando-se no interpretador algo do tipo:

```
Haskell >  Rua "Farofa" (Casa 3) ==  Rua "Farinha" (Casa 3)
```

a resposta será um erro:

```
ERROR: Endereco is not an instance of class "Eq"
```

Pode-se definir então uma função que iguala endereços:

```
iguala :: Endereco -> Endereco -> Bool

iguala (Rua x (Casa y)) (Rua a (Casa b))          = (x==a) && (y == b)
iguala (Rua x (Apto y z)) (Rua a (Apto b c))      = (x==a) && (y == b) && (z==c)
iguala _ _      = False
```

Exemplo:

```
Haskell >  iguala (Rua "Abobora" (Apto 13 403)) (Rua "Abobora" (Apto 13 403))
True
```

Com a função iguala é possível instanciar o tipo Endereco na classe Eq da seguinte maneira:

```
instance Eq Endereco where
(==) = iguala
```

Depois de feita a instanciação é possível usar o operador (==) diretamente em valores do tipo Endereco:

```
Haskell > (Rua "Abobora" (Apto 13 403)) == (Rua "Abobora" (Apto 13 403))
True
```

Também pode-se usar o operador (/=), pois na classe ele é definido como:

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

Então:

```
Haskell > (Rua "Azul" (Apto 1 403)) /= (Rua "Marrom" (Casa 10))
True
```

4.2 Classes Derivadas

Uma classe pode ser derivada de outra. Dessa maneira além de ter as suas operações próprias, possui também as operações da super-classe. Um exemplo de classe derivada é a classe Ord. Ela é definida de uma maneira similar:

```
class (Eq a) => Ord a where

        (<), (<=), (>=), (>)    :: a -> a -> Bool
        max, min                :: a -> a -> a
```

Para um tipo pertencer a esta classe, ele também tem que pertencer a classe Eq. Pode-se dizer também que a classe Ord *herda* as operações da classe Eq, o que tornaria a idéia mais parecida com a da orientação a objetos.

Haskell também permite heranças múltiplas, pois uma classe pode ter mais de uma super-classe:

```
Class (Eq t, Show t) => Minhaclasse t where ...
```

4.3 Contexto

Considere a definição da função elem, que devolve um valor booleano dizendo se um elemento pertence a uma lista:

```
elem x []     = False
elem x (y:ys) = x == y || (elem x ys)
```

Como foi visto até agora, o tipo desta função poderia ser polimórfico:

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

Mas analisando-se a definição da função, nota-se que ela só funciona se o tipo a puder ser igualado com o operador (==), pois toda a definição se baseia neste operador.

Como o tipo a tem que estar instanciado na classe Eq, uma melhor definição de tipo para a função elem seria:

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

Esta definição pode ser lida como “Para cada tipo a que pertencer a classe Eq, a função elem tem tipo a -> [a] -> Bool.

A expressão Eq a é chamada de *contexto* da função e não faz parte da expressão de tipo. O contexto de uma função pode ser omitido, porém é uma boa prática de programação utiliza-lo, pois facilita a compreensão da função.

4.4 Algumas Classes Importantes

Além das classes Ord e Eq citadas anteriormente existem várias outras classes pré-definidas em Haskell. Aqui serão mostradas algumas das mais importantes:

4.4.1 Enum

É a classe dos tipos que podem ser enumerados, podendo então gerar listas como

```
Haskell > [2,4, .. 8]
[2, 4, 6, 8]
```

A lista [2,4, ..8] é descrita na função enumFromThenTo 2 4 8. Pode-se gerar listas desta maneira em qualquer tipo instanciado na classe Enum, como por exemplo Char, ou qualquer outro Tipo Algébrico definido pelo programador.

As principais funções da classe Enum são:

```
class (Ord a)  => Enum a where
        enumFrom            :: a -> [a]              -- [n..]
        enumFromThen        :: a -> a -> [a]         -- [n,m..]
        enumFromTo          :: a -> a -> [a]         -- [n..m]
        enumFromThenTo      :: a -> a -> a -> [a]    -- [n,n'..m]
```

4.4.2 Read e Show

Os tipos instanciados na classe Show, são todos os tipos que podem ser convertidos para listas de caracteres (strings). A classe Read fornece operações para transformar strings em valores de algum tipo.

As principais funções são:

```
show :: (Show t)      => t -> String

read :: (Read t)      => String -> t
```

4.4.3 Classes de Números

A classe mais geral dos números é a classe Num. Todos os números possuem algumas operações em comum, são elas:

```
class (Eq a, Show a, Eval a) => Num a where
   (+), (-), (*)        :: a -> a -> a
   negate               :: a -> a
   abs, signum          :: a -> a
   fromInteger          :: Integer -> a
   fromInt              :: Int -> a
```

Outras operações numéricas são restritas a subclasses. Por exemplo, div e mod são operações da classe Integral, o que quer dizer que somente valem para os tipos Integer e Int, que a princípio são os únicos tipos instanciados nesta classe.

Outro exemplo é o operador de divisão (/) que só vale para os tipos da classe Fractional.

Uma boa maneira para aprender classes (e Haskell em geral) é consultar o script Preludio.hs que vem junto com as distribuições da linguagem. Lá encontra-se a definição de todas as classes, além de várias funções primitivas.

CAPÍTULO 5 – Tipos Algébricos

5.1 Tipos Algébricos

Até agora foram apresentados vários tipos intrínsecos da linguagem, como valores booleanos, caracteres e números. Porém existem certos problemas computacionais que são mais difíceis de serem modelados com estes valores, como por exemplo os meses.

Pode-se definir um tipo Meses da seguinte maneira:

```
data Meses = Jan | Fev | Mar | Abr | Mai | Jun | Jul | Ago | Set | Out | Nov | Dez
```

Este tipo é um *enumerated type*. Ele é formado por doze valores que são chamados de construtores (*constructors*) de tipo. Uma definição de tipo começa sempre com a palavra data, depois vem o nome do tipo (Meses), que deve começar com letra maiúscula (note que todos os tipos em Haskell começam com letra maiúscula), e seus construtores (que também começam com letra maiúscula).

Os construtores são todos os valores que um tipo pode assumir.

Um exemplo de tipo algébrico já conhecido é o tipo Bool:

```
data Bool = True | False
```

Pode-se criar um tipo que possua vários componentes:

```
type Nome = String
type Idade = Int

data Pessoas = Pessoa Nome Idade
```

As funções que manipulam os tipos algébricos podem ser definidas *por pattern matching*:

```
mostraPessoa :: Pessoas -> String

mostraPessoa (Pessoa nom idade) = “Nome: “ ++ nom ++ “ Idade: “ ++ show idade
```

Exemplo:

```
Haskell > mostraPessoa (Pessoa “Éderson Araújo” 22)
Nome: Éderson Araújo Idade: 22
```

Um tipo pode trabalhar com valores bem diferentes. Supondo que se queira trabalhar com figuras geométricas. O tipo pode assumir o valor de um círculo ou de um retângulo. Então:

```
data Forma = Circulo Float | Retangulo Float Float
```

O valor para o círculo poderia ser o raio, e para o retângulo poderia ser base e altura.

Para se definir uma função que calcula a área de um objeto do tipo Forma pode-se trabalhar novamente com *pattern matching*:

```
area :: Forma -> Float

area (Circulo r)            = pi * r * r
area (Retangulo b a)        = b * a
```

Outro exemplo do mesmo princípio seria para se trabalhar com endereços. Algumas ruas tem nomes e outras são representadas por números. Ex:

```
data Rua = Numero Int  Residencia | Nome String Residencia
```

```
data Residencia = Casa Int | Apartamento Int Int
```

Agora pode-se definir operações que formatam o endereço usando pattern matching em cima dos construtores.

Quando um tipo é definido, algumas classes podem se instanciadas diretamente através da palavra reservada deriving:

```
data Meses = Jan | Fev | Mar | Abr | Mai | Jun | Jul | Ago | Set | Out | Nov | Dez
        deriving (Eq, Show, Enum)
```

Desta maneira, pode-se fazer coisas do tipo:

```
Haskell > Jan
Jan

Haskell > Mar == Mar
True

Haskell > [Jan .. Set]
[Jan,Fev,Mar,Abr,Mai,Jun,Jul,Ago,Set]
```

5.2 Tipos Recursivos

Os tipos algébricos podem ser também recursivos.

Um exemplo típico é o de construção de árvores:

```
data Arvore = Null | Node Int Arvore Arvore
```

Uma árvore pode ser um valor nulo ou um *node* que é composto por um valor inteiro e duas sub-árvores.

Ex:

Node 22 Null Null

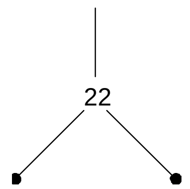

Figura 3. Árvore (1)

Node 12 (Node 1 Null Null) (Node 15 (Node 16 Null Null) Null)

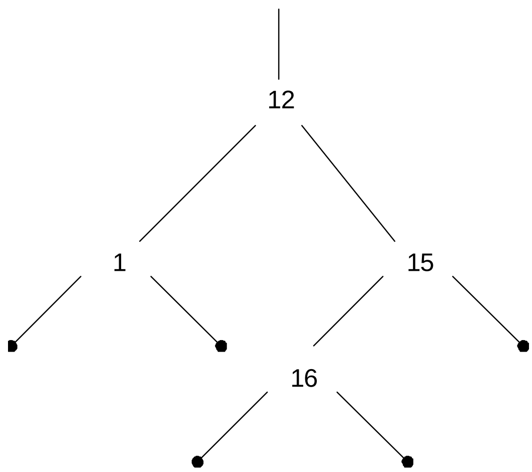

Figura 4. Árvore (2)

Pode-se fazer definições recursivas em cima de árvores, como por exemplo uma função que some todos os elementos:

```
somaArvore :: Arvore -> Int

somaArvore Null      = 0
```

```
somaArvore (Node valor esq dir)  =  valor + somaArvore (esq) + somaArvore (dir)
```

ou uma função que diz se um valor está na árvore ou não:

```
procuraValor :: Arvore -> Int ->Bool

procuraValor Null num       = False
procuraValor (Node valor esq dir) num
          | valor == num               = True
          | otherwise                  = False || (procuraValor esq num) ||
                               (procuraValor dir num)
```

Uma função que segue o mesmo princípio é a função ocorrencia que diz quantas vezes um valor aparece na árvore:

- Para uma árvore sem valores a resposta é 0.
- Se o valor do Node for igual ao valor procurado a resposta é 1 somado com o número de ocorrência do valor na árvore da esquerda, somada com a ocorrência na árvore da direita. Senão a resposta é a soma das ocorrências à direita com as da esquerda.

Tem-se então a seguinte definição:

```
ocorrencia :: Arvore -> Int -> Int

ocorrencia Null num          = 0
ocorrencia (Node valor esq dir) num
        | valor == num        = 1 + ocorrencia esq num + ocorrencia dir num
        | otherwise           = ocorrencia esq num + ocorrencia dir num
```

5.3 Tipos Algébricos Polimórficos

Os tipos Algébricos podem ser tipos com definições polimórficas. Um exemplo simples, mas ilustrativo, seria a definição de um tipo Pares. Um par poderia ser tanto dois números, quanto dois caracteres ou dois valores booleanos.

Temos:

```
data Pares u = Par u u
```

Um par seria

```
par1 :: Pares Int
par1 = Par 22
```

ou de qualquer outro tipo:

```
Par [1,2] [2,3,4]      :: Pares [Int]
Par False True         :: Pares Bool
```

(...)

A definição anterior dada para árvores trabalhava com números inteiros nos *nodes*. Pode-se modificar a definição para que a árvore contenha qualquer tipo de valor nos *nodes*. Então o tipo árvore seria:

```
data Arvore t = Null | Node t  (Arvore t) (Arvore t)
```

As definições de procuraValor e ocorrencia, precisariam ser modificadas apenas na tipagem:

```
procuraValor :: Arvore t -> Int ->Bool
```

```
ocorrencia :: Arvore t -> Int -> Int
```

Já a função somaArvore só funciona para árvores de números inteiros. Por isso o tipo passa a ser:

```
somaArvore :: Arvore Int -> Int
```

CAPÍTULO 6 – Abstract Data Types

6.1 Abstract Data Type (ADT)

Supondo que se queira modelar uma base de dados para uma loja de bebidas. Poderia-se ter a seguinte definição.

```
type Codigo = Int

type Produto = String

type Preco = Float

type Base = [(Codigo, Produto, Preco)]

base1 :: Base

base1 = [ (1, “Guarana”, 0.70), (2, “Cerveja Bacana lata”, 0.50), (3, “Uísque Do Bom”, 22.0) ......]
```

Teria-se então algumas definições em cima desta base.

```
insereProduto :: Base -> (Codigo, Produto, Preco) -> Base

retiraProduto :: Base -> Codigo -> Base

preco :: Base -> Codigo -> Preco
```

Se a base estiver ordenada pelo código do produto, fica muito mais fácil de se implementar as funções, pois todas fazem algum tipo de procura em cima da base. O único problema é que pode-se inserir um novo cadastro na base simplesmente usando o operador (:), pois a base é uma lista:

```
Haskell > (33, “Cachaça Maldição”, 0.55) : base1
[(33, “Cachaça Maldição”, 0.55), (1, “Guarana”, 0.70), (2, “Cerveja Bacana lata”,
0.50), (...)
```

Isto tornaria a base desordenada, e as funções definidas anteriormente não funcionariam mais. Para se modelar a loja de bebidas seria necessário criar um tipo que tivesse apenas as operações insereProduto, retiraProduto e preco.

Quando permitimos que um tipo funcione apenas para um certo conjunto de operações, chamamos este tipo de *abstract data type* (ADT).

6.2 Exemplo de ADT (Conjuntos)

Poderia-se implementar um conjunto como sendo uma lista de valores ordenados sem repetição. Mas para a lista manter estas qualidades seria necessário criar um ADT, para que o conjunto só fosse manipulado pelas funções a ele pertencentes.

A declaração de um ADT é feita da seguinte maneira:

```
type
        Conjunto t = [t]
        in
        vazio               :: Conjunto t,
        unitario            :: t -> Conjunto t,
        membroConj          :: Ord t => Conjunto t -> t -> Bool,
        uniao               :: Ord t => Conjunto t -> Conjunto t -> Conjunto t,
        inter               :: Ord t => Conjunto t -> Conjunto t -> Conjunto t,
        dif                 :: Ord t => Conjunto t -> Conjunto t -> Conjunto t,
```

```
conjIgual     :: Eq t => Conjunto t -> Conjunto t -> Bool,
subConj       :: Ord t => Conjunto t -> Conjunto t -> Bool,
leqConj       :: Ord t => Conjunto t -> Conjunto t -> Bool,
constConj     :: Ord t => [t] -> Conjunto t,
mapConj       :: Ord u => (t->u) -> Conjunto t -> Conjunto u,
filterConj    :: (t->Bool) -> Conjunto t -> Conjunto t,
foldConj      :: (t->t->t) -> t -> Conjunto t -> t,
mostra        :: Show t => Conjunto t -> ShowS,
card          :: Conjunto t -> Int,
conjUniao     :: Ord t => Conjunto (Conjunto t) -> Conjunto t,
conjInter     :: Ord t => Conjunto (Conjunto t) -> Conjunto t
```

Depois seguem as definições das funções.

As funções vazio e unitario são de simples definição:

```
vazio = []

unitario a = [a]
```

A função membroConj devolve um valor booleano que diz se o elemento passado como parâmetro está ou não no conjunto. Esta definição é recursiva, e utiliza a característica do conjunto ter seus elementos em ordem:

```
membroConj [] a = False
membroConj (a:x) b
        | a<b           = membroConj x b
        | a == b        = True
        | otherwise     = False
```

As funções uniao, inter e dif, que fazem respectivamente união, intersecção e diferença de dois conjuntos, possuem o mesmo tipo e definições semelhantes:

```
uniao [] a      = a
uniao a []      = a
uniao (a:x) (b:y)
        | a<b         = a : uniao x (b:y)
        | a == b      = a : uniao x y
        | otherwise   = b : uniao (a:x) y

inter [] a      = []
inter a []      = []
inter (a:x) (b:y)
        | a<b         = inter x (b:y)
        | a == b      = a : inter x y
        | otherwise   = inter (a:x) y

dif [] a = []
dif a [] = a
dif (a:x) (b:y)
        | a == b      = dif x y
        | a < b       = a : dif x (b:y)
        | otherwise   = dif (a:x) y
```

A subConj (devolve um valor booleano dizendo se um conjunto é ou não sub-conjunto de outro) é definida como as outras por *pattern matching* tendo três casos:

- Um conjunto vazio é sub-conjunto de outro
- Um conjunto não-vazio não é sub-conjunto de um conjunto vazio
- O terceiro caso é quando nenhum dos conjuntos passados como parâmetro são vazios. Então são feitas chamadas recursivas da função aproveitando o fato dos elementos estarem em ordem.

```
subConj [] a    = True
subConj x []    = False
subConj (a:x) (b:y)
        | a<b           = False
        | a == b        = subConj x y
        | a>b           = subConj (a:x) y
```

A função constConj transforma uma lista em um conjunto. Para isso se utiliza a função sort para ordenação dos elementos (função explicada anteriormente), e uma função eliminaRep, que retira os elementos repetidos da lista:

```
constConj = eliminaRep . sort

sort :: Ord t => [t] -> [t]

sort []         = []
sort (a:x)      = ins a (sort x)

ins :: Ord t => t -> [t] -> [t]

ins a []= [a]
ins a (b:y)
        | a <= b        = a : (b:y)
        | otherwise     = b : ins a y
```

```
eliminaRep :: (Ord t) => [t] -> [t]

eliminaRep [] = []
eliminaRep [a]        = [a]
eliminaRep (a:b:x)
        | a == b      = eliminaRep (b:x)
        | otherwise   = a : eliminaRep (b:x)
```

Exemplos:

```
Haskell > constConj [5, 7, 4, 3, 89, 23, 2, 3, 3, 7, 4]
[2, 3, 4, 5, 7, 23, 89]



Haskell > uniao (constConj [4, 3, 1, 22, 4]) (constConj [4, 34, 1, 3])
[1, 3, 4, 22, 34]



Haskell > inter (uniao (constConj [1,2,3]) (constConj [2,3,4,5])) (constConj [3,4,5])
[3, 4, 5]
```

Algumas definições são feitas simplesmente igualando-se uma função a outra:

```
conjIgual = (==)

leqConj = (<=)

filterConj = filter
```

```
foldConj = foldr

card = length
```

Na versão de map para conjuntos deve-se cuidar os elementos repetidos que podem aparecer:

```
mapConj f l = eliminaRep (map f l)
```

Pode-se definir as funções conjUniao e conjInter que fazem a união e intersecção de conjuntos de conjuntos, utilizando as funções foldConj, uniao e inter definidas anteriormente:

```
conjUniao = foldConj uniao []

conjInter = foldConj inter []
```

Para se poder trabalhar com os conjuntos, eles devem estar instanciados na classe show. Um conjunto pode ser exibido na tela da mesma maneira que uma lista. Por isso se usa a função showList.

```
mostra = showList

instance Show t => Show (Conjunto t) where
  showsPrec p = mostra
```

Para se facilitar algumas operações pode-se instanciar o tipo Conjunto em outras classes de tipos:

```
instance Eq t => Eq (Conjunto t) where
  (==) = conjIgual



instance Ord t => Ord (Conjunto t) where
  (<=) = leqConj
```

Existem várias outras definições que podem ser feitas sobre conjuntos. Para isso basta acrescentar a *type signature* da função na lista de funções e depois a sua definição.

Segue agora o script completo do tipo conjunto:

```
-- script de conjuntos
type
        Conjunto t = [t]
        in
        vazio           :: Conjunto t,
        unitario        :: t -> Conjunto t,
        membroConj      :: Ord t => Conjunto t -> t -> Bool,
        uniao           :: Ord t => Conjunto t -> Conjunto t -> Conjunto t,
        inter           :: Ord t => Conjunto t -> Conjunto t -> Conjunto t,
        dif             :: Ord t => Conjunto t -> Conjunto t -> Conjunto t,
        conjIgual       :: Eq t => Conjunto t -> Conjunto t -> Bool,
        subConj         :: Ord t => Conjunto t -> Conjunto t -> Bool,
        leqConj         :: Ord t => Conjunto t -> Conjunto t -> Bool,
        constConj       :: Ord t => [t] -> Conjunto t,
        mapConj         :: Ord u => (t->u) -> Conjunto t -> Conjunto u,
        filterConj      :: (t->Bool) -> Conjunto t -> Conjunto t,
        foldConj        :: (t->t->t) -> t -> Conjunto t -> t,
        mostra          :: Show t => Conjunto t -> ShowS,
        card            :: Conjunto t -> Int,
        conjUniao       :: Ord t => Conjunto (Conjunto t) -> Conjunto t,
```

```
        conjInter            :: Ord t => Conjunto (Conjunto t) -> Conjunto t
vazio = []

unitario a = [a]

membroConj [] a = False
membroConj (a:x) b
  | a<b          = membroConj x b
  | a == b       = True
  | otherwise    = False

uniao [] a       = a
uniao a []       = a
uniao (a:x) (b:y)
  | a<b          = a : uniao x (b:y)
  | a == b       = a : uniao x y
  | otherwise    = b : uniao (a:x) y

inter [] a       = []
inter a []       = []
inter (a:x) (b:y)
  | a<b          = inter x (b:y)
  | a == b       = a : inter x y
  | otherwise    = inter (a:x) y

dif [] a = []
dif a [] = a
dif (a:x) (b:y)
  | a == b       = dif x y
  | a < b        = a : dif x (b:y)
  | otherwise    = dif (a:x) y
```

```
conjIgual = (==)

subConj [] a  = True
subConj x []  = False
subConj (a:x) (b:y)
  | a<b         = False
  | a == b      = subConj x y
  | a>b         = subConj (a:x) y

leqConj = (<=)

constConj = eliminaRep . sort

sort :: Ord t => [t] -> [t]
sort []         = []
sort (a:x)      = ins a (sort x)

ins :: Ord t => t -> [t] -> [t]
ins a []= [a]
ins a (b:y)
  | a <= b      = a : (b:y)
  | otherwise   = b : ins a y

eliminaRep :: (Ord t) => [t] -> [t]
eliminaRep [] = []
eliminaRep [a]          = [a]
eliminaRep (a:b:x)
  | a == b      = eliminaRep (b:x)
  | otherwise   = a : eliminaRep (b:x)
```

```
mostra = showList

instance Eq t => Eq (Conjunto t) where
  (==) = conjIgual

instance Ord t => Ord (Conjunto t) where
  (<=) = leqConj

instance Show t => Show (Conjunto t) where
  showsPrec p = mostra

mapConj f l = eliminaRep (map f l)

filterConj = filter

foldConj = foldr

card = length

conjUniao = foldConj uniao []


conjInter = foldConj inter []
```

CAPÍTULO 7 – IO

7.1 Interação com o Usuário

Haskell, como as outras linguagens de programação, possui funções que se comunicam com o sistema operacional para realizar entrada e saída de dados. Estas operações trabalham com valores do tipo (IO t), e durante a sua avaliação requisitam operações de IO ao sistema operacional.

Ex:

```
main = putStr "Saída de dados!!!"
```

sendo que

```
putStr :: String -> IO ().
```

Se o valor devolvido por uma função for do tipo IO, o interpretador Haskell não responde simplesmente imprimindo o valor na tela, e sim mandando uma requisição ao sistema operacional para que faça a entrada ou saída de dados. Ex:

```
Haskell > main
Saída de dados !!!
```

O tipo IO é polimórfico. Se olharmos para o tipo da função getChar

```
getChar :: IO Char
```

sabe-se que ela realiza uma *ação* e retorna um caracter. Quando uma função não retorna nada de útil utiliza-se o tipo ( ). Por exemplo, a função

```
putChar :: Char -> IO ( )
```

tem como argumento um caracter, e não devolve nenhum valor. É o mesmo caso da função putStr.

Uma seqüência de entrada e saída de dados é expressa através de uma expressão do. Segue um exemplo de programa simples utilizando o do.

```
main =  do
            putStr (“Escreva uma palavra: “)
            palavra <- getLine
            putStr (“Palavra invertida: “++ reverse palavra)
```

A função getLine faz com que o sistema operacional leia uma linha, e associe esta seqüência de caracteres à variável a esquerda da flecha (<-), ou seja palavra. Esta variável só pode ser acessada dentro da expressão do. Por isso a função tem tipo:

```
getLine :: IO String.
```

O programa roda da seguinte maneira:

```
Haskell > main
Escreva uma palavra: Haskell
Palavra invertida: lleksaH

Haskell >
```

sendo que a palavra sublinhada foi a entrada do usuário.

Note que a função main possui o seguinte tipo:

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

pois ela não devolve nenhum valor, simplesmente realiza uma série de ações de entrada e saída.

7.2 Arquivos

Existem duas funções principais para se trabalhar com o sistema de arquivos:

```
writeFile :: String -> String -> IO ()

readFile :: String -> IO String
```

O funcionamento delas, pode ser facilmente demonstrado através de exemplos. Primeiro uma função para criar um arquivo contendo uma string digitada pelo usuário:

```
main = do
        putStr ("Escreva uma linha e tecle ENTER: ")
        linha <- getLine
        nome <- criaArq linha
        putStr ("A linha \n" ++ linha ++ "\nesta no arquivo " ++ nome ++ "!")

criaArq :: String -> IO String
criaArq  linha = do
                putStr ("Nome do Arquivo a ser criado: ")
                nome <- getLine
                writeFile nome linha
                return (nome)
```

A função principal pega uma linha e a usa como parâmetro para a função criaArq. Esta solicita o nome do arquivo a ser criado e o cria com a função writeFile, então devolve o nome do arquivo com a função

return :: a -> IO a.

A interação com o usuário ocorre da seguinte maneira:

```
Haskell > main
Escreva uma linha e tecle ENTER: Trabalhando com Arquivos
Nome do Arquivo a ser criado: Haskell.txt
A linha
Trabalhando com Arquivos
esta no arquivo Haskell.txt!

Haskell >
```

Pode-se fazer uma função que crie o arquivo Haskell2.txt com o conteúdo de Haskell.txt mais uma linha digitada pelo usuário:

```
adiciona = do
        putStr ("Escreva uma linha para adicionar ao arquivo Haskell2.txt:\n")
        linha <- getLine
        arquivo <- readFile "Haskell.txt"
        writeFile "Haskell2.txt" (arquivo ++ "\n" ++  linha)
        putStr ("Linha adicionada!")
```

A função readFile associa Haskell.txt a variável arquivo, isso é feito *by-need*, ou seja o conteúdo só é lido quando necessário, seguindo a estratégia da *lazy evaluation*.

7.3 Interações Infinitas

Usando a recursão pode se trabalhar com entrada de dados ilimitada.

Para ilustrar este conceito, segue um exemplo de função que lê vários nomes e os exibe em ordem alfabética:

```
main = do
          nomes <- leNomes
          putStr (unlines (sort nomes))


leNomes = do
          putStr ("Escreva um nome: ")
          nome <- getLine
          if nome == ""
               then return []
               else do
                    nomes <- leNomes
                    return ([nome] ++ nomes)

sort [] = []
sort (a:b)     = sort [x | x <- b, x<a]
          ++ [a] ++
          sort [x | x <- b, x>= a]
```

A função leNomes é chamada recursivamente até que receba uma string vazia. O controle é feito através da função (if then else), que funciona como em outras linguagens de programação. leNomes devolve uma lista de strings, onde cada elemento é um nome que foi digitado pelo usuário. Esta lista é ordenada na função principal pelo sort que utiliza um algoritmo diferente da função ordenacao vista anteriormente. O resultado é exibido depois de passar pela função

```
unlines :: [String] -> String
```

que recebe uma lista de strings, e a transforma em uma string colocando o caracter de nova linha ('\n') entre os elementos. Um exemplo do uso da função seria:

```
Haskell > main
Escreva um nome: Joao
Escreva um nome: Marcelo
Escreva um nome: Andre
Escreva um nome: Carlos
Escreva um nome:
Andre
Carlos
Joao
Marcelo

Haskell >
```

7.4 Mônadas

Apesar do sistema de I/O da linguagem Haskell parecer com programação imperativa, é puramente funcional. Ele é baseado na teoria das *Mônadas*. Mesmo assim, não é necessário se compreender a teoria das Mônadas para se programar utilizando I/O.

Os operadores de Mônadas utilizados para construir o sistema de entrada e saída da linguagem Haskell também podem ser usados para outros propósitos de programação. Mais sobre Mônadas pode ser visto em **[THO 96]**.

CAPÍTULO 7 – Construção de Módulos

7.1 Módulos

Pode-se pensar na implementação da estrutura de dados pilha, como sendo uma lista:

```
data Pilha t = Stack [t]
        deriving (Eq,Show)
```

e então definir-se algumas operações básicas sobre ela.

A função push coloca um elemento no topo da pilha:

```
push :: t -> Pilha t -> Pilha t

push x (Stack y)      = Stack (x:y)
```

a função pop retira o elemento do topo da pilha:

```
pop :: Pilha t -> t

pop (Stack [])          = error "Pilha vazia!!"
pop (Stack (a:b))              = a


pilhaVazia :: Pilha t

pilhaVazia = Stack []
```

Exemplos:

```
Haskell > push 1 pilhaVazia
Stack [1]
```

```
Haskell > pop (Stack [4,5,6])
4
```

Esta implementação de pilha pode ser reutilizada por outros programas em Haskell. Para isso é necessário criar um módulo. O módulo Pilha seria construído da seguinte maneira:

```
module Pilha ( Pilha (Stack), pilhaVazia, push, pop) where
(...)
```

Para se criar um módulo, utiliza-se a palavra reservada module, e em seguida o nome do módulo. Após o nome, lista-se todas as funções que se quer utilizar em outros programas. Logo depois vem a palavra where e as implementações.

Quando um outro programa precisar utilizar o módulo pilha, no início do script deve-se utilizar a palavra reservada import .

```
import Pilha ( Pilha (Stack), pilhaVazia, push, pop)
```

Depois de import deve-se listar as funções que se deseja utilizar do módulo. Se a lista for omitida todas as funções estarão disponíveis:

```
import Pilha.
```

7.2 Criando Um ADT

Para tornar o tipo pilha um ADT, basta não incluir na lista de funções públicas o construtor do tipo:

```
module Pilha ( Pilha , pilhaVazia, push, pop) where
(...)
```

Então:

```
Haskell > pop (Stack [1,2])
ERROR: Undefined constructor function "Stack"
```

Pode-se então criar uma função que transforme uma lista em pilha, no módulo pilha:

```
listaEmPilha:: [t] -> Pilha t
listaEmPilha x = Stack x
```

e inclui-la na lista de funções:

```
module Pilha ( Pilha , pilhaVazia, push, pop, listaEmPilha) where
```

então:

```
Haskell > pop (listaEmPilha [1,2])
1
```