Instituto Federal do Rio Grande do Norte

Haskell

Linguagem de Programação Funcional

www.cefetrn.br

Haskell

Grupo: Heitor, Lucas, Raquel, Ruth

Agenda

O que é Haskell?

Porque usar Haskell?

História do Haskel ....

Futuro do Haskell

Implementações

Passo a passo com Haskell

Programação com Haskell

O que é Haskell?

O que é Haskell

- Nomeada em homenagem ao lógico Haskell Brooks Curry, como uma base para functional languages.
- É baseada no *lambda calculus*
- Open source

O que é Haskell?

- Avançada linguagem de programação puramente funcional, polimorficamente, estaticamente tipada, lenta...
  - Linguagem Funcional:
    - programas são executados por avaliação de **expressões**; Ex. Planilha eletrônica: o valor de uma célula é especificado em função do valor de outra(s)
    - evita o uso de estado e dados mutáveis;
    - funções (first-class) são tratadas como qualquer outro valor e podem ser passadas como argumentos a outras funções ou ser retornadas como resultado de uma função. é possível definir e manipular funções alojadas em blocos de código.

Porque usar Haskell?

Porque usar Haskell?

- Rápido desenvolvimento de softwares robustos, concisos e corretos
- Forte suporte para **integração com outras linguagens**, concorrência e o paralelismo interno, depuradores de programas, profilers, bibliotecas ricas e uma comunidade ativa
- Software flexível, manutenível de alta qualidade.(http://www.haskell.org)
- A pureza de Haskell simplifica extremamente o **raciocínio sobre programas paralelos.**
  - Bibliotecas e extensões para programar aplicações simultâneas e paralelas em Haskell.

Porque usar Haskell?

- Aprender Haskell pode fazer você um programador melhor em qualquer linguagem. (http://www.haskell.org/haskellwiki/Introduction#What_is_Haskell.3F)
- Produtividade do programador substancialmente aumentada
- Código mais curto, mais claro, e mais manutenível
- Menos errors, maior confiabilidade.
- Um menor "gap semantico" entre o programador e a linguagem.
- Shorter lead times.

História do Haskell

"Being LazyWith Class"

História do Haskell

- O conceito de avaliação preguiçosa já estava difundido no meio acadêmico desde o final da década de 1970. Esforços nessa área incluíam técnicas de redução de grafo e a possibilidade de uma mudança radical na arquitetura de von Neumann.
- Conferência *Functional Programming Languages and Computer Architecture* (FPCA '87) - setembro de 1987
  - criação de um comitê com o objetivo de construir um padrão aberto para tais linguagens.

História do Haskell

- A primeira reunião do comitê - janeiro de 1988,
- metas da linguagem:
  - linguagem e fácil ensino,
  - completamente descrita através de uma sintaxe e semântica formal
  - disponível livremente.
- Primeira versão 1 de abril de 1990; versão 1.1 agosto de 1991, a versão 1.2 em março de 1992, versão 1.3 maio de 1996 versão 1.4 em abril de 1997.
- Haskell 98, janeiro de 1999 - versão mínima, estável e portável da linguagem e o biblioteca para ensino. Esse padrão sofreu uma revisão em janeiro de 2003.

História do Haskell

A linguagem continua evoluindo, sendo as implementações Hugs e GHC consideradas os padrões.

A partir de 2006 começou o processo de definição de um sucessor do padrão 98, conhecido informalmente por Haskell′ ("Haskell Prime").

Futuro do Haskell

Futuro do Haskell

- Haskell' é a próxima versão oficial
- Extensões de Haskell
- Variações de Haskell

Próxima Versão Oficial - Haskell'

A atual versão é a Haskell 98.

A construção de uma nova versão já está em desenvolvimento. Haskell' será um refinamento de Haskell 98. Irá adotar um conjunto de extensões de linguagens a fim de padronizar um conjunto de bibliotecas.

Mais informações: *hackage.haskell.org/trac/haskell-prime*

Extensões de Haskell

- **Views**: permite múltiplos construtores lógicos
- **Modelos de Guards:** generalizar guards em definições de funções.
- **Álgebra Básica:** reformula classes numéricas
- **Proposta de layout acessível**: elimina vírgulas e parênteses.

Variações de Haskell

- **GPH, Glasgow Parallel Haskell**: variação de Haskell com estruturas para diferenciar construções paralelas de seqüenciais.
- **O'Haskell**, **Haskel++**: Versões orientadas a objeto de Haskell.

Implementações

Implementações

- **GHC**. O Glasgow Haskell Compiler gera código nativo de diferentes arquiteturas e pode também gerar código C. Mais popular compilador Haskell.
- **Hugs** é um interpretador de bytecode. Oferece rápida compilação dos programas e razoável velocidade de execução. Também dispõe de uma simples biblioteca gráfica.
- **nhc98** é outro compilador que gera bytecode. O bytecode resultante executa significativamente mais rápido do que o equivalente do Hugs. Nhc98 foca na minimização do uso de memória, e é uma boa escolha para máquinas velhas/lentas.

Implementações (cont.)

- **HBC** é outro compilador Haskell para código nativo. Seu desenvolvimento não está ativo, mas ele é funcional.
- **Helium** é um novo dialecto do Haskell. O foco é na facilidade de aprendizado. Actualmente carece de *typeclasses* , tornando-o incompatível com muitos programas Haskell.

Passo a Passo com Haskell

Passo a Passo com Haskell

Passo 1 : Instalar

Passo 2 : Iniciar

Passo 3 : Escreva o primeiro programa

Passo 1: Instalar

- GHC Compilador e interpretador (GHCi). (http://www.haskell.org/ghc/)

- Hugs Unicamente interpretador. Portátil e mais leve do que o GHC.
(http://www.haskell.org/hugs/)

Passo 2 : Iniciar

1. Abra um terminal

2.
   - Se você instalou o GHC, digite **ghci**(o nome do executável do interpretador GHC) no promt de comando.

   - Se você instalou Hugs, digite **hugs**.

Passo 3: Escreva o primeiro programa

```
Prelude> "Hello, World!"
"Hello, World!"
```

O Haskell imprimiu o resultado.

>>> Você pode tentar uma variação para imprimir diretamente a saída padrão:

```
Prelude> putStrLn "Hello World"
Hello World
```

Passo 3: Escreva o primeiro programa

Usando um compilador Haskell, você pode compilar o código para para um executável *standalone* . Crie um arquivo fonte hello.hs contendo:

```
main = putStrLn "Hello, World!"
```

E compile o arquivo com:

```
$ ghc -o hello hello.hs
```

>>> Você pode então correr o executável (**./hello** no Unix **hello.exe** e no Windows):

```
$ ./hello
Hello, World!
```

Programação com Haskell

Programação com Haskell

Expressões e Funções
Inteiros
Caracteres e Strings
Números em Ponto Flutuante
Tuplas
Funções Recursivas
Listas
Tipos Algébricos
Entrada e Saída
Haskell e Prolog

Expressões e Funções

Conceitos

- 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.

Uso de expressões

- Haskell >”Ola Mundo True!!”

“Ola Mundo True!!”

- Foi passado para o interpretador uma *string* e foi retornado a mesma seqüência de caracteres, pois ela já se encontra normalizada(simplificada)

Uso de expressões

- Outras formas de utilizar expressões em Haskell:

```
Haskell> 4 + 3
7
Haskell> ((9*6)+(59/3)) *27
1989.0
```

Funções pré-definidas

- Em Haskell existem várias funções pré-definidas que podem ser empregadas para a construção de expressões
- Haskell> reverse “Ola Mundo True”

“eurt odnuM alO”

- Observamos que a função **reverse** inverte a ordem dos caracteres em uma *string*

Scripts Haskell

- Apesar de existirem várias funções pré-definidas, o fundamental da programação funcional é que o usuário defina as suas próprias funções.
- As funções são definidas através de Scripts
- Um script contém definições de funções associando nomes com valores e tipos

Scripts Haskell - Exemplo

```
--
-- 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
```

Análise do exemplo

- Tudo o que for escrito depois de dois travessões (--) é considerado comentário e não é interpretado

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

Análise do exemplo

- O “::” é usada para anotação de tipos e pode ser lido como “possui tipo”.

```
idade :: Int
maiorDeIdade :: Bool
Atribuições
idade = 17
maiorDeIdade = (idade>=18)
```

Análise do exemplo

- 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

```
quadrado :: Int -> Int
quadrado x = x * x
```

Ex.:

```
Haskell> quadrado 2
4
```

Análise do exemplo

- A função mini devolve o menor valor entre os seus dois argumentos, que são valores do tipo Int. Para obter a resposta é realizado um teste dos valores para se decidir qual é o menor.
- Para testar os valores é necessário o uso de *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.

```
mini :: Int -> Int -> Int
mini a b
| a <= b = a
| otherwise = b
```

Ex.:

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

Inteiros

- O tipo Int é o tipo dos números inteiros em Haskell.
- Podemos aplicar operadores e funções para os tipos inteiros.

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

Inteiros

- 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

Inteiros

- A linguagem Haskell permite que o programador defina os seus próprios operadores.

```
-- script do operador criado por mim
(&&&) :: Int -> Int -> Int
a &&& b
| a < b = a
| otherwise = b
```

Ex.:

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

Inteiros

- Pode-se trabalhar com ordenação e igualdade com os números inteiros, assim como 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

Uso de expressões

- Exemplos:

Haskell> 29 > 15

True

Haskell> 22 < 15

False

Caracteres e Strings

- O tipo Char é o tipo composto de caracteres, dígitos e caracteres especiais, como nova linha, tabulação,aspas simples, etc.
- Caracteres individuais são escritos entre aspas simples: ‘b’ é o caracter b e ‘3’ é o caracter três
- Exemplos de caracteres especiais:

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

Tabela 5. Caracteres Especiais

Caracteres e Strings

Os caracteres são ordenados internamente pela tabela ASCII.
Por isso:
Haskell> ‘a’ < ‘z’
True

Haskell> ‘A’< ‘a’
True

Haskell> ‘C’< ‘d’
False
•Pode-se utilizar a barra para representar o caracter por seu número:
Haskell > ‘\66’
‘B’

Caracteres e Strings

- Em Haskell podemos usar funções para transformar um número em caracter, e um caracter em número inteiro, baseando-se para isso na tabela ASCII.

  chr :: Int -> Char

  ord :: Char -> Int

- As listas de caracteres pertencem ao tipo String,podendo ser representadas por aspas duplas
- Ex.:
- “Bora meu Povo!!”
- “Se Liga na Parabula!!”

Caracteres e Strings

- Listas podem ser concatenadas usando o operador (++).
- Haskell > “Preciso de” ++ “\nfrases “ ++ “melhores”
- Resultado

  “Preciso de

  frases melhores”

Caracteres e Strings

- A linguagem Haskell permite o uso de *sinônimos aos nomes de tipos.*
- *Ex.:* type String = [Char]
- Nessa caso o tipo String é um sinônimo de uma lista de caracteres. Por exemplo:

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

True

Números em Ponto Flutuante

- Em haskell existe também o tipo Float, que trabalha com números fracionários que são representados em ponto flutuante.
- Os números podem ser escritos com casas decimais ou utilizando notação científica;231.6 e-2 que significa 231.61 ´ 10-2, ou simplesmente 2.3161.
- O tipo Float aceita os operadores já analisados anteriormente (+, - , *, ^, = =, /=, <=,>=, <, >).

Números em Ponto Flutuante

- Outras funções relacionadas ao tipo Float:

/	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

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.

Tuplas

```
-- 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
```

Exemplo prático:

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

Funções Recursivas

Grande parte das definições em Haskell serão recursivas, principalmente as que necessitam de algum tipo de repetição.

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

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

Funções Recursivas

Grande parte das definições em Haskell serão recursivas, principalmente as que necessitam de algum tipo de repetição.

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

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

Listas em Haskell

1- Listas
2- Operadores
3- Funções sobre listas
4- List Comprehensions
5- Definições
6- Outras funções úteis sobre listas
7- Listas infinitas
8- Erros

1- Listas

Para qualquer tipo t, pode-se criar uma lista com elementos do tipo t, que será do tipo [t]:

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

Pode-se, ainda trabalhar com lista de lista, lista de tuplas e lista de funções:

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

1- Listas

As listas vazias 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].

1- Listas

Existem, ainda, outras maneiras de se descrever listas:
[a .. b] é a lista [a, a+1, ..., b]. Ex:

```
Haskell > [2 .. 5]
[2, 3, 4, 5]
```

```
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]
```

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

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:

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

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

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

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

2- Operadores

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

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

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

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

```
Haskell > ['a','b']++['c']++['d','e']
"abcde"
```

3- Funções sobre listas

Notação utilizada para listas:

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)**.

3- Funções sobre listas

Soma do elementos de uma lista:

somaLista :: [Int] -> Int

Para esta função existem dois casos:

- Caso Básico: Somar os elementos de uma lista vazia [] que irá resultar em 0; (1)

- Passo Indutivo: Somar os elementos de uma lista não vazia. Então a soma dos elementos de uma lista não vazia (a:x) é dada somando a à soma dos elementos de x. (2)

3- Funções sobre listas

Soma do elementos de uma lista:

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

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

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)

3- Funções sobre listas

Soma do elementos de uma lista:

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

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

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

3- Funções sobre listas

Dobrar elementos de uma lista:

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

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

3- Funções sobre listas

Determinar o tamanho de uma lista:

Função **polimórfica**:
length :: [t] -> Int

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

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

3- Funções sobre listas

Ordenando elementos de uma lista:

Para ordenar utiliza-se uma abordagem top-down.

```
ordenacao :: [Int] -> [Int]
ordenacao []        = []
ordenacao (a:x)     = insere a (ordenacao x)

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

Inserir um elemento em uma lista vazia é simples:

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

3- Funções sobre listas

Ordenando elementos de uma lista:

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
```

3- Funções sobre listas

Ordenando elementos de uma lista:

```
ordenacao :: [Int] -> [Int]
ordenacao []        = []
ordenacao (a:x)     = insere a (ordenacao x)

insere :: Int -> [Int] -> [Int]
insere e [] = [e]
insere e (a:x)
    | e <=a     = e:(a:x)
    | otherwise = a : insere e x
```

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

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]

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

4- List Comprehensions

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. even retorna true se seu argumento for par:

```
Haskell > [a | a<- [1, 8], even a]
[8]
Haskell > [a+b | a<- [1, 8, 3], b <- [2,7,4]]
[3,8,5,10,15,12,5,10,7]
Haskell > [a+b | a<- [1, 8, 3], b <- [2,7,4], even a, even b]
[10,15,12]
```

4- List Comprehensions

Filtrando strings:

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

```
Haskell > remove ‘ ‘ “Remove os espaços em branco!”
"Removeosespaçosembranco!”
```

4- List Comprehensions

Exemplo Prático:

```
nomes :: [(Int, String, Float)] -> [String]
nomes list = [pegaNome a | a <-list]
             where pegaNome (a,b,c) = b
nomes list = [b | (a,b,c) <-list]
```

```
Haskell > nomes [ (1, "Ana", 10.0), (2, "Jose", 6.8)]
["Ana","Jose"]
```

4- 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 (**foldr1**),

filtering, que significa filtrar alguns elementos (**map**) e

mapping que é a aplicação de funções a todos os elementos da lista (**filter**).

Os outros casos são combinações destes três, ou recursões primitivas.

Essas são **high order functions**: funções que recebem outras funções como argumento.

4- Definições

foldr1: Esta função coloca um operador entre os elementos de uma lista:

foldr1 (⊕) [x1, x2, ..., xn] = x1 ⊕ x2 ⊕ ... ⊕ xn

```
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
```

4- Definições

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

```
map :: (t -> u) -> [t] -> [u]
map f [] = []
f (a:x) = f a : map f x
```

Definição utilizando **list comprehension**:

```
map f list = [f a | a < -list]
Haskell > map length [“Haskell”, “Hugs”, ”GHC”]
[7, 4, 3]
Haskell > map (2*) [1, 2, 3]
[2, 4, 6]
```

4- Definições

map: A função filter filtra a lista através de um predicado ou propriedade (t -> Bool):

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

Definição utilizando **list comprehension**:

```
filter p x = [ a | a <- x, p a]
par :: Int -> Bool
par n = (n `mod` 2 == 0)
Haskell > filter par [2, 4, 5, 6, 10, 11]
[2, 4, 6, 10]
```

6- Outras funções úteis sobre listas

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
```

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

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

6- Outras funções úteis sobre listas

A função **drop n** gera uma lista sem os n primeiros elementos da lista parâmetro:

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

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

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

6- Outras funções úteis sobre listas

takeWhile e **dropWhile**:

```
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.

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

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

7- Listas Infinitas

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.

Ex: uns = 1 : uns

```
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 ^C {Interrupted}
```

7- Listas Infinitas

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)
```

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

Haskell > iterate (+2) 2
[2, 4, 6, 8, 10, 12, 14 ^C {Interrupted}
```

7- Listas Infinitas

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 ..] ]
```

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

Tipos Algébricos

- *Representar problemas computacionais mais complexos*
- *representar um tipo cujos elementos podem ser um inteiro ou uma string*
- *representar o tipo árvore*

Exemplo

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

Entrada e Saída Padrão

- Visualização de resultado no dispositivo de saída padrão

```
module Main where
main = putStrLn "Hello World!"
```

Entrada e Saída em arquivos

```
type File = String
writeFile :: File -> String -> IO ()
appendFile :: File -> String -> IO ()
readFile :: File -> IO String
```

Entrada e Saída em arquivos

- A função **writeFile** cria um novo arquivo, com o nome especificado pelo primeiro argumento e escreve nesse arquivo a string passada como segundo argumento

Entrada e Saída em arquivos

- A função **appendFile**, ao invés de reescrever o
- conteúdo do arquivo, simplesmente grava no final do mesmo a string passada como argumento.

Entrada e Saída em arquivos

- A função **readFile** lê o conteúdo do arquivo, retornando-o como um string.

Haskell e Prolog

Haskell	Prolog
membro:: (Eq a) => a->[a]->Bool **membro** x [] = False **membro** x (y:ys) = x == y || membro x ys **concat**:: [a] -> [a] -> [a] **concat** [] ys = ys **concat** (x:xs) ys = x : concat xs ys	**membro**(X,[X|_]). **membro**(X,[_|Y]):-member(X,Y). **concat**([],L,L). **concat**([X|L1],L2,[X|L3]):-concat(L1, L2,L3).

Bibliografia

http://pt.wikipedia.org/wiki/Haskell_(linguagem_de_programa%C3%A7%C3%A3o)
http://www.haskell.org/
http://learnyouahaskell.com/
http://homepages.dcc.ufmg.br/~lucilia/cursos/func/
http://www.macs.hw.ac.uk/~dubois/ProgramacaoHaskell.pdf
http://www.haskell.org/tutorial/haskell-98-tutorial.pdf