Haskell

Aula 03

Haskell
Tipos Abstratos de Dados

- Um Tipo Abstrato de Dados, é um tipo que aceita somente um conjunto definido de operações.
- Por exemplo uma lista que seja associada ao movimento financeiro de um banco, não pode simplesmente sumir com um valor, ele deve ser retirado de um lugar e adicionado em outro.
- Se a estrutura de dados fosse implementada em uma lista, poderíamos intencionalmente ou não, alterar um valor e ele se perder.
- Com TADs evitamos a retirada ou deposito indevido pela não definição de um método que permita esta alteração direta de valores, somente sendo possibilitado por uma transação de transferência.

Haskell
TADs - Motivação

- type Nome = [String]
- type Curso = String
- type Aluno = (Matricula, Nome, Curso)
- type Base = [Aluno]

- tamanhoLista::[a]->Int
- tamanhoLista [a]

- aluno1 = (001, "Wagner Al-Alam", "CC")
- aluno2 = (002, "Joao Silva", "CC")
- aluno3 = (003, "Luciano Huck", "Jornalismo")

- base1 = [aluno1, aluno2]

Haskell
TADs - Motivação

- Método para validar entrada

```
verificaMatricula::Aluno->Aluno->Bool
verificaMatricula (a, b, c) (d, e, f)
   |a == d = False
   |otherwise = True

insereAluno::Base->Aluno->Base
insereAluno base aluno
   |foldr1(&&) (map(verificaMatricula aluno) base) = aluno:base
   |otherwise = base
```

Haskell
TADs - Motivação

- Main> base1
- [(1,"Wagner Al-Alam","CC"),(2,"Joao Silva","CC")]

- Main> insereAluno base1 aluno2
- [(1,"Wagner Al-Alam","CC"),(2,"Joao Silva","CC")]

- Main> insereAluno base1 aluno3
- [(3,"Luciano Huck","Jornalismo"),(1,"Wagner Al-Alam","CC"),(2,"Joao Silva","CC")]

- Main> aluno1:base1
- [(1,"Wagner Al-Alam","CC"),(1,"Wagner Al-Alam","CC"),(2,"Joao Silva","CC")]

Haskell
TADs

- Para transformar o exemplo anterior teríamos que alterar a definição para que somente os métodos insereAluno e removeAluno, alterem a base de dados, assim impossibilitando que insiram um aluno repetido pelo acesso direto a uma lista.
- Isto é possível, se a função (:) não estiver disponível na lista em questão.

Haskell
Módulos

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

push :: t -> Pilha t -> Pilha t
push x (Stack y) = Stack (x:y)

pop :: Pilha t -> t
pop (Stack []) = error “Pilha vazia!!”
pop (Stack (a:b)) = a

pilhaVazia :: Pilha t
pilhaVazia = Stack []
```

Haskell
Módulos

- 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.
```

Haskell
TAD Pilha

- Se na lista de funções, não importarmos o construtor do tipo, esta definição de pilha torna-se um TAD.

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

  (...)

- Logo:

  Haskell > pop (Stack [1,2])

  ERROR: Undefined constructor function "Stack“

- Entao devemos criar uma função que transforme uma lista em uma pilha:

  listaEmPilha:: [t] -> Pilha t

  listaEmPilha x = Stack x

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

Haskell
TADs

```
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
```

Haskell
TADs

```
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
```

Haskell
TADs

```
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
```

Haskell
IO

- Funções que se comunicam diretamente com o sistema operacional para executar entrada ou saída de dados são chamadas de IO e são do tipo (IO t).
  - Leitura de Teclado
  - Escrita na Tela
  - Leitura de Arquivo
  - Escrita em Arquivo

- Se o valor de retorno for um IO, no haskell não será simplesmente impresso na tela, será enviada uma requisição ao SO.

Haskell
IO – Escrita na tela

- main = putStr "Saída de dados!!“

  Main> main

  Saída de dados!!

- Neste caso o retorno não foi simplesmente impresso na tela pelo interpretador e sim pelo SO através de uma requisição de IO.
- O tipo IO é polimórfico.
- Uma seqüência de entrada e saída de dados é expressa através de uma expressão do.

`putChar :: Char -> IO ()`	Escreve um caracter
`putStr :: String -> IO ()`	Escreve uma sequência de caracteres
`putStrLn :: String -> IO ()`	Escreve uma sequência de caracteres e muda de linha
`print :: Show a => a -> IO ()`	Escreve um valor.

Haskell
IO – Leitura do teclado

`getChar :: IO Char`	Lê um caracter
`getLine :: IO String`	Lê uma linha e converte-a numa só sequência de caracteres
`getContents :: IO String`	Lê todo o conteúdo da entrada e converte-a numa só sequência de caracteres
`interact :: (String -> String) -> IO ()`	recebe uma função de sequências de caracteres para sequências de caracteres como argumento. Todo o conteúdo da entrada é passado como argumento para essa função, e o resultado dessa aplicação é visualizado.
`readIO :: Read a => String -> IO a`	Lê uma sequência de caracteres.
`readLine :: Read a -> IO a`	Lê uma sequência de caracteres e muda de linha.

Haskell
IO – Leitura e escrita na tela

```
relay::IO()  --função sem retorno
relay = do
   putStr "Digite uma linha: "
   a<-getLine
   putStr a

Main> relay
Digite uma linha: Isto é um teste de IO em Haskell
Isto é um teste de IO em Haskell
```

Haskell
IO – Leitura e escrita na tela

```
leInt :: IO(Int)
leInt = do
   putStr "Digite um valor inteiro: "
   readLn

main :: IO ()
main = do
   n1 <- leInt
   n2 <- leInt
   putStr "A soma e´: "
   print (n1+n2)
```

```
Main> main
Digite um valor inteiro: 5
Digite um valor inteiro: 4
A soma e´: 9
```

Haskell
IO – Leitura e escrita na tela

```
leInt :: IO(Int)
leInt = do
   putStr "Digite um valor inteiro: "
   readLn
```

```
Main> reverse leInt
ERROR - Type error in application
*** Expression     : reverse leInt
*** Term           : leInt
*** Type           : IO Int
*** Does not match : [a]
```

Haskell
IO – Leitura de arquivos

- Nomes de arquivos e diretórios são objetos do tipo String, e podem especificar o path completo até o arquivo, ou apenas o nome do arquivo relativo ao diretório corrente. O formato do nome do arquivo corresponde ao utilizado no sistema operacional Unix.
- Arquivos podem ser abertos para leitura, escrita ou leitura/escrita. Essa operação associa ao arquivo um *handler* (do tipo *Handle*), que é usado para posteriores referências ao arquivo em operações de leitura e escrita.
- Os dados lidos em um arquivo são carrefados como uma String
  - type File = String
  - writeFile :: File -> String -> IO ()
  - appendFile :: File -> String -> IO ()
  - readFile :: File -> IO String

Haskell
IO – Leitura de arquivos

- Exemplo
  - Ler um arquivo e jogar na tela

```
leArq :: IO ()
leArq = do
   putStr "Digite o nome do arquivo de entrada: "
   ifile <- getLine
   s <- readFile ifile
   putStr s
```

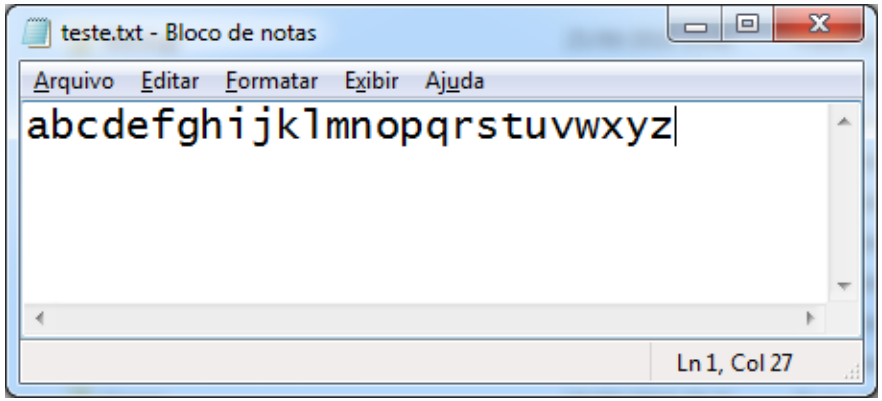
```
Main> leArq
Digite o nome do arquivo de entrada: teste.txt
abcdefghijklmnopqrstuvwxyz
```

Haskell
IO – Escrita de arquivos

- geraArq:: String -> IO()
- geraArq a = do
- putStr "Digite o nome do arquivo que deseja gerar: "
- file <- getLine
- writeFile file a

- Main> geraArq ['a'..'z']
- Digite o nome do arquivo que deseja gerar: teste.txt

- No arquivo teste.txt



Haskell
Interface Gráfica

- WXHASKELL
  - Provê interface gráfica para Windows, Mac, Linux
- http://en.wikibooks.org/wiki/Haskell/GUI
  - Exemplo de tela gerada pelo wxhaskell

Referências

- [1] **http://www.macs.hw.ac.uk/~dubois/ProgramacaoHaskell.pdf**
- [2] http://www.marcosrodrigues6.hpg.ig.com.br/cap1.htm
- [3] http://www.cin.ufpe.br/~alms/pdf/JogosEducativosLinguagensFuncionais.pdf
- [4] http://caioariede.com/2009/aprendendo-erlang-parte-1
- [5] http://www.haskell.org/haskellwiki/Introduction
- [6] http://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_funcional#
- [7] http://www.erlang.se/publications/bjarnelic.pdf
- [8] http://www.haskell.org/tutorial/intro.html
- [9]**http://www.mat.uc.pt/~pedro/lectivos/ProgramacaoFuncional/apontamentosHaskellcap6.pdf**
- [10] Concepts, Techniques, and Models of Computar Programming, Peter Van Ray and Seif Haridi. 2003.
- [11] http://www-usr.inf.ufsm.br/~andrea/elc117/IOHaskell.pdf