Programação Funcional
18ª Aula — Plataforma Haskell

Pedro Vasconcelos
DCC/FCUP

2014

A plataforma Haskell

- Compilador e interpretador (GHC/GHCi)
- Ferramentas de desenvolvimento
- Um conjunto de módulos de base:
  - estruturas de dados;
  - conversões de/para texto (*parsing*, *pretty-printing*);
  - acesso a funcionalidades do sistema operativo (ex: rede, gráficos);
  - teste e *debugging* de programas;
  - ...

`http://www.haskell.org/platform/contents.html`

Algumas módulos de base

Data.List	operações sobre listas
Data.Set	conjuntos finitos
Data.Map	tabelas de associação

Data.List

Funções sobre listas (para além das do Prelúdio):

```
insert :: Ord a => a -> [a] -> [a]
delete :: Eq a => a -> [a] -> [a]
nub :: Eq a => [a] -> [a]
union :: Eq a => [a] -> [a] -> [a]
intersect :: Eq a => [a] -> [a] -> [a]
sort :: Ord a => [a] -> [a]
find :: (a -> Bool) -> [a] -> Maybe a
```

. . . muitas outras omitidas.

Exemplos

```
> import Data.Char
> import Data.List

> insert 3 [1,2,5,7]      -- lista ordenada
[1,2,3,5,7]
> nub "banana"
"ban"
> delete 'a' "banana"
"bnana"
> find isUpper "banana"
Nothing
> find (>'b') "banana"
Just 'n'
```

Exemplos (cont.)

```
> sort [5,1,3,1,2]
[1,1,2,3,5]
> sort "a maria tinha um cordeiro"
"    aaaacdehiiimmnoorrrtu"
> sort ["a","maria","tinha","um","cordeiro"]
["a","cordeiro","maria","tinha","um"]
```

Data.Set

- Conjuntos finitos
- Implementação usando árvores equilibradas
- Pesquisa e inserção eficientes: $O(\log n)$ para n elementos

Data.Set (cont.)

```
data Set a

empty :: Set a
fromList :: Ord a => [a] -> Set a
insert, delete :: Ord a => a -> Set a -> Set a
member :: Ord a => a -> Set a -> Bool
size :: Set a -> Int
union, intersection, difference ::
        Ord a => Set a -> Set a -> Set a
```

...muitas outras funções omitidas.

Exemplos

```
> import Data.Set
> let a = fromList [1,3..20]
> let b = fromList [1,5..20]
> intersection a b
fromList [1,5,9,13,17]
> difference a b
fromList [3,7,11,15,19]
> member 15 (difference a b)
True
> size (difference a b)
5
```

Data.Map

- Tabelas de associações entre *chaves* e *valores* (dicionários)
- Implementado usando árvores equilibradas
- Pesquisa e inserção eficientes: $O(\log n)$ para n entradas

Data.Map (cont.)

```
data Map k a                -- chaves k, valores a

empty :: Map k a
fromList :: Ord k => [(k,a)] -> Map k a
insert :: Ord k => k -> a -> Map k a -> Map k a
delete :: Ord k => k -> Map k a -> Map k a
size :: Map k a -> Int
lookup :: Ord k => k -> Map k a -> Maybe a
findWithDefault :: Ord k => a -> k -> Map k a -> a
```

...muitas outras operações omitidas.

Exemplos

```
> import Data.Map
> let t = fromList [("CC",60), ("MIERSI",80)
                    ("AST",5), ("MAT",10)]
> lookup "MIERSI" t
    Ambiguous occurrence ‘lookup’...
    ...
> Data.Map.lookup "MIERSI" t
Just 80
> Data.Map.lookup "BIO" t
Nothing
> Data.Map.findWithDefault 0 "BIO" t
0
```

Exemplos (cont.)

Usamos *nomes qualificados* para evitar ambiguidade.

```
> import qualified Data.Map as Map
> let t = Map.fromList [("CC",60), ("MIERSI",80)
                        ("AST",5), ("MAT",10)]
> Map.lookup "MIERSI" t
Just 80
> Map.lookup "BIO" t
Nothing
> Map.findWithDefault 0 "BIO" t
0
```

Inserções e remoções

Set e *Map* são estruturas de dados funcionais:

- as inserções e remoções criam *sempre* novas estruturas;
- não há modificações.

```
> let a = Set.fromList [1,3,5]
> let a' = Set.insert 4 a
> a'
fromList [1,3,4,5]
> a
fromList [1,3,5]
```

Inserções e remoções (cont.)

```
> let t = Map.fromList [("CC",40), ("MIERSI",80)]
> let t'= Map.insert "CC" 50 t
> t'
fromList [("CC",50), ("MIERSI",80)]
> t
fromList [("CC",40), ("MIERSI",80)]
```

Contar palavras

Exemplo de uso: contar palavras distintas de um texto.

Contar palavras (cont.)

Vamos acumular palavras num conjunto:

```
Set String
```

Exemplo:

```
"a maria cordeiro tinha um cordeiro maria"
```

produz o conjunto

```
fromList ["a","maria","tinha","um","cordeiro"]
```

Contar palavras (cont.)

```
import Data.Char
import qualified Data.Set as Set

main :: IO ()
main = do txt <- getContents
          let ws = map clean (words txt)          -- lista
          let s = Set.fromList ws     -- construir conjunto
          print (Set.size s)          -- número de palavras
  where
    clean = map toLower . filter isLetter
```

Contar ocorrências

Variante: contar número de ocorrências de cada palavra num texto.

Contar ocorrências (cont.)

Acumulamos as contagens num dicionário:

```
Map String Int
```

Exemplo:

```
"a maria cordeiro tinha um cordeiro maria"
```

produz o dicionário

```
fromList [("a",1),("maria",2),("tinha",1),
          ("um",1),("cordeiro",2)]
```

Contar ocorrências (cont.)

```
import Data.Char
import qualified Data.Map as Map

main :: IO ()
main = do txt <- getContents
          let ws = map clean (words txt)
          -- construir e imprimir contagens
          print (foldl add Map.empty ws)
  where
    clean = map toLower . filter isLetter
    add d w = let c = Map.findWithDefault 0 w d
              in Map.insert w (c+1) d
```

Mais alguns exemplos

Data.Time funções de relógio e tempo

Network.HTTP protocolo de rede HTTP

Data.Time

```
import Data.Time

main = do t <- getCurrentTime   -- hora e data atual
          putStrLn ("Data e hora atual: " ++ show t)
```

Network.HTTP

```
import Network.HTTP

-- buscar uma página HTML como texto
main = do r <- simpleHTTP (getRequest url)
          html <- getResponseBody r
          putStrLn html
   where url = "http://www.haskell.org"
```

HackageDB

`http://hackage.haskell.org/`

- Repositório de bibliotecas e aplicações *open-source*
- Interface *web* para pesquisa de pacotes por categorias
- Instalação automática usando a ferramenta `cabal-install`

Hoogle

`http://www.haskell.org/hoogle/`

- Motor de pesquisa sobre bibliotecas de Haskell
- Pesquisa por nomes e/ou tipos