Exercícios para tutoria. Semana 7.

Programação Funcional

Prof. Rodrigo Ribeiro

Modelando um SGDB

Setup

```
module Semana7 where
```

Introdução

O objetivo desta lista de exercícios é a implementação de funções para simular um interpretador de consultas em um sistema gerenciador de banco de dados.

Para isso, representaremos tabelas usando o tipo de dados `Table`

```
data Table
  = Table {
      tableName   :: String,         -- nome da tabela
      tableFields :: [Field],        -- campos
      values :: [[String]]           -- valores associados a cada campo
    } deriving (Eq, Ord)
```

Um valor do tipo `Table` é formado por três componentes, em que o primeiro componente é o nome da tabela, o segundo é a lista dos campos da tabela e o terceiro é a lista das linhas da tabela, sendo cada linha uma lista em que cada elemento em uma posição i da lista corresponde à informação da coluna i. Todas as informações contidas na tabela são do tipo `String`.

Cada campo de uma tabela é representado por um valor do tipo `Field`, que é formado pelo nome do campo e seu respectivo tipo. Tipos de dados são representados pelo tipo `Type` e especificam os tipos de dados mais comuns presentes em bancos de dados relacionais. Os tipos `Field` e `Type` são apresentados a seguir:

```
data Field
  = Field  {
      fieldName :: String,      -- nome do campo
```

```
      fieldType :: Type        -- tipo de dados associado com este campo
    } deriving (Eq, Ord, Show)

data Type = TyInt                -- números inteiros
          | TyDouble             -- números de ponto flutuante
          | TyBool               -- valores lógicos
          | TyVarChar (Maybe Int) -- Strings.Pode especificar o comp. máx. n
                                 -- por Just n. Nothing => 255
          | TyDate               -- Datas
          | TyCurrency           -- valores monetários
          deriving (Eq, Ord)
```

Para exemplificar a representação a ser utilizada neste trabalho, considere o trecho de código a seguir, onde é mostrada uma tabela de exemplo:

```
client :: Table
client = Table "Cliente" fieldsCliente dataCliente

fieldsCliente :: [Field]
fieldsCliente
   = [ Field "id" TyInt
     , Field "nome" (TyVarChar (Just 15))
     , Field "cpf" (TyVarChar (Just 11))]

dataCliente :: [[String]]
dataCliente
   = [ ["1", "Jose da Silva", "23333245678"]
     , ["2", "Joaquim Souza", "09863737213"]
     , ["3", "Roberto Martins", "45627819081"]
     ]
```

Observe que os valores `fieldsCliente` e `dataCliente` são utilizados apenas para criar a lista de campos e dos valores associados com a tabela criada.

Outro exemplo de tabelas é apresentado a seguir.

```
address :: Table
address
   = Table "Endereco"
           fieldsEndereco
           dataEndereco

fieldsEndereco :: [Field]
fieldsEndereco = [ Field "cpf" (TyVarChar (Just 11))
                 , Field "rua" (TyVarChar (Just 15))
                 , Field "bairro" (TyVarChar (Just 15))
                 , Field "cidade" (TyVarChar (Just 15))
                 , Field "estado" (TyVarChar (Just 2))
                 ]
```

```haskell
dataEndereco :: [[String]]
dataEndereco = [ ["23333245678", "rua 2", "alfa", "jurema do sul", "SC"]
               , ["09863737213", "rua 89", "beta", "jurema do norte", "SC"]
               , ["45627819081", "rua 10", "gama", "jurema do leste", "SC"]
               ]
```

Com base no apresentado, resolva as questões propostas a seguir.

Exercício 1. Um esquema de uma tabela de um banco de dados relacional é um produto dos tipos de cada uma das tabelas de um banco de dados. Considere o seguinte tipo de dados que representa um esquema de uma tabela de um banco de dados relacional:

```haskell
infixr 5 :*:

data Schema = Type :*: Schema
            | Nil
            deriving (Eq, Ord)
```

Onde o construtor de dados `:*:` representa o produto do tipo de um campo de uma tabela e o esquema que representa o restante da tabela. Como exemplo, o esquema que representa a tabela cliente mostrada anteriormente é:

```haskell
esquemaCliente :: Schema
esquemaCliente
   = TyInt               :*:
     TyVarChar (Just 15) :*:
     TyVarChar (Just 11) :*: Nil
```

O construtor de dados `Nil` é utilizado apenas como um marcador de fim do produto do esquema de uma tabela.

O tipo (esquema) de uma tabela é representado pelo produto cartesiano de cada uma das colunas que compõe a tabela. Valores do tipo `Type` podem ser impressos (convertidos em `String`'s) de acordo com a seguinte tabela de equivalência:

Valor do tipo Type	String correspondente
TyInt	Int
TyDouble	Double
TyBool	Bool
TyVarChar (Just n)	VarChar[n]
TyVarChar Nothing	VarChar
TyDate	Date
TyCurrency	Currency

Para a impressão de esquemas, basta considerar que o construtor de dados `:*:` é o equivalente ao produto cartesiano `X` entre tipos de cada coluna de uma tabela.

Como exemplo, o esquema da tabela cliente, apresentado anteriormente, pode ser impresso como:

```
Int X VarChar[15] X VarChar[11]
```

Observe que o construtor de dados `Nil` é substituído pela string vazia.

a) Desenvolva uma instância de `Show` para o tipo `Type` que reflita a conversão de valores deste tipo em strings de acordo com a tabela apresentada anteriormente.

```
instance Show Type where
  show = undefined
```

b) Desenvolva uma instância para `Show` para o tipo `Schema` que permita imprimir esquemas de tabelas de maneira similar ao exemplo apresentado anteriormente para a tabela cliente.

```
instance Show Schema where
 show = undefined
```

c) Implemente a função

```
schema :: Table -> Schema
schema = undefined
```

que dada uma tabela, representada pelo tipo de dados `Table`, retorne o seu esquema correspondente.

d) Defina uma instância de `Show` para o tipo `Table`, de maneira que a tabela cliente apresentada anteriormente seja impressa da seguinte maneira:

```
Cliente:
--------------------------------
id nome            cpf
--------------------------------
1  Jose da Silva   23333245678
2  Joaquim Souza   09863737213
3  Roberto Martins 45627819081
```

```
instance Show Table where
  show = undefined
```

Exercício 2. Defina a função

```
count :: Table -> Int
count = undefined
```

que retorna o número de registros presentes em uma tabela.

Exercício 3. Defina a função

```
project :: Table -> [String] -> Either String Table
project = undefined
```

que receba como argumento uma tabela e uma lista de nomes de campos desta tabela e retorne como resultado uma nova tabela que contenha somente as informações presentes nas colunas especificadas pelo segundo argumento. Como exemplo, se executarmos

```
project client ["id","nome"]
```

o resultado ser'a:

```
Right Cliente-id-nome:
--------------------
id nome
--------------------
1  Jose da Silva
2  Joaquim Souza
3  Roberto Martins
```

Observe que o nome da tabela resultante é formado pela concatenação do nome da tabela com o nome dos campos que foram fornecidos como argumento para a função `project`. O resultado desta função é expresso pelo tipo `Either String Table`}. O construtor de tipos `Either` é normalmente utilizado para representar funções que podem resultar em erros. No caso da função `project`, ela deve retornar uma mensagem de erro, sempre que seja passado como parâmetro um nome de campo que não esteja presente na tabela em que deseja-se realizar a projeção. Isto é, se executarmos

```
project client ["id", "teste"]
```

deverá ser retornado o seguinte resultado:

```
Left "The fields:\nteste\nisnt defined in table: Cliente"
```

em que `Left` é o construtor de dados do tipo `Either` que é utilizado para representar erros.

Exercício 4. Implemente a função

```
restrict :: Table -> Condition -> Either String Table
restrict t c = undefined
```

Que seleciona todos os registros de uma tabela que atendem a uma determinada condição. Condições são representadas pelo tipo `Condition`, apresentado a seguir:

```
data Condition
  = Condition {
      field     :: String
    , condition :: String -> Bool
    }
```

em que `field` representa o nome do campo que será utilizado no processo de seleção de registros e `condition` é uma função que verifica se um valor deste

campo atende ou não a condição especificada. Como exemplo, considere o seguinte valor do tipo `Condition`, que representa a condição: Selecione todos os nomes que começam com a letra R:

```
cond :: Condition
cond = Condition "nome" (\s -> head s == 'R')
```

Ao executarmos

```
restrict client cond
```

deve ser impresso o seguinte valor:

```
Right Cliente:
--------------------------------
id nome            cpf
--------------------------------
3  Roberto Martins 45627819081
```

Exercício 5. O objetivo deste exercício é definir uma função para combinar duas tabelas.

Duas tabelas apenas podem ser combinadas se elas possuem um campo em comum, isto é, uma coluna com o mesmo nome e tipo.

A tabela resultante da combinação de duas tabelas deve possuir todas as colunas da primeira tabela, seguidas pelas colunas da segunda tabela que não ocorrem na primeira tabela. Portanto, o número de colunas na tabela resultante é menor do que a soma do número de colunas de cada uma das tabelas combinadas. Cada registro na nova tabela é formado pela combinação dos registros correspondentes das duas tabelas combinadas, ou seja, por registros que têm o mesmo valor na coluna comum às duas tabelas. O nome da nova tabela é formado pela concatenação dos nomes das duas tabelas que são combinadas.

Como exemplo, considere a seguinte tabela que representa endereços de clientes:

```
Endereco:
-----------------------------------------------------
cpf         rua    bairro cidade          estado
-----------------------------------------------------
23333245678 rua 2  alfa   jurema do sul   SC
09863737213 rua 89 beta   jurema do norte SC
45627819081 rua 10 gama   jurema do leste SC
```

Observe que ela possui um campo em comum com a tabela cliente (o campo `cpf`). O resultado de exercutarmos

```
join client address
```

em que `address` é o valor que representa a tabela de endereços acima, deve ser:

```
Right Cliente-Endereco:
-------------------------------------------------------------------------------
```

```
id nome            cpf         rua    bairro cidade          estado
-------------------------------------------------------------------------
1  Jose da Silva   23333245678 rua 2  alfa   jurema do sul   SC
2  Joaquim Souza   09863737213 rua 89 beta   jurema do norte SC
3  Roberto Martins 45627819081 rua 10 gama   jurema do leste SC
```

É importante ressaltar que a ordem das parâmetros fornecidos para a operaçãao de junção altera o resultado. O resultado de executarmos

```
join address client
```

é:

```
Right Endereco-Cliente:
-------------------------------------------------------------------------
cpf         rua    bairro cidade          estado id nome
-------------------------------------------------------------------------
23333245678 rua 2  alfa   jurema do sul   SC     1  Jose da Silva
09863737213 rua 89 beta   jurema do norte SC     2  Joaquim Souza
45627819081 rua 10 gama   jurema do leste SC     3  Roberto Martins
```

A operação de junção pode ser decomposta em dois passos:

- Calcular o produto cartesiano das duas tabelas. Para isso, é criada uma nova tabela que possuirá todos os campos de ambas as tabelas envolvidas no produto. Os registros são obtidos realizando todas as combinações de ambas as tabelas. Isto é feito pegando cada registro da primeira tabela e concatenando-o com cada registro correspondente na segunda tabela.
- Selecionar todos os registros onde o campo em comum de duas tabelas possui o mesmo valor.

Considerando o apresentado, faça:

a) Implemente a função

```
cartProd :: Table -> Table -> Either String Table
cartProd = undefined
```

que calcula o produto cartesiano de duas tabelas conforme descrito acima. Esta função deve retornar uma mensagem de erro, caso as duas tabelas fornecidas como parâmetro não possuam um campo em comum.

b) Implemente a função

```
restrictProd :: Table -> Either String Table
restrictProd = undefined
```

que seleciona os registros adequados do produto cartesiano, conforme descrito acima.

c) De posse das funções anteriores, implemente a função

```haskell
join :: Table -> Table -> Either String Table
join t1 t2 = undefined
```

que realiza a junção de duas tabelas como explicado.