Ficha5-ATS

Luís Braga - https://github.com/BragaMann

December 2019

Contents

1 Ficha5

```haskell
module Ficha5 where
--stack ghci --package QuickCheck -- .\ficha5.hs
import Test.QuickCheck

import Control.Monad
import Control.Monad.State.Strict

genNota :: Gen Word
genNota = choose (0,20)

genNota' :: Gen Word
genNota' = elements [0..20]

--genMarca :: Gen String
--genMarca = frequency [(120,return "Renault"),(85,return "Mercedes"),(12,return "Porsche")
    ,(4,return "Ferrari")]

-- 1.2
{-
Exercicio 1. Escreva um gerador para automoveis cujos atributos constam dos
seguintes tipos de dados:
-}

data Carro = Carro Tipo Marca Matricula NIF CPKm Autonomia
             deriving Show

data Tipo = Combustao
          | Eletrico
          | Hibrido
          deriving Show

type Marca = String
type Matricula = String
type NIF = String -- NIF Proprietario
type CPKm = Float -- Consumo por Km
type Autonomia = Int

{-
De acordo com as estatisticas do ACP, atualmente 70% dos automoveis usam
motores a combustao, 25% sao hibridos e apenas 5% sao eletricos. O consumo
por quilometro (CPKm)   e  dado em euros e   e  um valor entre 0.1 e 2 euros.
As marcas automoveis existentes em Portugal podem ser obtidas no site do
standvirtual. A matricula tem o seguinte formato  AA -11-22   .
A funcao que gera aleatoriamente um carro de acordo com estes requisitos
recebe uma lista de NIFs v  alidos  e tem tipo:

Para construir este gerador, implemente:

1. um gerador genTipo que gera um tipo de carro, de acordo com as es-
tat sticas anteriormente fornecidas.

2. um gerador genCPKm que gera um valor de consumo por Km aleatorio, de
acordo com os valores anteriormente fornecidos.
```

```
2

3. um gerador genAutonomia que gera um valor de autonomia aleatorio.
Para tal, faca uma pesquisa rapida para encontrar valores razoaveis para
usar neste gerador.
4. um gerador genMarca que gera uma marca alea  oria . Para tal, use as
marcas existentes em Portugal.
5. um gerador genMatricula que gera uma matr  cula  aleatoria, de acordo
com o padrao anteriormente fornecido.
6. um gerador genCarro que, dado uma lista de NIFs validos, gera um carro,

utilizando um desses NIFs e todos os geradores anteriormente implemen-
tados.
-}



genCarro :: [NIF] -> Gen Carro
genCarro nifs = do tipo <- genTipo
                   marca <- genMarca
                   matricula <- genMatricula
                   nif <- elements nifs
                   cpkm <- genCPKm
                   autonomia <- genAutonomia
                   return (Carro tipo marca matricula nif cpkm autonomia)
-- sample (genCarro ["112736222","287317237","316595818","479532116","561859338","378838664"])
genTipo :: Gen Tipo
genTipo = frequency [(75, return Combustao), (25, return Hibrido), (5, return Eletrico)]

genCPKm :: Gen CPKm
genCPKm = elements [0.1 .. 2]

genAutonomia :: Gen Autonomia
genAutonomia = elements [100 .. 700]

genMarca :: Gen Marca
genMarca = frequency [(21, return "Aston Martin"),(2377, return "Audi"),(4998, return "BMW")
    ,(11, return "Cadilac"),(1605, return "Citroen"),(48, return "Ferrari"),(6, return "Rolls
    Royce"),(38, return "Tesla")]

genMatricula :: Gen Matricula
genMatricula = do x <- f ['A'..'Z']
                  y <- g
                  z <- g
                  return (x ++ "-" ++ y ++ "-" ++ z)
             where f a = vectorOf 2 $ elements a
                   g = f ['1'..'9']

genNif :: Gen NIF
genNif = vectorOf 9 $ elements ['1'..'9']

{-
Exerc  cio  2. Defina um tipo de dados para representar estudantes e as suas
notas. Um estudante e definido pelo seu nome, numero e um tipo de registo
(Normal, Militar, Trabalhador). Defina um gerador para estudantes, sabendo
que 80% dos estudantes s  ao  normais, 15% sao trabalhadores estudantes e 5%
sao militares.
-}

data Estudante = Estudante Nome Numero TipoR
                 deriving Show

type Nome = String
type Numero = Integer

data TipoR = Normal
           | Militar
           | Trabalhador
           deriving Show

genTipoR :: Gen TipoR
genTipoR = frequency [(80, return Normal), (15, return Trabalhador), (5, return Militar)]

genNome :: Gen Nome
genNome = elements nomes
```

```
127
128 nomes :: [String]
129 nomes = ["Manuel", "Jose", "Joao"]
130
131 genNumero :: Gen Numero
132 genNumero = elements [80000..85000]
133
134 genEstudante :: Gen Estudante
135 genEstudante = do nome <- genNome
136                   numero <- genNumero
137                   tipo <- genTipoR
138                   return (Estudante nome numero tipo)
139
140 {-
141 Exerc   cio   3. Considere o seguinte tipo de dados para definir expressoes ar-
142 itmeticas:
143
144 data Expr = Add Expr Expr
145 | Mul Expr Expr
146 | Const Float
147 1. Implemente um gerador para expressoes aritmeticas, em que 80% das
148 operacoes sao somas e 20% das operacoes sao multiplicacoes. A prob-
149 abilidade de se gerar uma expressao aritmetica ou um valor constante na
150 chamada recursiva devera ser igual.
151
152 2. Re-implemente este gerador para expressoes aritmeticas, utilizando o com-
153 binador sized para definir o tamanho da expressao aritmetica. Usando
154 este combinador, o gerador ira receber um valor como argumento que
155 indicara o tamanho da expressao a gerar.
156 -}
157
158 data Expr = Add Expr Expr
159           | Mul Expr Expr
160           | Const Float
161           deriving Show
162
163 --1
164
165 genExpr' :: Gen Expr
166 genExpr'   = frequency [(100,do f <- elements [0.1..1000]
167                                 return (Const f))
168                         ,(80,do l <- genExpr'
169                                 r <- genExpr'
170                                 return (Add l r))
171                         ,(20,do l <- genExpr'
172                                 r <- genExpr'
173                                 return (Mul l r))]
174
175 -- 2
176
177 instance Arbitrary Expr where
178             arbitrary = sized genExpr
179
180 genExpr s = frequency [(1,do f <- arbitrary
181                              return (Const f))
182                        ,(s,do l <- genExpr s'
183                               r <- genExpr s'
184                               return (Add l r))
185                        ,(s,do l <- genExpr s'
186                               r <- genExpr s'
187                               return (Mul l r))]
188       where s' = div s 2
189
190 {-
191 Exerc   cio   4. Considere o seguinte tipo de dados para representar Binary Search
192 Trees (BST), em que se guardam numeros inteiros nos nodos da arvore:
193 data BST = Empty
194 | Node BST Int BST
195 deriving Show
196 1. Define um gerador para BSTs.
197 2. Defina funcoes sobre BSTs, e experimente as funcoes definidas em arvores
198 geradas:
199
200 -}
201
202 data BST = Empty
```

```haskell
         | Node BST Int BST

instance Show BST where
       show = pp_BST

pp_BST = unlines . layoutTree

indent :: [String] -> [String]
indent = map ("  "++)

layoutTree Empty = []
layoutTree (Node left here right)
        = indent (layoutTree right) ++ [show here] ++ indent (layoutTree left)

insert :: BST -> Int -> BST
insert Empty x = Node Empty x Empty
insert (Node t1 v t2) x
       | v == x = Node t1 v t2
       | v  < x = Node t1 v (insert t2 x)
       | v  > x = Node (insert t1 x) v t2

insertAll :: BST -> [Int] -> BST
insertAll bst [] = bst
insertAll bst (x:xs) = insertAll (insert bst x) xs

genListaInt = (arbitrary :: Gen [Int])

genBST = do l <- genListaInt
            return (insertAll Empty l)

-- versao 2 que gera por limites
instance Arbitrary BST where
       arbitrary = sized $ aux 0 1000
              where aux min max 0 = genVazio
                    aux min max n = frequency [(1,genVazio)
                                              ,(4,genNodo min max n)]
                    genVazio = return Empty
                    genNodo min max n = do v <- choose(min, max)
                                           l <- aux min v (n `div` 2)
                                           r <- aux v max (n `div` 2)
                                           return (Node l v r)

genBST' = (arbitrary :: Gen BST)
-- generate genBST'
{-
Defina funcoes sobre BSTs, e experimente as funcoes definidas em arvores
geradas:
-}

--a)
size :: BST -> Int
size Empty = 0
size (Node l x r) = 1 + size l + size r

--b)
height :: BST -> Int
height Empty = 0
height (Node l _ r) = 1 + (Prelude.max (height l) (height r))

--c)
max :: BST -> Int
max (Node _ x Empty) = x
max (Node _ x r) = Ficha5.max r

-- d)
inorder :: BST -> [Int]
inorder Empty = []
inorder (Node l v r) = inorder l ++ [v] ++ inorder r

--e)
ordered :: BST -> Bool
ordered Empty = True
ordered bst = isOrdered (Ficha5.min bst) (Ficha5.max bst) bst

min :: BST -> Int
min (Node Empty x _) = x
```

```
min (Node l x _) = Ficha5.min l

isOrdered :: Int -> Int -> BST -> Bool
isOrdered _ _ Empty = True
isOrdered min max (Node l x r) = x >= min && x <= max && (isOrdered min x l) && (isOrdered x
    max r)

--f)
balanced :: BST -> Bool
balanced Empty = True
balanced (Node l _ r) = abs (height l - height r) <= 1 && balanced l && balanced r

--g)
foldT :: (Int -> a -> a) -> a -> BST -> a
foldT _ a Empty = a
foldT f a (Node l x r) = f x (foldT f acc r)
                      where acc = foldT f a l

-- Binary Tree Gen

data BT = Vazio
        | Nodo BT Int BT
        deriving Show

instance Arbitrary BT where
      arbitrary = sized $ aux
        where aux 0 = genVazio
              aux n = frequency [(1,genVazio)
                                ,(4,genNodo n)]
              genVazio = return Vazio
              genNodo n = do v <- arbitrary
                             l <- aux (n `div` 2)
                             r <- aux (n `div` 2)
                             return (Nodo l v r)

genBT = (arbitrary :: Gen BT)

{-
Exerc   cio   5. Relembre a ficha de An   alise   Est   atica   resolvida anteriormente.
Nesta ficha, usa-se uma linguagem definida pelos seguintes tipos de dados:
data P = R Its
type Its = [It]
data It = Block Its
| Decl String
| Use String

1. Implemente um gerador para blocos de codigo nesta linguagem, sem qual-
quer nocao de validade de codigo. Use o analisador sintatico anteriormente

desenvolvido para avaliar se os resultados deste gerador sao os esperados.
2. Implemente um gerador para blocos de codigo nesta linguagem. O codigo
produzido devera ser valido, isto e, apenas se devera poder usar variaveis
declaradas no scope atual ou no scope anterior, e nao se podera declarar

uma variavel duas vezes no mesmo scope. Use o analisador sintatico ante-
riormente desenvolvido para avaliar se os resultados deste gerador sao os
esperados.
-}

{-
1.3 Geracao com Estado
Geracao mais complexas podem envolver a criacao de um estado para seleccao
correcta dos valores seguintes. Usando QuickCheck, e possivel implementar isso
de duas formas:
    implementar todas os geradores recebendo e alterando um estado;
    por o gerador numa pilha de transformers.
A primeira opcao nao e ideal pois nao nos e possivel reutilizar as funcoes
para geracao de listas de elementos (entre outros geradores). Neste caso seria
necessario reimplementar essas funcoes.
A segunda opcao e mais complexa, mas fornece muito mais possibilidades.
-}
--O tipo do gerador fica:
type Gerador st a = StateT st Gen a

--sendo possivel executa-lo com:
executar :: st -> Gerador st a -> Gen a
```

```haskell
executar st g = evalStateT g st

--Dentro de um gerador sera necessario agora fazer lift das funcoes de gerador:
genNotaSt :: Gerador () Word
genNotaSt = lift $ choose (0,20)
--sample (executar ()  genNotaSt)

--mas e possivel aceder directamente ao estado para ler o seu valor:
genNotaSt' :: Gerador (Word, Word) Word
genNotaSt' = get >>= \ (a,b) -> lift $ choose (a,b)
-- sample (executar (0,20)  genNotaSt')

--ou para guardar novo valor:
genNotaSt'' :: Gerador [Word] Word
genNotaSt'' = do
              l <- get
              n <- lift $ choose (0,20)
              put (n:l)
              return n
--sample (executar [1..20]  genNotaSt'')

{-
Exercicio 1. Implemente um gerador de numeros unicos, i.e., a cada geracao    stack ghci --
    package QuickCheck -- .\ficha5.hs
de um numero, esse numero nao podera ter sido gerado antes.
-}

-- haskell struct
data GenState
       = GenState
       {
              stNotas :: [Int]
       }deriving Show

defaultGenState :: GenState
defaultGenState
  = GenState
  {
       stNotas = []
  }


type GenNums = [Int]

defaultNums :: GenNums
defaultNums = []

genNum :: Gerador [Int] ()
genNum = do
         s <- get
         l <- lift $ choose(0,1000)
         if (elem l s) then genNum
         else put (l:s)

genNums :: Int -> Gerador [Int] [Int]
genNums 0 = do
            s <- get
            return s
genNums n = do
            genNum
            r <- genNums (n-1)
            return r
--sample (executar defaultNums (genNums 3))
```