Paradigmas de Programação

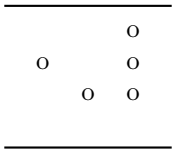
Fabrício Olivetti de França

07 de Agosto de 2018

Game of Life

O Jogo da Vida ou *Game of Life* ou *Conway's Game of Life* modela um sistema evolutivo simples baseado em células. Geralmente é jogado em um tabuleiro bidimensional.

Cada posição do tabuleiro ou está vazia ou contém uma célula viva:

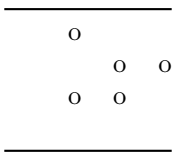


Cada posição tem oito vizinhos imediatos: cima, baixo, direita, esquerda e as diagonais. As posições da extremidade também possuem oito vizinhos, ou seja, o vizinho acima da posição mais em cima é a posição mais embaixo.

Dada uma configuração inicial do tabuleiro, a próxima geração é dado pela aplicação simultânea das seguintes regras:

- Uma célula viva sobrevive se tem precisamente dois ou três vizinhos com células vivas.
- Uma posição vazia cria uma nova célula viva se possui extamente três vizinhos vivos.

Aplicando essa regra no tabuleiro acima, geramos:



Repetindo a aplicação das regras infinitamente, podemos gerar uma sequência

infinita que pode levar a estabilidade, ciclo ou comportamento aleatório.

Crie um projeto com o *stack* chamado *life*.

Funções de tela

O nosso tabuleiro será representado por uma lista de posições, só manteremos nessa lista as posições contendo células vivas:

```
import Control.Monad
import Control.Applicative
import Data.List

type Pos = (Int, Int)

largura :: Int
largura = 10

altura :: Int
altura = 10

type Board = [Pos]

glider :: Board
glider = [(4,2),(2,3),(4,3),(3,4),(4,4)]
```

Duas funções úteis para manipulação de tela são dadas abaixo:

```
-- | limpa a tela
cls :: IO ()
cls = putStr "\ESC[2J"

-- | coloca o cursor na posição x,y
goto :: Pos -> IO ()
goto (x, y) = putStr ("\ESC[" ++ show y ++ ";" ++ show x ++ "H")
```

1. Crie uma função chamada `writeAt` que recebe uma posição e uma String e escreve a String nessa posição, use o *do-notation*:

```
writeAt :: Pos -> String -> IO ()
writeAt p xs = ???
```

Altere a função `main` para:

```
main :: IO ()
main = do cls
          writeAt (2,2) "hello worldn"
          return ()
```

E teste o resultado.

2. Crie uma função chamada `showCells` que executa a sequência `writeAt p "O"` para cada `p` de um tabuleiro `b`, use a função `sequenceA`:

```
showCells :: Board -> IO [()]
showCells b = ???
```

Altere a função `main` para:

```
main :: IO ()
main = do cls
          showCells glider
          return ()
```

E teste.

Funções do Jogo da Vida

1. Implemente as funções `isAlive` que verifica se uma posição `p` contém célula viva no tabuleiro `b` e `isEmpty`, que é a negação de `isAlive`. Utilize a função `elem` para fazer a verificação:

```
isAlive :: Board -> Pos -> Bool
isAlive b p = ???

isEmpty :: Board -> Pos -> Bool
isEmpty b p = ???
```

2. Defina agora a função `neighbs` para gerar a lista de vizinhos para a posição `(x, y)`, a função `wrap` garante que as coordenadas estarão dentro do tabuleiro:

```
neighbs :: Pos -> [Pos]
neighbs (x,y) = map wrap ???

wrap :: Pos -> Pos
wrap (x, y) = ( ((x-1) `mod` largura)  + 1,
                ((y-1) `mod` altura) + 1)
```

3. Agora podemos verificar o número de vizinhos de vivo de certa célula com a composição de `isAlive` e `neighbs`. A ideia é pegar a lista de vizinhos, filtrar aqueles que estão vivos e verificar o tamanho da lista resultante:

```
liveNeighbs :: Board -> Pos -> Int
liveNeighbs b p = ???
```

4. Defina a função `survivors` que verifica quais células sobrevivem pela primeira regra:

```haskell
regra1 :: Board -> Pos -> Bool
regra1 b p = n==2 || n==3
  where n = liveNeighbs b p

survivors :: Board -> [Pos]
survivors b = ???
```

5. Finalmente vamos verificar quantas posições darão vida a novas células. Levando em conta que apenas posições vizinhas a células vivas podem gerar novas células, primeiro crie uma lista dos vizinhos das células (DICA: ao mapear a função `neighbs` você gera uma lista de listas, use `concat` para transformar em uma lista única e `nub` para remover duplicatas), filtre as não vazias e verifique se existem exatamente 3 vizinhos vivos:

```haskell
births :: Board -> [Pos]
births b = [p | p <- ???,
                ???,
                ???]
```

Agora podemos produzir a nova geração concatenando os sobreviventes com os que nasceram:

```haskell
nextGen :: Board -> Board
nextGen b = survivors b ++ births b

life :: Board -> IO ()
life b = do cls
            showCells b
            wait 100000      -- ajuste para seu PC
            life (nextGen b)

wait :: Int -> IO [()]
wait n = sequenceA [return () | _ <- [1..n]]

main :: IO ()
main = life glider
```

6. Teste outros tamanhos de tabuleiros e padrões extraídos de https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#Examples_of_patterns.