Programação Funcional
10ª Aula — O Jogo da Vida

Pedro Vasconcelos
DCC/FCUP

2014

O Jogo da Vida

- Um *autómato celular* inventado pelo matemático John H. Conway.
- O jogo desenrola-se numa grelha bi-dimensional.
- Cada posição está vazia ou tem uma célula.
- A colónia de células evolui por gerações.
- Determinamos uma nova geração pelas seguintes regras:
  1. morrem as células com *menos do que 2 ou mais do que 3 vizinhos*;
  2. sobrevivem células com *2 ou 3 vizinhos*;
  3. nasce uma nova célula em cada posição vazia com *exactamente 3 vizinhos*.

`http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

Objetivo

Um programa que:

- simula a passagem de *n* gerações;
- mostra a sucessão de gerações no terminal.

Baseado na solução do livro *Programming in Haskell* de Graham Hutton (capítulo 9).

Representação do jogo

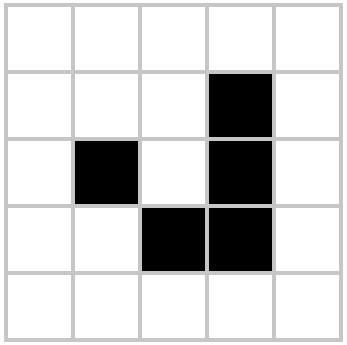
Vamos representar a colónia de células por uma lista de coordenadas:

```
type Pos = (Int,Int)                    -- coluna, linha

type Cells = [Pos]                      -- coordenadas das células
```

Exemplo: um *glider*.

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



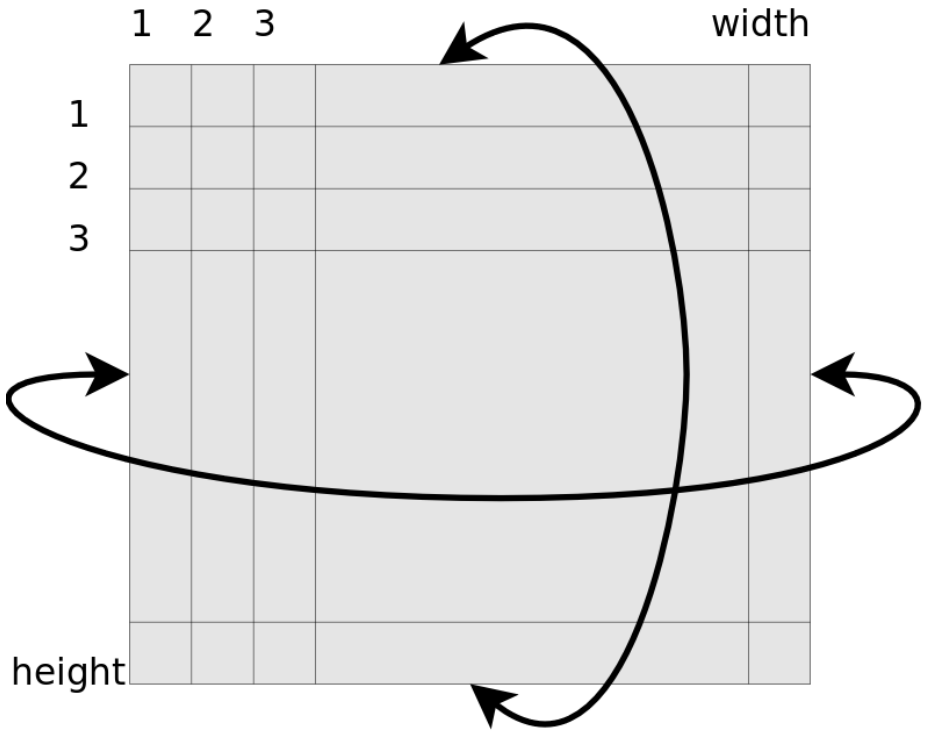
Representação do jogo (cont.)

Para facilitar a visualização:

- largura e altura limitadas;

```
width, height :: Int
width = 80
height = 24
```

- lados esquerdo/direito e de topo/baixo são ligados.



Algumas funções auxiliares

```
-- testar se uma posição está viva ou morta
-- usa 'elem' (do prelúdio-padrão)

isAlive, isEmpty :: Cells -> Pos -> Bool
isAlive ps p = elem p ps
isEmpty ps p = not (isAlive ps p)
```

Algumas funções auxiliares (cont.)

```
-- obter as 8 posições vizinhas
neighbs :: Pos -> [Pos]
neighbs (x,y)  =  map wrap [(x-1,y-1), (x,y-1),
                            (x+1,y-1), (x-1,y),
                            (x+1,y)  , (x-1,y+1),
                            (x,y+1)  , (x+1,y+1)]

-- garantir que uma posição está dentro do tabuleiro
wrap :: Pos -> Pos
wrap (x,y) =  ((x-1) `mod` width + 1,
               (y-1) `mod` height + 1)
```

Algumas funções auxiliares (cont.)

```
-- contar células vivas entre as vizinhas
liveneighbs :: Cells -> Pos -> Int
liveneighbs ps = length . filter (isAlive ps) . neighbs
```

Transição entre gerações

A nova geração depende apenas da geração atual. Assim, vamos definir uma função de transição entre gerações.

As novas celulas são as sobreviventes mais os nascimentos:

```
nextgen :: Cells -> Cells
nextgen ps = survivors ps ++ births ps
```

Falta definir duas funções auxiliares:

```
survivors, births :: Cells -> Cells
```

Transição entre gerações (cont.)

```
-- sobreviventes duma geração
survivors :: Cells -> Cells
survivors ps
   = [p | p<-ps, elem (liveneighbs ps p) [2,3]]

-- nascimentos duma geração
-- 'nub' remove repetidos duma lista
births :: Cells -> Cells
births ps
    =  [p | p<-nub (concat (map neighbs ps)),
                isEmpty ps p,
                liveneighbs ps p == 3]
```

Visualização

Uma função para fazer a animação de *n* gerações da colónia partindo duma configuração inicial.

```
life :: Cells -> Int -> IO ()
life ps n
  | n>0 = do { cls
             ; printCells ps
             ; wait 500
             ; life (nextgen ps) (n-1)
             }
  | otherwise =  return ()
```

Esta função não devolve um resultado útil — o objetivo é fazer animação no terminal.

Visualização (cont.)

Funções auxiliares de IO:

```
cls :: IO ()                          -- limpar o terminal
printCells :: Cells -> IO ()          -- mostrar a colónia
wait :: Int -> IO ()                  -- esperar (ms)
```

Usamos:

- função `usleep` para esperar (*standard* POSIX);
- sequências ANSI de controlo do terminal (`http://en.wikipedia.org/wiki/ANSI_escape_code`) para limpar a posicionar o texto.

Sumário

- Uma implementação simples do jogo do vida de Conway.
- Separação entre computação e interação patente nos tipos de funções; por ex:

```
liveneighbs :: Cells -> Pos -> Int      -- computação
nextgen :: Cells -> Cells               -- computação
printCells :: Cells -> IO ()            -- visualização
life :: Cells -> Int -> IO ()             -- interação
```

- Facilita a compreensão e extensão do programa.

Extensões

- Ler a configuração inicial da entrada padrão.
- Contar o número de células ao longo das gerações.
- Detetar casos especiais:
  - todas as células mortas;
  - repetições (naturezas mortas);
  - ciclos de periódo fixo.
- Melhorar a visualização: símbolos Unicode, cores, etc.
- Configuração inicial aleatória (usando `randomRIO`)

```
import System.Random
main = do x<-randomRIO (1,10) -- entre 1 e 10
          print x
```