Linguagens de Programação - Campo Minado - Haskell

UFJF

Matheus Gomes Luz Werneck (201835037)
Pedro Henrique Almeida Cardoso Reis (201835039)

Juiz de Fora, 11 de Junho de 2023

Contents

1 Introdução

Este relatório detalha a implementação completa e funcional do jogo Campo Minado utilizando a linguagem de programação Haskell, uma linguagem de programação funcional.

2 Instruções para o uso do projeto

Para jogar o jogo primeiramente é necessário ter o *ghci* instalado em sua máquina e o *cabal*. É recomendável antes de executar o projeto utilizar o comando *cabal clean*. A compilação do jogo é simples, para isso, basta: executar o comando **cabal build** e depois **cabal run**

3 Exemplo de como jogar

Para jogar a nossa versão do Campo Minado é bem simples. Segue abaixo a lista de comandos:

- **Abrir uma célula:** Para abrir uma célula basta digitar a coluna e a posição da linha;
- **Posicionar uma bandeira:** Para marcar uma célula, é necessário digitar o caractere + seguido da Coluna e Linha que você deseja marcar;
- **Remover bandeira:** Para remover a marcação de uma célula, digite o caractere - seguido da Coluna e Linha que você deseja remover a marcação;

É importante ressaltar que é necessário colocar um espaço entre cada caractere digitado, caso contrário, o código quebra. Veja a imagem abaixo como exemplo de como jogar:

4 Sobre o desenvolvimento

4.1 Tabuleiro

Durante a implementação do tabuleiro, trabalhar com as Estruturas de Dados nativas da linguagem Haskell estava causando alguns problemas. Diante disso, encontramos o tipo de dados **Cell**, representando assim uma célula do jogo e facilitando o nosso progresso no desenvolvimento do código.

- **isMine**: Indica se a célula contém uma mina ou não;
- **isOpen**: Indica se a célula está aberta ou fechada;
- **isFlagged**: Indica se a célula está marcada com uma bandeira ou não;

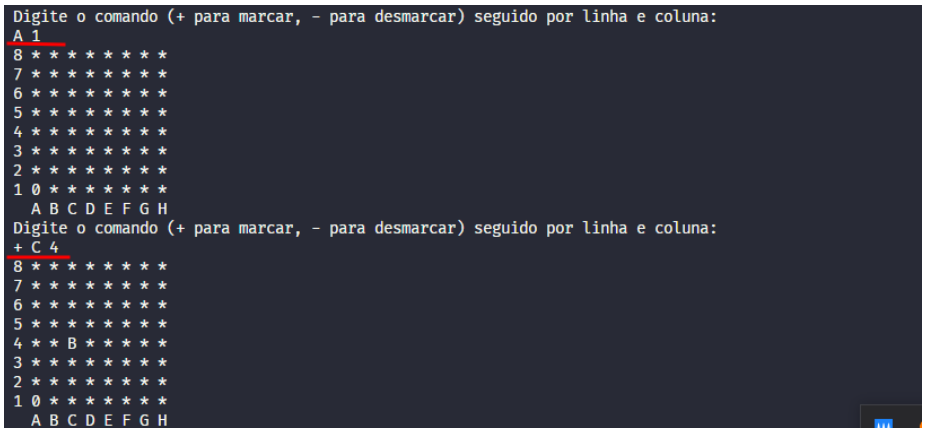


Figure 1: Exemplo de uma jogada

- **nearbyMines**: Representa o número de minas adjacentes à célula. Esse valor é calculado durante o jogo e indica quantas minas estão presentes nas células vizinhas.

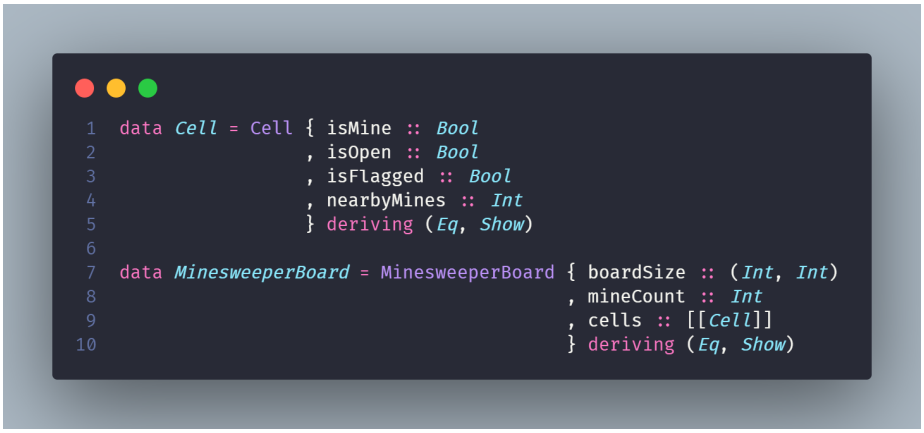
Essas propriedades são utilizadas para controlar o estado de cada célula no jogo Campo Minado. Através delas, é possível determinar se uma célula contém uma mina, se está aberta, se está marcada com uma bandeira e quantas minas estão próximas a ela.

Temos também `MinesweeperBoard`, representando o tabuleiro do jogo e que possui os seguintes campos:

- `boardSize`: é uma tupla que indica o número de linhas e colunas do tabuleiro.
- `mineCount`: indica o número total de minas no tabuleiro.
- `cells`: Matriz 2D;

4.2 Posicionamento das bombas

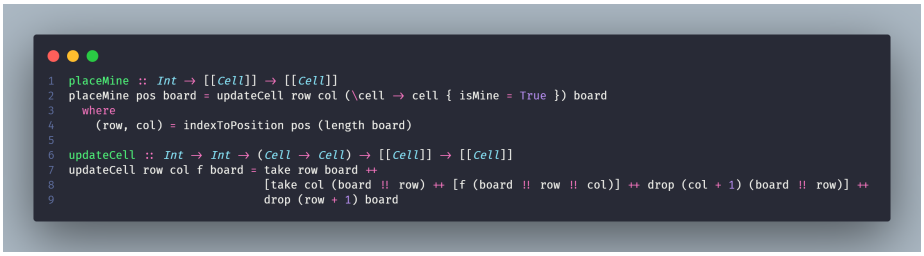
Para fazer o posicionamento das bombas no jogo primeiramente geramos uma lista de posições aleatórias usando a função *randomRs* do módulo *System.Random*. Em seguida, é criado um tabuleiro vazio usando a função *createEmptyBoard*, que retorna uma matriz 2D de células preenchidas com valores iniciais padrão. Para cada posição gerada aleatoriamente, a função *placeMine* é chamada para colocar uma mina na célula correspondente do tabuleiro. Essa mesma função utiliza a função *updateCell* para atualizar a célula na posição específica, marcando-a como uma mina. Dessa forma, as bombas são posicionadas aleatoriamente no tabuleiro, garantindo que não haja mais bombas do que o número especificado pelo jogador e que elas sejam distribuídas de forma aleatória em diferentes células do tabuleiro.



```haskell
data Cell = Cell { isMine :: Bool
                 , isOpen :: Bool
                 , isFlagged :: Bool
                 , nearbyMines :: Int
                 } deriving (Eq, Show)

data MinesweeperBoard = MinesweeperBoard { boardSize :: (Int, Int)
                                         , mineCount :: Int
                                         , cells :: [[Cell]]
                                         } deriving (Eq, Show)
```

Figure 2: Exemplo da montagem do tabuleiro



```haskell
placeMine :: Int -> [[Cell]] -> [[Cell]]
placeMine pos board = updateCell row col (\cell -> cell { isMine = True }) board
  where
    (row, col) = indexToPosition pos (length board)

updateCell :: Int -> Int -> (Cell -> Cell) -> [[Cell]] -> [[Cell]]
updateCell row col f board = take row board ++
                             [take col (board !! row) ++ [f (board !! row !! col)] ++ drop (col + 1) (board !! row)] ++
                             drop (row + 1) board
```

Figure 3: Exemplo da montagem do tabuleiro

4.3 Jogadas e tratamentos

A função *PlayGame* é responsável por iniciar e gerenciar o loop principal do jogo do campo minado. Ela recebe como argumento o tabuleiro do jogo representado pelo tipo MinesweeperBoard. O *case* utilizado nessa estrutura serve para fazer o tratamento do comando digitado. Ela verifica o valor de *firstChar* e executa o bloco de código correspondente a cada caso. Nessa mesma função também fazemos o tratamento da jogada, como por exemplo, marcar uma posição já marcada, abrir uma posição já aberta e executar uma ação fora dos índices do tabuleiro.

```haskell
playGame :: MinesweeperBoard -> IO ()
playGame board = do
  putStrLn "Digite o comando (+ para marcar, - para desmarcar) seguido por linha e coluna:"
  positionStr <- getLine
  let firstChar = head positionStr
  case firstChar of
    _ | isUpperCase firstChar  -> do
      let (rowChar, colPos) = parsePosition positionStr
      let row = colPos - 1
      let col = alphabetNumber rowChar
      if validatePosition (row, col) board
        then do
          let cell = getCell (row, col) board
          if isOpen cell
            then do
              putStrLn "Esta posição já está aberta!"
              playGame board
            else if isFlagged cell
                   then do
                     putStrLn "Esta posição está marcada!"
                     playGame board
                   else do
                     let newBoard = openCell (row, col) board
                     printBoard newBoard
                     if isMine (getCell (row, col) newBoard)
                       then do
                         putStrLn "Game Over! Você foi explodido!"
                         showBombs newBoard  -- Mostra as bombas em caso de derrota
                       else if allCellsOpened newBoard
                              then do
                                putStrLn "Parabéns! Você venceu!"
                                return ()  -- Encerra o jogo em caso de vitória
                              else playGame newBoard
        else do
          putStrLn "Posição inválida!"
          playGame board
    '+' -> do
      let (rowChar, colPos) = parsePositionCommand positionStr
      let row = colPos - 1
      let col = alphabetNumber rowChar
      if validatePosition (row, col) board
        then do
          let cell = getCell (row, col) board
          if isOpen cell
            then do
              putStrLn "Esta posição já está aberta!"
              playGame board
            else if isFlagged cell
                   then do
                     putStrLn "Esta posição já está marcada!"
                     playGame board
                   else do
                     let newBoard = flagCell (row, col) board
                     printBoard newBoard
                     if allMinesFlagged newBoard
                       then do
                         putStrLn "Você marcou todas as bombas corretamente!"
                         if allCellsOpened newBoard
                           then do
                             putStrLn "Parabéns! Você venceu!"
                             return ()  -- Encerra o jogo
                           else playGame newBoard
                       else playGame newBoard
        else do
          putStrLn "Posição inválida!"
          playGame board
```

Figure 4: Trecho do código para Jogada e tratamentos

5 Dificuldades ao longo da implementação

O fato de Haskell ser uma linguagem de paradigma funcional dificultou bastante a implementação, uma vez que estamos acostumados a programar em linguagens imperativas como Java, JavaScript e Python. Além disso, a criação do tabuleiro foi bem desafiador, já que como mencionado no início desse relatório, tivemos alguns problemas em usar as Estruturas de Dados nativas de Haskell. Com esses problemas gastamos muito tempo em busca de encontrar uma solução que se encaixava no nosso trabalho.