

# Relatório TP2

<b>Grupo.....</b>	<b>1</b>
<b>Início do Jogo:.....</b>	<b>1</b>
<b>Decisões de projeto.....</b>	<b>1</b>
Início.....	1
Criação da Mesa.....	2
Revelar e Marcar Bombas.....	5
Condições de Fim de Jogo.....	9
Simulação.....	10
Vitória:.....	11
Derrota:.....	12

## Grupo

- Filipe Brinati Furtado - 201865563C
- Lucca Oliveira Schröder - 201765205C

## Início do Jogo:

Para executar o jogo, primeiramente é necessário fazer um set package, já que as funções isDigit, intercalate e mapMaybe estão dentro de um pacote escondido do Data. Então o seguinte comando é necessário:

```
- :set -package base
```

Com isso, é só realizar a chamada da função main e seguir os comandos que o jogo te passa.

```
- main
```

## Decisões de projeto

### Início

Inicialmente, foi decidido como a construção da mesa seria. Como essa compartilhava da mesma estrutura do trabalho anterior, foi primeiramente tentado realizar o mesmo modelo de “espaços ocupados” usados no trabalho anterior. No entanto, ao correr do

desenvolvimento do jogo e conhecimento da linguagem utilizada, foi notado que nesse caso seria melhor fazer o uso de uma matriz para representar a mesa e atualizar os espaços de acordo com os comandos entregues pelo jogador. Com isso, nós fizemos inicialmente os tipos de dado Board e Cell, enquanto os outros foram definidos ao longo do desenvolvimento:

```
-- Definição dos tipos de dados
```

```
type Coord = (Int, Int)

data Cell = Empty | Number Int | Bomb | MarkedBomb | ActualBomb
deriving (Eq)

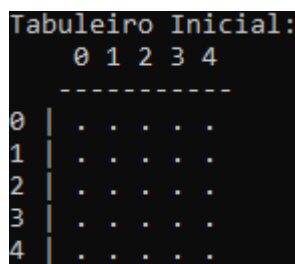
data Action = Reveal | Mark | UnMark

type Board = [[Cell]]
```

## Criação da Mesa

Com essas definições, nós começamos a desenvolver a mesa, aplicando as bombas e as funções de impressão. Inicialmente, foi notado que no trabalho é pedido que a mesa seja representada tanto por números quanto letras, no entanto, também foi percebido que o tamanho da mesa não é necessariamente limitado, já que o jogador passa o tamanho do tabuleiro inicialmente.

Com isso, em vez de limitar o jogador, foi decidido em usar números tanto para as linhas e colunas, e, para facilitar a decisão do jogador, na representação da mesa é apresentado a linha e coluna como a seguir:



```
Tabuleiro Inicial:
  0 1 2 3 4
  -----
0 | . . . . .
1 | . . . . .
2 | . . . . .
3 | . . . . .
4 | . . . . .
```

Além disso, como pode ser visto na imagem, os espaços não revelados ou marcados são representados por pontos. A criação da mesa e as funções de representações são apresentadas a seguir:

```
-- Funções de criação do tabuleiro
```

```
createEmptyBoard :: Int -> Int -> Board
```

```

createEmptyBoard rows cols = replicate rows (replicate cols Empty)

placeBombs :: Int -> Int -> Int -> Board -> IO Board
placeBombs rows cols numBombs board = do
    bombCoords <- generateBombCoords rows cols numBombs
    return (foldr (\c acc -> updateCell c Bomb acc) board bombCoords)

generateBombCoords :: Int -> Int -> Int -> IO [Coord]
generateBombCoords rows cols numBombs = do
    gen <- newStdGen
    let allCoords = [(x, y) | x <- [0 .. rows - 1], y <- [0 .. cols - 1]]
        shuffledCoords = shuffleBombs gen allCoords
    return (take numBombs shuffledCoords)

shuffleBombs :: StdGen -> [a] -> [a]
shuffleBombs gen xs = go gen xs []
    where
        go _ [] acc = acc
        go g xs acc =
            let (n, newG) = randomR (0, length xs - 1) g
                elem = xs !! n
                remaining = take n xs ++ drop (n + 1) xs
            in go newG remaining (elem : acc)

```

Como pode ser visto, a mesa é primeiramente criada vazia, ocupando todos os espaços disponíveis com o dado “Empty” e depois as bombas são colocadas no tabuleiro, sempre retornando o novo tabuleiro gerado.

Além disso, antes da geração da mesa, a função main faz a chamada da função de criação para fazer a configuração do início do jogo. Essas configurações e chamadas podem ser vistas nas seguintes funções:

```

-- Execução do jogo

main :: IO ()
main = do
    putStrLn "Bem-vindo ao jogo Campo Minado!"
    putStrLn "Digite o número de linhas do tabuleiro:"
    rows <- readLn
    putStrLn "Digite o número de colunas do tabuleiro:"
    cols <- readLn
    putStrLn "Digite o número de bombas:"
    numBombs <- readLn
    playGame rows cols numBombs

```

```

playGame :: Int -> Int -> Int -> IO ()
playGame rows cols numBombs = do
    let maxBombs = rows * cols `div` 2
        adjustedBombs = min numBombs maxBombs
    putStrLn $ "Número de Bombas: " ++ show adjustedBombs
    emptyBoard <- return $ createEmptyBoard rows cols
    board <- placeBombs rows cols adjustedBombs emptyBoard
    putStrLn "Tabuleiro Inicial:"
    printBoard board
    playTurn rows cols board

```

*-- Funções de impressão do tabuleiro*

```

printBoard :: Board -> IO ()
printBoard board = do
    let rows = length board
        cols = length (head board)
    putStrLn $ "      " ++ unwords (map show [0 .. cols - 1])
    putStrLn $ "    " ++ replicate (cols * 2 + 1) '-'
    mapM_ (printRow cols) (zip [0 ..] board)

```

```

printRow :: Int -> (Int, [Cell]) -> IO ()
printRow cols (rowIdx, cells) = do
    putStr $ show rowIdx ++ " | "
    mapM_ (putStr . cellToChar) cells
    putStrLn ""

```

```

cellToChar :: Cell -> String
cellToChar cell =
    case cell of
        Empty -> ". "
        Number n -> show n ++ " "
        Bomb -> ". "
        MarkedBomb -> "+ "
        ActualBomb -> "+ "

```

Também é possível ver a limitação do número de bombas na função “playGame”. Com essa execução, o estado do jogo deve ficar o seguinte:

```

Bem-vindo ao jogo Campo Minado!
Digite o número de linhas do tabuleiro:
5
Digite o número de colunas do tabuleiro:
5
Digite o número de bombas:
20000
Número de Bombas: 12
Tabuleiro Inicial:
  0 1 2 3 4
  -----
0 | . . . . .
1 | . . . . .
2 | . . . . .
3 | . . . . .
4 | . . . . .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):

```

## Revelar e Marcar Bombas

Primeiramente, para marcar as bombas, foi criada uma função que altera a informação das células para “MarkedBomb”, no entanto, apenas mudar a informação de uma célula não salva a informação do que essa era anteriormente. Com isso, foi inicialmente tentado colocar um número no dado como pode ser visto no dado “Number”, mas foi depois decidido criar um dado só para o caso de bombas e manipulado como pode ser visto no código a seguir:

```

markCell :: Coord -> Board -> Board
markCell coord@(x, y) board =
  let cell = getCell board coord
  in case cell of
    Empty -> updateCell coord MarkedBomb board
    Bomb -> updateCell coord ActualBomb board
    _ -> board

unmarkCell :: Coord -> Board -> Board
unmarkCell coord@(x, y) board =
  let cell = getCell board coord
  in case cell of
    MarkedBomb -> updateCell coord Empty board
    ActualBomb -> updateCell coord Bomb board
    _ -> board

```

Em seguida, foi começado a desenvolver a função de revelar bombas. Essa se apresentou um pouco mais complicada por causa da necessidade de gerar uma função de leitora, a qual depois seria utilizada para a função de marcar bombas, e as condições de erro requeridas na descrição do trabalho. Para isso, nós temos a função “playTurn”, a qual inicializa a análise de todas essas condições:

```

playTurn :: Int -> Int -> Board -> IO ()

```

```

playTurn rows cols board = do
  putStrLn "Digite as coordenadas para revelar uma célula ou
  marcar/desmarcar uma célula (linha coluna ação):"
  input <- getLine
  case parseInput input of
    Just (coord, action) ->
      case action of
        Reveal -> do
          let newBoard = revealCell coord rows cols board
          if isBomb (getCell newBoard coord)
            then do
              putStrLn "Você perdeu! O jogo acabou."
              putStrLn "Tabuleiro Final:"
              printBoard newBoard
            else do
              putStrLn "Tabuleiro Atual:"
              printBoard newBoard
              if checkWin newBoard
                then do
                  putStrLn "Parabéns! Você venceu o jogo."
                  putStrLn "Tabuleiro Final:"
                  printBoard newBoard
                else playTurn rows cols newBoard
        Mark -> do
          let newBoard = markCell coord board
          putStrLn "Tabuleiro Atual:"
          printBoard newBoard
          playTurn rows cols newBoard
        UnMark -> do
          let newBoard = unmarkCell coord board
          putStrLn "Tabuleiro Atual:"
          printBoard newBoard
          playTurn rows cols newBoard
    Nothing -> do
      putStrLn "Entrada inválida. Tente novamente."
      playTurn rows cols board

```

Para realizar a análise da entrada, as seguintes funções foram implementadas:

```

parseInput :: String -> Maybe (Coord, Action)
parseInput input =
  case words input of

```

```

[xStr, yStr, actionStr] -> do
  x <- parseCoord xStr
  y <- parseCoord yStr
  action <- parseAction actionStr
  return ((x, y), action)
_ -> Nothing

parseCoord :: String -> Maybe Int
parseCoord str =
  if all isDigit str then Just (read str) else Nothing

parseAction :: String -> Maybe Action
parseAction str =
  case str of
    "-" -> Just UnMark
    "+" -> Just Mark
    "r" -> Just Reveal
    _ -> Nothing

```

Com a ação desejada interpretada, existem 4 opções de ações que podem ser executadas: marcar, desmarcar, revelar e nada. Com isso, marcar e desmarcar foram apresentadas acima, sobrando apenas explicar o funcionamento da ação de revelar, que ocorre nas seguintes funções:

```

revealCell :: Coord -> Int -> Int -> Board -> Board
revealCell coord@(x, y) rows cols board =
  case getCell board coord of
    Empty ->
      if isRevealed coord board
      then board -- A célula já foi revelada, não faz nada
      else
        let newBoard = updateEmptyWithNumbers coord rows cols board
        in newBoard
    _ -> updateCell coord (getCell board coord) board

updateEmptyWithNumbers :: Coord -> Int -> Int -> Board -> Board
updateEmptyWithNumbers coord@(x, y) rows cols board =
  let bombCount = countAdjacentBombs coord rows cols board
  newBoard = updateCell coord (Number bombCount) board
  in newBoard

countAdjacentBombs :: Coord -> Int -> Int -> Board -> Int
countAdjacentBombs (x, y) rows cols board =
  let adjacentCoords = coordenadasAdjacentes (x, y) rows cols

```

```

        cells = map (getCell board) adjacentCoords
    in length $ filter isBomb cells

isRevealed :: Coord -> Board -> Bool
isRevealed coord board = case getCell board coord of
    Empty -> False
    Number _ -> True
    Bomb -> True
    MarkedBomb -> True
    ActualBomb -> True

-- Funções de manipulação do tabuleiro

updateCell :: Coord -> Cell -> Board -> Board
updateCell (x, y) cell board =
    take x board
    ++ [take y (board !! x) ++ [cell] ++ drop (y + 1) (board !! x)]
    ++ drop (x + 1) board

getCell :: Board -> Coord -> Cell
getCell board (x, y) = board !! x !! y

coordenadasAdjacentes :: Coord -> Int -> Int -> [Coord]
coordenadasAdjacentes (x, y) rows cols =
    let coords = [(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)]
    in filter (\(x', y') -> x' >= 0 && x' < rows && y' >= 0 && y' <
cols) coords

```

Com essas funções é possível fazer a revelação da célula desejada. Como a chamada dessa função é feita antes da análise de vitória ou derrota, foi necessário criar a condição na função `isRevealed` da célula ser uma bomba. Assim, se essa função retornar “True” nada acontece.

Caso essa seja falsa, é atualizado o valor da célula para o tipo “Number” e seu respectivo inteiro. Além disso, é possível notar na função “`coordenadasAdjacentes`” que apenas as células na vertical e horizontal são consideradas.

Um exemplo da execução do jogador é apresentado a seguir, unico detalhe a ser considerado é que para esse exemplo, as bombas estão sendo representadas por “\*”:



```

Bem-vindo ao jogo Campo Minado!
Digite o número de linhas do tabuleiro:
3
Digite o número de colunas do tabuleiro:
3
Digite o número de bombas:
1
Número de Bombas: 1
Tabuleiro Inicial:
  0 1 2
-----
0 | . * .
1 | . . .
2 | . . .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
0 0 r
Tabuleiro Atual:
  0 1 2
-----
0 | 1 * .
1 | . . .
2 | . . .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
1 0 r
Tabuleiro Atual:
  0 1 2
-----
0 | 1 * .
1 | 0 . .
2 | . . .

```

## Condições de Fim de Jogo

Em conclusão, dentro da condição reveal do “playTurn”, também é analisado se o jogador ganhou ou perdeu como apresentado a seguir:

```

Reveal -> do
  let newBoard = revealCell coord rows cols board
  if isBomb (getCell newBoard coord)
    then do
      putStrLn "Você perdeu! O jogo acabou."
      putStrLn "Tabuleiro Final:"
      printBoard newBoard
    else do
      putStrLn "Tabuleiro Atual:"
      printBoard newBoard
      if checkWin newBoard
        then do
          putStrLn "Parabéns! Você venceu o jogo."
          putStrLn "Tabuleiro Final:"
          printBoard newBoard
        else playTurn rows cols newBoard

```

Na condição de derrota, é checado se a célula é uma bomba a partir da função isBomb:

```
isBomb :: Cell -> Bool
isBomb cell = cell == Bomb
```

Na condição de vitória, é checado se todas as células que não são bombas ou bombas marcadas corretamente, estão numeradas:

```
checkWin :: Board -> Bool
checkWin board =
    all (\row -> all (\cell -> isBomb cell || isNumbered cell ||
isActualBomb cell) row) board

isBomb :: Cell -> Bool
isBomb cell = cell == Bomb

isActualBomb :: Cell -> Bool
isActualBomb cell = cell == ActualBomb

isNumbered :: Cell -> Bool
isNumbered cell = case cell of
    Number _ -> True
    _ -> False
```

No mais, se nenhuma das condições é atendida, o jogo continua.

## Simulação

A seguir é apresentado uma vitória e uma derrota respectivamente:

Vitória:

```
ghci> main
Bem-vindo ao jogo Campo Minado!
Digite o número de linhas do tabuleiro:
2
Digite o número de colunas do tabuleiro:
2
Digite o número de bombas:
40
Número de Bombas: 2
Tabuleiro Inicial:
  0 1
  ----
0 | . .
1 | . .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
0 0 r
Tabuleiro Atual:
  0 1
  ----
0 | 2 .
1 | . .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
1 0 +
Tabuleiro Atual:
  0 1
  ----
0 | 2 .
1 | + .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
0 1 +
Tabuleiro Atual:
  0 1
  ----
0 | 2 +
1 | + .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
0 1 -
Tabuleiro Atual:
  0 1
  ----
0 | 2 .
1 | + .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
0 1 +
Tabuleiro Atual:
  0 1
  ----
0 | 2 +
1 | + .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
1 1 r
Tabuleiro Atual:
  0 1
  ----
0 | 2 +
1 | + 0
Parabéns! Você venceu o jogo.
Tabuleiro Final:
  0 1
  ----
0 | 2 +
1 | + 0
```

Derrota:

```
ghci> main
Bem-vindo ao jogo Campo Minado!
Digite o número de linhas do tabuleiro:
2
Digite o número de colunas do tabuleiro:
2
Digite o número de bombas:
40
Número de Bombas: 2
Tabuleiro Inicial:
  0 1
  ----
0 | . .
1 | . .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
0 0 r
Tabuleiro Atual:
  0 1
  ----
0 | 1 .
1 | . .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
0 1 +
Tabuleiro Atual:
  0 1
  ----
0 | 1 +
1 | . .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
1 0 +
Tabuleiro Atual:
  0 1
  ----
0 | 1 +
1 | + .
Digite as coordenadas para revelar uma célula ou marcar/desmarcar uma célula (linha coluna ação):
1 1 r
Você perdeu! O jogo acabou.
Tabuleiro Final:
  0 1
  ----
0 | 1 +
1 | + .
```