Implementação do Jogo Campo Minado em Haskell

Diego Paiva e Silva

Junho, 2019

Abstract

O presente documento tem como objetivo esclarecer as principais decisões de projeto tomadas na implementação de uma versão do jogo *Campo Minado* em linguagem de programação funcional Haskell, referente ao trabalho prático da disciplina *DCC019 - Linguagem de Programação*.

1 Modelagem e Estruturas de Dados

O tabuleiro do jogo foi modelado em uma matriz quadrada n x $n \mid 1 \leq n \leq 26$, onde as linhas são rotuladas com caracteres do alfabeto (por isso o limite superior de n é igual a 26) e as colunas são rotuladas com digitos. O exemplo abaixo ilustra o estado **inicial** do tabuleiro de um jogo com $n = 6$:

A	*	*	*	*	*	*
B	*	*	*	*	*	*
C	*	*	*	*	*	*
D	*	*	*	*	*	*
E	*	*	*	*	*	*
F	*	*	*	*	*	*
	1	2	3	4	5	6

Figure 1: Tabuleiro com dimensão $n = 6$.

Foram definidos dois tipos de dados principais para modelar o jogo no computador:

1. *Mine*: Representa uma mina do jogo. A única informação relevante acerca de uma mina é sua posição no tabuleiro, ou seja, sua linha (um caracter do alfabeto, e.g *A*) e sua coluna (um digito, e.g *3*).

```
data Mine = Mine {
    row :: Char,
    col :: Int
} deriving (Eq, Show, Ord)
```

2. *Minesweeper*: É uma abstração do jogo em si, contendo o estado atual do tabuleiro (representado através de uma matriz de caracteres) e as minas secretas (representadas através de um lista de *Mine*).

```
data Minesweeper = Minesweeper {
    board :: [[Char]],
    mines :: [Mine]
} deriving (Eq, Show, Ord)
```

Com estes tipos de dados definidos torna-se possível controlar o estado corrente do tabuleiro e a distribuição das minas sorteadas no início do jogo.

2 Cabal

Como a implementação utiliza diferentes bibliotecas, é necessário a utilização do ***cabal*** (Common Architecture for Building Applications and Libraries) [1] para realizar a configuração, o build e a instalação dos pacotes e dependências do projeto.

3 Jogo

3.1 Inicialização

Para iniciar o jogo o jogador precisa passar, da raíz do projeto, exatamente dois argumentos por linha de comando, na seguinte ordem: dimensão do tabuleiro e número de minas. Exemplificando, o comando abaixo inicia um jogo num tabuleiro 4 x 4 com 5 minas:

```
cabal run 4 5
```

Caso o jogador inicie um jogo com um tabuleiro de dimensão fora do intervalo $[1, 26]$, o mesmo é automaticamente inicializado com a máxima dimensão possível ($n = 26$). O mesmo conceito se aplica à quantidade de minas, exceto que o valor deve estar definido dentro do intervalo válido $[1, \frac{n^2}{2}]$ e, caso isso não ocorra, o jogo é inicializado com $\frac{n^2}{2}$ minas.

3.2 Sorteio das Posições das Minas

Para distribuir as minas aleatoriamente pelo tabuleiro, utilizou-se a biblioteca ***System.random***. A partir da função ***randomR***, foi possível gerar um valor aleatório para a linha e coluna de cada mina. É importante ressaltar que foi implementado um mecanismo na função de geração que impede a criação de minas repetidas.

3.3 Interação com o Jogador

A cada turno, o jogador tem a opção de três movimentos distintos:

1. Abrir uma posição (e.g *A1*)
2. Marcar uma posição como mina (e.g *+D2*)
3. Desmarcar uma posição que está marcada como mina (e.g *-D2*)

Para certificar que o comando informado pelo jogador é um comando válido, foi utilizada a biblioteca de expressões regulares ***Text.Regex.PCRE***. A expressão regular responsável por validar a entrada depende da dimensão n do tabuleiro, do último algarismo μ de n e do rótulo α da ultima linha do tabuleiro, e pode ser expressa através do seguinte algoritmo:

if $n \leq 9$ **then**

`[+|-]?[(a-(toLower` α`))|(A-`α`)]([1-`μ`])`

else if $10 \leq n \leq 19$ **then**

`[+|-]?[(a-(toLower` α`))|(A-`α`)]([1-9]|1[0-`μ`])`

else

`[+|-]?[(a-(toLower` α`))|(A-`α`)]([1-9]|1[0-9]|2[0-`μ`])`

end if

A construção acima garante que o movimento do jogador sempre consistirá em uma operação válida sob uma posição existente do tabuleiro. Caso o jogador informe um movimento inapropriado, é solicitado ao mesmo um novo comando.

3.4 Detecção de Minas Próximas

Quando o jogador abre uma célula de linha L e coluna C que não contém uma mina, a célula deve informar a quantidade de minas vizinhas. Como o tabuleiro é modelado em uma matriz, torna-se trivial determinar quantas minas há na proximidade da célula aberta, bastando apenas:

1. Verificar se há alguma mina na lista de minas cuja linha seja L e a coluna seja $C+1$ ou $C-1$.
2. Verificar se há alguma mina na lista de minas cuja coluna seja C e a linha seja $L+1$ ou $L-1$.

Esta tarefa é desempenhada pela função `nearbyMines`.

3.5 Deteccção de Fim de Jogo

O jogo é encerrado se **(i)** o jogador abre uma posição que contém uma mina ou **(ii)** consegue abrir todas as células do tabuleiro. A condição **(i)** é extremamente simples de ser verificada, bastando apenas verificar se a posição informada pelo usuário é a posição de alguma mina da lista de minas. Já para verificar a condição **(ii)** foi implementada a função `filterBoard` (semelhante à função nativa `filter`) que retorna a matriz filtrada a partir de uma condição passada por uma função booleana. Quando o número de células diferentes de '*' é igual ao número total de células do tabuleiro, significa que todas as posições já foram abertas, sinalizando que o jogador venceu o jogo.

4 Referências

[1] `https://www.haskell.org/cabal/`