Paradigmas de Programação
Batalha Naval

Departamento de Informática
Universidade da Beira Interior

Data limite para entrega: Quinta-feira, 4 de Junho de 2020, 23h59m
Apresentações/Defesas: Sexta-feira, 5 de Junho de 2020

Código Base: Encontra-se disponível no Moodle o ficheiro TP_Base.zip que contém código base para a implementação deste trabalho. As indicações presentes nos ficheiros de código são para ser consideradas em conjunção com este enunciado.

Entrega: Via Moodle. Enviar um ficheiro zip com o seguinte conteúdo:

1. Todos os ficheiros do projecto. Todas as funções devem estar devidamente documentadas com comentários no código.
2. Um ficheiro README com a informação seguinte:
   - o nome e número de todos os elementos do grupo;
   - como correr o programa;
   - breve descrição sobre como funciona a solução apresentada e todos os detalhes que sejam relevantes para a sua compreensão;
   - eventuais alterações à implementação base fornecida;
   - extras implementados (se existentes).

   Devem ser evidenciados os esforços feitos para garantir que a solução segue o paradigma funcional.
3. Documento com as provas requerida na Secção 2.

A submissão do trabalho final no Moodle deverśer feita por apenas um aluno.

1 Batalha Naval (17 pontos)

Neste trabalho vão implementar o jogo da batalha naval em *Haskell* usando uma abordagem funcional. Batalha naval é um jogo de tabuleiro de dois jogadores, no qual os jogadores têm de adivinhar em que posições estão os navios do oponente. O objectivo é afundar todos os navios do jogador adversário.

Antes do início do jogo, cada jogador coloca os seus navios nas posições desejadas. O jogo consiste em jogadas alternadas em que cada jogador indica qual a coordenada/posição que pretende atacar[1].

O objectivo deste trabalho é implementar em *Haskell* uma solução que permita um humano jogar um jogo de batalha naval contra o computador. A implementação tem de ser feita usando o paradigma de programação funcional.

O jogo deve respeitar os seguintes pontos:

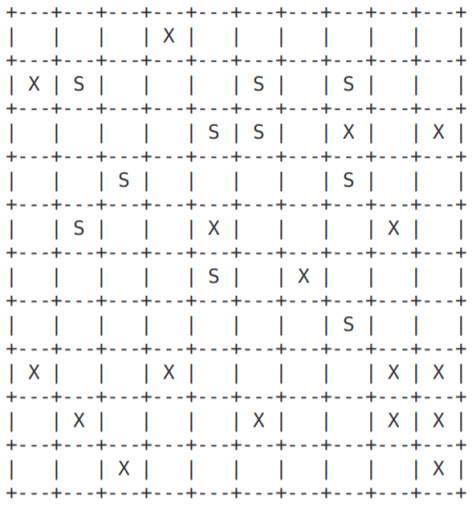
- O tamanho do tabuleiro é 10×10 mas todo o código tem de ser desenvolvido de forma a que esse tamanho possa ser alterado sem afectar o funcionamento do jogo. Este tamanho está definido no programa como `boardSize`.
- Os navios a serem colocados por ambos os jogadores são os seguintes:
  - Um porta-aviões (ocupa cinco posições);
  - Dois navios-tanque (ocupam quatro posições cada);
  - Três contratorpedeiros (ocupam três posições cada);
  - e Quatro submarinos (ocupam duas posições cada).

  O programa deve também ser desenvolvido de forma a que a alteração do número e tipo de navios não afecte o funcionamento do jogo.
- Os tipos de dados apresentados no código base têm de ser utilizados. Qualquer alteração deve ser devidamente justificada.
- A informação sobre as posições de cada tabuleiro tem de ser implementada como uma função (`BoardF`).
- Para a implementação do jogo devem utilizar pelo menos um *Monad Transformer*, por exemplo, `StateT` combinado com a monad `IO`.

[1] Para uma descrição mais detalhada do jogo podem consultar, por exemplo, `https://pt.wikipedia.org/wiki/Batalha_naval_(jogo)`

1.1 Colocação dos navios

- As posições que compõem um navio estão conectadas em linha recta.
- Os navios têm de ser colocados na sua totalidade dentro do tabuleiro (i.e. não podem ocupar posições fora do tabuleiro).
- Os navios não se podem sobrepor.
- Os navios podem ser colocados verticalmente ou horizontalmente.
- Coordenadas que já foram atacadas apresentam um S em caso de sucesso (i.e. se parte de um navio foi afundada) e um X em caso de insucesso.
  A figura seguinte mostra um exemplo de um tabuleiro durante um jogo:



1.2 O jogo

- O computador colocará todos os seus navios de forma automática.
- Ao jogador será pedida a coordenada inicial e final de cada navio (o programa deverá gerar as restantes coordenadas e certificar-se que o tamanho é o correcto para o tipo de navio – caso não o seja, deve informar o jogador e pedir novas coordenadas). O input tem de ser exactamente no seguinte formato:

  ```
  (1,1);(1,5)
  ```

  que representa um navio porta-aviões, sendo (1,1) a coordenada inicial do navio e (1,5) a coordenada final.
- Cada jogador indica na sua vez qual a posição que prentende atacar; o humano joga primeiro; o computador joga automaticamente logo de seguida.

- Após cada jogada (tanto do computador como do jogador) será apresentado no ecrã o estado actual do jogo (i.e. os dois tabuleiros) e uma mensagem que indica sucesso ou insucesso. A jogada seguinte só acontece depois de o jogador pressionar *Enter*.
- Deverá ser apresentada uma mensagem a indicar quando um navio é afundado, sempre que tal aconteça. Um navio é afundado quando todas as suas posições tiverem sido atacadas.
- A mesma posição não pode ser atacada mais do que uma vez por nenhum dos jogadores (computador ou humano). Se tal acontecer uma nova coordenada tem de ser pedida/gerada.
- O jogo termina quando um dos jogadores afunda todos os navios do oponente. O programa deve indicar quem é o vencedor.

Nota: Terão de criar todos os algoritmos para que o computador coloque os navios e jogue de forma automática. Uma solução aleatória é aceite como implementação básica. Um algoritmo bem pensado, eficaz (i.e. que torna o computador um jogador desafiante) e justificado obterá pontos extra (ver pontuações detalhadas abaixo).

1.3 Pontuações máximas para cada componente

As pontuações apresentadas são para funcionalidades implementadas na totalidade, sem quaisquer erros e de acordo com o detalhado neste documento e no código base. As pontuações podem também variar tendo em conta factores tais como a elegância da solução, abordagem funcional e validação do input do utilizador. Não utilizar uma abordagem funcional pode levar à perda total dos pontos.

- Implementação completa das funcionalidades de colocação dos navios e jogadas do humano: **5 pontos**
- Implementação completa das funcionalidades de colocação dos navios e jogadas do computador (implementação aleatória): **5 pontos**.
- Implementação de algoritmos mais avançados para colocação dos navios e jogadas do computador que tornam o computador um jogador desafiante: até **1.5 pontos**.
- Implementação do jogo completo com uma interface com o utilizador elegante e intuitiva: **3.5 pontos**.
- Extras: até **2 pontos** (e.g. interface gráfica, que pode ser web).

A implementação tem de ser feita usando o paradigma de programação funcional. A pontuação atribuída terá em conta não só a correcção da solução mas também a concisão, elegância, e legibilidade do código, bem como o uso apropriado do paradigma funcional.

Todo o programa deve ser desenvolvido de forma a que a interacção com o utilizador lide de forma apropriada com qualquer error de input ou inconsistência no estado do jogo. Será tomada em consideração a qualidade da interação com o utilizador.

Conduta: Submissões com código semelhante serão investigadas de acordo com o *Regulamento Disciplinar dos Estudantes da Universidade da Beira Interior*.

2 Provas (3 valores)

Considerem a seguinte definição de um operador ($\diamond$) de composição de monads:

```
◇ :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
(f ◇ g) x = (f x) >>= g
```

e as definições das seguintes funções:

```
mapm :: Monad m => (a -> b) -> (m a -> m b)
mapm f mx = do {x <- mx; return (f x)}

concatm :: Monad m => m(m a) -> m a
concatm mxs = do {x <- mxs; x}
```

As funções `mapm` and `concatm` podem ser definidas em termos do operador $\diamond$ da seguinte forma:

```
mapm f p = (id ◇ (return.f)) p
concatm ps = (id ◇ id) ps
```

Assumam o seguinte:

```
1. (f ◇ g) ◇ h = f ◇ (g ◇ h) -- ◇ e' associativo

2. (f ◇ g) . h = (f . h) ◇ g

3. return ◇ g = g
```

- **Provem** o seguinte:

  ```
  4. (return . h) ◇ g = g . h
  ```

- Usando as definições e leis apresentadas acima, **provem o seguinte**:

  ```
  mapm f . concatm = concatm . mapm (mapm f)
  ```

As provas devem ser apresentadas no formato utilizado nas aulas, ou seja:

$$\begin{array}{ll} & expressao1 \\ = & \quad \{ \quad \text{Justificação} \quad \} \\ & expressao2 \end{array}$$