Universidade do Minho
Escola de Engenharia
Departamento de Informática

Projecto de Laboratórios de Informática I

Licenciatura em Engenharia Informática

1º Ano — 1º Semestre

Ano Lectivo 2022/2023

— Fase 1 de 2 —

Data de Lançamento: 23 de Outubro de 2022

Data Limite de Entrega: 13 de Novembro de 2022

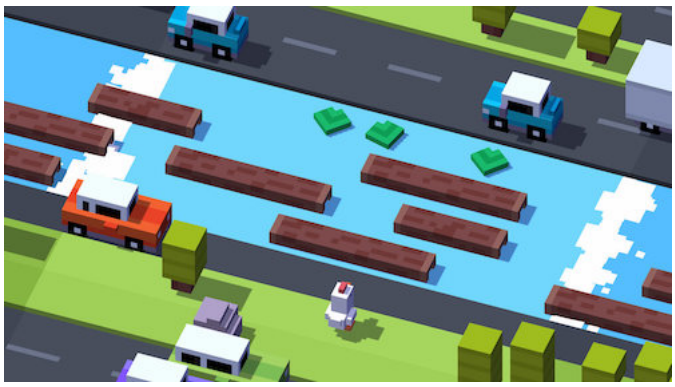
Outubro de 2022

1 Introdução

Neste enunciado apresentam-se as tarefas referentes à primeira fase do Projecto de Laboratórios de Informática I 2022/2023. O projecto consiste na implementação faseada de um pequeno jogo utilizando a linguagem *Haskell*. Por seu turno, cada fase consiste na implementação de um conjunto de *Tarefas*, definidas na secção seguinte.

1.1 Descrição do jogo

O jogo a implementar na presente edição é conhecido como *Crossy Road*[1]. O objectivo do jogo consiste em controlar um personagem ao longo de um mapa infinito, procurando chegar o mais longe possível.



Ao longo de um mapa o jogador terá que atravessar:

1. Rios, onde não poderá cair à água e, para isso, terá que saltar para cima de um dos troncos.
2. Estradas, onde terá que evitar ser atropelado por um carro.
3. Relva, onde terá que contornar as árvores.

Para evitar que o jogador simplesmente permaneça na mesma posição o tempo todo, o mapa irá automaticamente deslizar ao fim de um algum tempo. Caso o jogador fique para trás, isto é, deixe de estar visível no mapa, perde.

O jogador pode mover-se nas quatro direcções possíveis sem, contudo, sair do mapa. Com efeito, as únicas situações em que o jogador pode escapar do mapa são:

[1] https://poki.com/en/g/crossy-road

1. Quando o mapa desliza e o jogador fica para trás.

2. Quando o tronco em que o jogador se encontra eventualmente desaparece do mapa.

Em ambas as situações, o jogador perde e o jogo termina imediatemente.

Num mesmo rio ou estrada, troncos e carros deslocam-se numa direcção comum e a uma velocidade constante. Eventualmente estes obstáculos sairão do mapa por um dos lados, voltando a reaparecer no lado oposto, como se os limites do mapa fossem *wormholes*.

1.2 Tipos de dados

Apresentamos de seguida o modelo de jogo que deverá ter em conta na realização das tarefas propostas na secção seguinte. Será fornecido um módulo *Haskell* comum contendo estas definições preliminares, pelo que não necessita de as copiar.

1.2.1 Mapa

O *mapa* do jogo será representado por uma lista infinita de linhas. Cada linha denota um tipo de *terreno* e os *obstáculos* nela presentes.

```
type Largura = Int
data Mapa    = Mapa Largura [(Terreno, [Obstáculo])]
```

Em cada linha, ou seja, em cada par `(Terreno, [Obstáculo])`, o comprimento da lista de obstáculos corresponderá à largura do mapa. Isto significa que cada elemento da lista de obstáculos tem comprimento unitário. Por exemplo, se o mapa tem largura 5, então todas as listas de obstáculos no mapa terão comprimento 5.

Há três tipos de terrenos: *Rio*, *Estrada* e *Relva*.

```
type Velocidade = Int
data Terreno    = Rio Velocidade
                | Estrada Velocidade
                | Relva
```

O parâmetro *Velocidade* indica simultaneamente a velocidade a que os obstáculos no terreno em causa de deslocam e a direcção (da direita para a esquerda se o valor for negativo, ou da esquerda para a direita se o valor for positivo.) Assim, `Rio 3` indica que os troncos nesta linha se devem mover à velocidade

3 da esquerda para a direita, enquanto `Estrada -1` indica que os carros nesta linha se movem à velocidade 1 da direita para a esquerda.

Finalmente, representamos os obstáculos através do seguinte tipo:

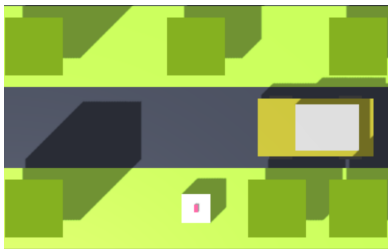
```
data Obstáculo = Nenhum
               | Tronco
               | Carro
               | Árvore
```

Implicitamente, cada tipo de terreno restringe os obstáculos que nele podem ocorrer. Por exemplo, não devemos ter carros em rios, troncos em estradas, *etc.* Com efeito, o único obstáculo que é admissível em qualquer tipo de terreno é `Nenhum`, que serve para denotar a ausência de obstáculo.

O mapa deve ser lido de cima para baixo. Ou seja, a primeira linha do mapa representa a linha mais acima do jogo. Para um exemplo completo, repare na Figura 1 que ilustra simultaneamente a representação interna e gráfica de um mapa. Note que o personagem não faz parte desta abstracção.

```
Mapa 5 [(Relva,      [Árvore, Nenhum, Árvore, Nenhum, Árvore])
       ,(Estrada -1, [Nenhum, Nenhum, Nenhum, Carro,  Carro])
       ,(Relva,      [Árvore, Nenhum, Nenhum, Árvore, Árvore])]
```

(a) Representação interna.



(b) Representação gráfica.

Figura 1: Exemplo de um mapa e sua representação interna.

1.2.2 Personagem

A abstracção para o *personagem* (ou *jogador*) do jogo deverá ter em conta a sua *posição* no mapa. Utilizamos para isso as coordenadas (x, y) usuais, onde a coordenada $(0, 0)$ denota o canto superior esquerdo do mapa.

```
type Coordenadas = (Int, Int)
```

```
data Jogador     = Jogador Coordenadas
```

A título de exemplo, no mapa da figura 1 o jogador encontrar-se-ia na posição $(2, 2)$, ou seja, na terceira linha, terceira coluna.

1.2.3 Jogo

Finalmente, um *jogo* é simplesmente composto pela informação do mapa e do personagem:

```
data Jogo = Jogo Jogador Mapa
```

2 Tarefas

2.1 Tarefa 1 – Validação de um mapa

O objectivo desta tarefa é implementar a função:

```
mapaVálido :: Mapa -> Bool
```

que verifica se um dado mapa não viola nenhuma das seguintes restrições:

1. Não existem obstáculos em terrenos impróprios, *e.g.* troncos em estradas ou relvas, árvores em rios ou estradas, *etc.*
2. Rios contíguos têm direcções opostas.
3. Troncos têm, no máximo, 5 unidades de comprimento.
4. Carros têm, no máximo, 3 unidades de comprimento.
5. Em qualquer linha existe, no mínimo, um "obstáculo" `Nenhum`. Ou seja, uma linha não pode ser composta exclusivamente por obstáculos, precisando de haver pelo menos um espaço livre.
6. O comprimento da lista de obstáculos de cada linha corresponde exactamente à largura do mapa.
7. Contiguamente, não devem existir mais do que 4 rios, nem 5 estradas ou relvas.

2.2 Tarefa 2 – Geração contínua de um mapa

O objectivo desta tarefa consiste em implementar a função:

```
estendeMapa :: Mapa -> Int -> Mapa
```

utilizando, para isso, duas outras funções auxiliares que deve também implementar:

```
próximosTerrenosVálidos   :: Mapa -> [Terreno]
próximosObstáculosVálidos :: Int  -> (Terreno, [Obstáculo]) -> [Obstáculo]
```

A função `próximosTerrenosVálidos` deve gerar a lista de terrenos passíveis de serem usados numa nova linha no topo do mapa dado. Ignore, para os propósitos desta função, o parâmetro *velocidade*, assumindo, para este, o valor 0. Por exemplo, quando o mapa dado é vazio, então todos os terrenos são válidos, *i.e.* tem-se:

```
próximosTerrenosVálidos (Mapa _ []) == [Rio 0, Estrada 0, Relva]
```

Contudo, se o topo do mapa já contiver – contiguamente – 4 rios, então, respeitando o critério 7 da *Tarefa 1*, não podemos ter outra linha de rio, e então:

```
próximosTerrenosVálidos (Mapa _ [(Rio _, _)
                                ,(Rio _, _)
                                ,(Rio _, _)
                                ,(Rio _, _)
                                ,_]) == [Estrada 0, Relva]
```

Analogamente, a função `próximosObstáculosVálidos` deve gerar a lista de obstáculos passíveis de serem usados para continuar uma dada linha do mapa. O parâmetro do tipo `Int` corresponde à largura do mapa. Nesta função, deve ter em atenção não só os obstáculos permitidos no tipo de terreno indicado como as regras a respeito do comprimento dos obstáculos, como descritas na *Tarefa 1*. Por exemplo, teríamos:

```
próximosObstáculosVálidos 10 (Rio _, []) == [Nenhum, Tronco]
```

ou

```
próximosObstáculosVálidos 10 (Estrada _, []) == [Nenhum, Carro]
```

Note-se que o parâmetro inteiro também condiciona esta função, no sentido em que se o comprimento da lista de obstáculos já atinge a largura do mapa, então mais nenhum obstáculo é possível adicionar:

```
próximosObstáculosVálidos 2 (Estrada _, [Carro, Nenhum]) == []
```

Finalmente, a função `estendeMapa` deve gerar e adicionar uma nova linha válida ao topo (*i.e.* primeira linha, visto de cima para baixo) de um dado mapa. Assuma que o mapa dado é válido. O parâmetro do tipo `Int` é um inteiro aleatório (no intervalo $[0, 100]$) que pode usar para acrescentar alguma pseudo-aleatoriedade à geração da nova linha. Lembre-se de definir velocidades para os terrenos gerados.

3 Tarefa 3 – Movimentação do personagem e obstáculos

O objectivo desta tarefa é implementar a função:

```
animaJogo :: Jogo -> Jogada -> Jogo
```

que movimenta os obstáculos (de acordo com a *velocidade*) do terreno em que se encontram), e o personagem, de acordo com a *jogada* dada: as jogadas possíveis são dadas pelo seguinte tipo de dados:

```
data Direcção = Cima
              | Baixo
              | Esquerda
              | Direita
data Jogada   = Parado
              | Move Direcção
```

Note:

1. Numa estrada ou rio com velocidade v, os obstáculos devem mover-se $|v|$ unidades na direcção determinada.

2. As jogadas `Move Cima`, `Move Baixo`, *etc.* fazem com que o jogador se mova 1 unidade para cima, baixo, *etc,* respectivamente.

3. Mesmo quando o jogador não efectua qualquer movimento (*i.e.* a sua jogada é `Parado`), se o personagem se encontrar em cima de um tronco, o jogador acompanha o movimento tronco.

4. O jogador não consegue escapar do mapa através dos seus movimentos. Por exemplo, se o jogador se encontrar na linha de topo do mapa, então mover-se para cima não tem qualquer efeito, uma vez que já se encontra no limite do mapa.

5. Ao deslocar os obstáculos de uma linha, lembre-se que estes, assim que desaparecerem por um dos lados do mapa, devem reaparecer no lado oposto.

6. O efeito de deslize do mapa **não** é para ser implementado nesta função. Por outras palavras, as dimensões do mapa não devem sofrer alterações após invocar esta função.

4 Tarefa 4 – Determinar se o jogo terminou

O objectivo desta tarefa é implementar a função:

```
jogoTerminou :: Jogo -> Bool
```

que indica se o jogador perdeu o jogo, onde `True` significa que sim. Para isso deve testar se o jogador se encontra fora do mapa, na água, ou "debaixo" de um carro (*i.e.* na mesma posição de um carro.)

5 Entrega e Avaliação

A data limite para conclusão de todas as tarefas desta primeira fase é de **13 de Novembro de 2022** e a respectiva avaliação terá um peso de **40%** na nota final da UC. A submissão será feita automaticamente através do GitLab onde, nessa data, será feita uma cópia do repositório de cada grupo, sendo apenas consideradas para avaliação os programas e demais artefactos que se encontrem no repositório nesse momento. O conteúdo dos repositórios será processado por ferramentas de detecção de plágio e, na eventualidade de serem detectadas cópias, estas serão consideradas fraude dando-se-lhes tratamento consequente.

Para além dos programas *Haskell* relativos às tarefas será considerada parte integrante do projecto todo o material de suporte à sua realização armazenado no repositório do respectivo grupo (código, documentação, ficheiros de teste, *etc.*). A utilização das diferentes ferramentas abordadas no curso (como *Haddock* ou *git*) deve seguir as recomendações enunciadas nas respectivas sessões laboratoriais. A avaliação desta fase do projecto terá em linha de conta todo esse material, atribuindo-lhe os seguintes pesos relativos:

Componente	**Peso**
Avaliação automática da Tarefa 1	15%
Avaliação automática da Tarefa 2	15%
Avaliação automática da Tarefa 3	15%
Avaliação automática da Tarefa 4	15%
Avaliação qualitativa das tarefas	15%
Documentação do código	10%
Quantidade e qualidade dos testes	10%
Utilização do sistema de versões	5%

Os grupos de trabalho devem ser compostos por **dois elementos** pertencentes ao **mesmo turno PL**. A nota final será atribuída **independentemente** a cada membro do grupo em função da respectiva prestação. A avaliação automática será feita através de um conjunto de testes que não serão revelados aos grupos. A avaliação qualitativa incidirá sobre aspectos da implementação não passíveis de serem avaliados automaticamente (como a estrutura do código ou elegância da solução implementada).