ExciteBike

1ª fase do projeto de Laboratórios de Informática 1 2019/20

Introdução

Neste enunciado apresentam-se as tarefas referentes à primeira fase do projecto da unidade curricular de Laboratórios de Informática I. O projecto será desenvolvido por grupos de 2 elementos, e consiste em pequenas aplicações Haskell que deverão responder a diferentes tarefas (apresentadas adiante).

O objetivo do projeto deste ano é implementar um jogo 2D de corridas entre motas. A ideia geral do jogo é percorrer a pista o mais rápido possível, realizando saltos e evitando obstáculos.

Tarefas

Importante: Cada tarefa deverá ser desenvolvida num módulo Haskell independente, nomeado `Tarefa`***`n`***`_2019li1g`***`xxx`***`.hs`, em que ***`n`*** é o número da tarefa e ***`xxx`*** o número do grupo, que estará associado ao repositório SVN de cada grupo. Os grupos **não devem alterar** os nomes, tipos e assinaturas das funções previamente definidas, sob pena de não serem corretamente avaliados.

Para aceder pela primeira vez ao repositório SVN do seu grupo, pode executar o seguinte comando (substituindo `g999` pelo número do seu grupo e `a1` pelo nome do seu utilizador):

```
svn checkout svn://svn.alunos.di.uminho.pt/2019li1g999 --username 2019li1g999a1
```

Tarefa 1 - Gerar mapas

O objectivo desta tarefa é implementar um mecanismo de geração de mapas. Os inputs serão o número de pistas, o comprimento de cada pista, e um número inteiro positivo para usar como semente num gerador pseudo-aleatório. O output deverá ser um mapa impresso no formato abaixo descrito.

O gerador pseudo-aleatório a utilizar é dado pela função `geraAleatorios :: Int -> Int -> [Int]` que recebe o número de elementos a gerar, uma semente, e produz uma lista de valores pseudo aleatórios com valores entre 0 e 9. Esta função irá ser providenciada aos alunos.

Mapas

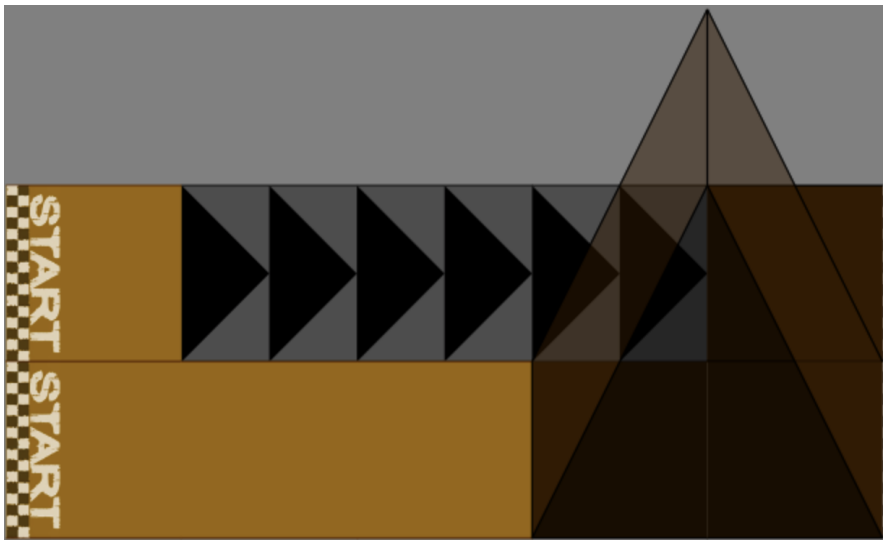
Um mapa é uma grelha ou matriz de peças, em que cada peça é uma recta de altitude constante ou uma rampa com uma altitude inicial e uma altitude final (altitudes são sempre números positivos). Uma peça do mapa pode ser feita de diferentes pisos, com diferentes propriedades.

```
type Mapa = [Pista]
type Pista = [Peca]
data Peca = Recta Piso Int | Rampa Piso Int Int
data Piso = Terra | Relva | Lama | Boost | Cola
```

Apresenta-se abaixo um exemplo de um mapa simples com 2 pistas, ambas de comprimento 5:

```
m = [[Recta Terra 0,Recta Boost 0,Recta Boost 0,Recta Boost 0,Recta
Lama 0]
    ,[Recta Terra 0,Recta Terra 0,Recta Terra 0,Rampa Lama 0 2,Rampa
Lama 2 0]
    ]
```

Visualmente, o mapa `m` corresponde à seguinte figura:



Pode visualizar graficamente este e outros mapas que defina em https://li1.lsd.di.uminho.pt/mapviewer/MapViewer.jsexe/run.html.

Mapas válidos

Para efeitos deste trabalho prático, só vão ser considerados mapas válidos.
Um mapa é válido quando:

1. Todas as pistas começam com Recta Terra 0;
2. Todas as pistas têm o mesmo comprimento;
3. Não há alturas negativas;
4. As alturas inicial e final de peças adjacentes na mesma pista são iguais. Por exemplo, a altura final de uma peça do tipo rampa terá que ser igual à altura inicial da peça que lhe sucede.

Mapas pseudo-aleatórios

O conteúdo das células não fixas (i.e. as células que não sejam as primeiras na sua pista) do mapa deve ser determinado a partir de uma sequência de números pseudo-aleatórios entre 0 e 9. Num mapa de dimensão 2x5, o número de células cujo conteúdo tem que ser gerado é 8 (não é 10 porque se assume que todas as pistas começam com a peça `Recta Terra 0`), pelo que para determinar o respectivo conteúdo iremos precisar de gerar uma sequência de 16 números aleatórios (2 para cada peça). Se a semente inicial for `1` (tal como utilizado para gerar o mapa `m`), essa sequência pode ser gerada da seguinte forma:

```
Tarefa1> geraAleatorios 16 1
[5,6,5,7,8,7,4,5,0,8,1,5,4,1,6,3]
```

O conteúdo de cada célula será determinado a partir do respectivo par de números aleatórios, sendo estes distribuídos pelas células sequencialmente, linha por linha. Por exemplo, num mapa de dimensão 2x5, a sequência anterior iria ser distribuída pelas células (cujo conteúdo não está fixo à partida) da seguinte forma:

-	5 6	5 7	8 7	4 5
-	0 8	1 5	4 1	6 3

A forma como se determina o **piso** de uma peça a partir do primeiro número aleatório é a seguinte:

Gama	**Conteúdo**
0 a 1	Piso de `Terra`.
2 a 3	Piso de `Relva`.
4	Piso de `Lama`.
5	Piso de `Boost`.
6 a 9	O piso da peça anterior.

A forma de determinar o **tipo** de uma peça tem como base a altura final da peça anterior e o segundo número aleatório. Mais especificamente, o segundo número aleatório dita uma diferença entre a altura final da peça anterior e a altura final da peça a ser gerada (de acordo com as regras relativas a mapas válidos, a altura inicial da peça a ser gerada tem que ser igual à altura final da peça anterior). Com base nesta diferença determinamos então o tipo da peça. A correspondência entre o número aleatório e a diferença de alturas é apresentado na tabela abaixo.

Gama	**Conteúdo**
0 a 1	`Rampa` que sobe com diferença de **Gama**+1 para a altura anterior.
2 a 5	`Rampa` que desce com diferença máxima de **Gama**-1 para a altura anterior. Note que existe uma altura mínima de 0. Caso a altura anterior e a altura atual sejam iguais, é uma `Recta`.

6 a 9	`Recta` com a altura anterior.

Por exemplo, dados dois números aleatórios `piso = 6` e `tipo = 3`, se a peça anterior é `Rampa Terra 4 6`, a nova peça a gerar será uma `Rampa` com o mesmo piso que a peça anterior que desce 2 unidades (`tipo-1`) em relação à altura anterior, obtendo a peça `Rampa Terra 6 4`. Dados os mesmos números aleatórios, se a peça anterior é `Rampa Relva 2 0`, a nova peça a gerar será uma `Recta` com o mesmo piso, obtendo a peça `Recta Relva 0`.

Note que os mapas gerados são sempre garantidamente válidos.

Funções a implementar

O objectivo desta tarefa é definir a função

```
gera :: Int -> Int -> Int -> Mapa
```

que gera um mapa pseudo aleatório com as dimensões desejadas. Por exemplo, deverá ser verdade que `gera 2 5 1 == m`, usando as os parâmetros `2 5 1` e o mapa `m` definidos acima. Pode consultar o resultado de `gera 2 5 1` no visualizador de mapas. Consulte a documentação online da Tarefa 1 (https://li1.lsd.di.uminho.pt/doc/src/Tarefa1_2019li1g000.html) para mais detalhes.

Tarefa 2 - Efetuar jogadas

O objectivo desta tarefa é, dada uma descrição do estado do jogo e uma jogada de um dos jogadores, determinar o efeito dessa jogada no estado do jogo.

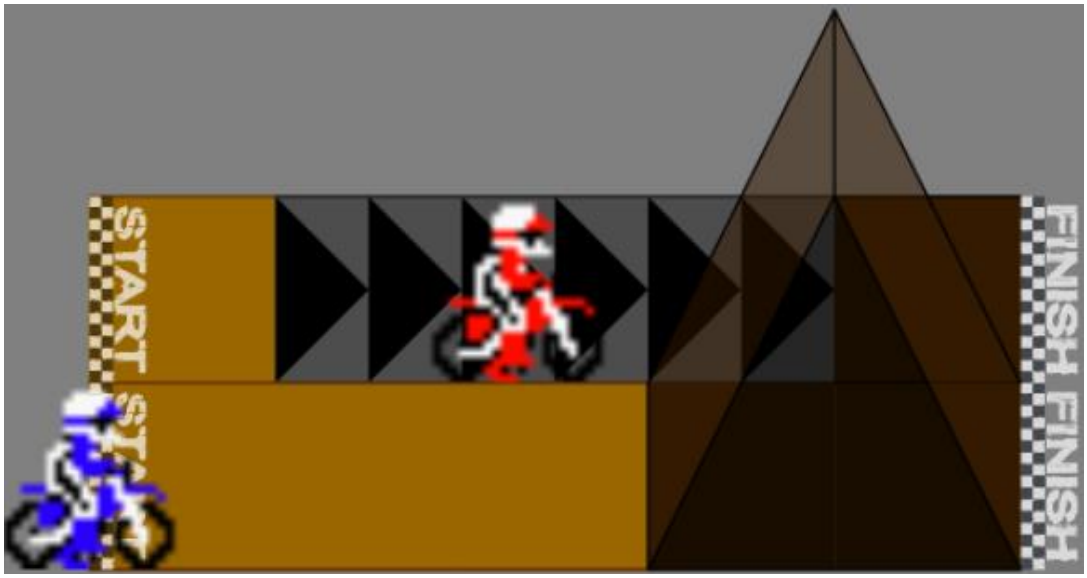
Jogadas

Uma jogada consiste numa movimentação numa dada direção, uma ordem para acelerar ou desacelerar, ou o disparo de cola para a retaguarda, de acordo com o seguinte tipo:

```
data Direcao = C | D | B | E
data Jogada = Movimenta Direcao | Acelera | Desacelera | Dispara
```

Estado do jogo

Considere a seguinte imagem que representa graficamente um estado do jogo:



Mais concretamente, o estado do jogo consiste num mapa e em informação adicional sobre jogadores, e é representado pelo seguinte formato:

```
data Estado = Estado
         { mapaEstado      :: Mapa
         , jogadoresEstado :: [Jogador] }
```

O estado de exemplo pode ser portanto representado pelo valor:

```
e = Estado m js
```

O estado `e` contém 2 jogadores (cujas cores correspondem às cores na figura):

```
js = [jogador1,jogador2]
jogador1 = Jogador 0 2.3 0 5 (Chao True)
jogador2 = Jogador 1 0 0 5 (Chao False)
```

Jogadores

Um jogador é identificado pelo seu índice na lista (havendo no máximo tantos jogadores quanto o número de pistas), e possui informação tal como a pista em que se encontra, a distância que percorreu no eixo do x[1], a sua (norma do seu vetor de) velocidade atual, a capacidade da sua pistola de cola (>=0), e o seu estado atual:

[1] Note que cada peça numa pista ocupa 1 unidade no eixo do x.

```
data Jogador = Jogador
  { pistaJogador :: Int
  , distanciaJogador :: Double
  , velocidadeJogador :: Double
  , colaJogador :: Int
  , estadoJogador :: EstadoJogador }
```

O estado do jogador indica se está atualmente no chão vivo, no chão morto ou no ar com uma determinada altura e inclinação. Um jogador pode estar morto por um determinado tempo, indicado por o seu `timeoutJogador`, em cujo caso, as jogadas efetuadas por este jogador não terão nenhum efeito no estado do jogo. Um jogador pode deslocar-se para uma direção onde não exista uma peça, desde que não esteja morto, neste caso a jogada não terá nenhum efeito no jogo.

```
data EstadoJogador
    = Chao { aceleraJogador :: Bool }
    | Morto { timeoutJogador :: Double }
    | Ar { alturaJogador :: Double, inclinacaoJogador :: Double,
gravidadeJogador :: Double }
```

O processamento de uma jogada deverá ser efetuado de acordo com as seguintes regras:

- Quando um jogador se movimenta para cima e para baixo muda de pista. Só o pode fazer quando está no `Chao`. Se a diferença absoluta de alturas entre as duas pistas for menor ou igual a 0.2, então o jogador transita de `Chao` para `Chao`. Quando a diferença for maior do que 0.2, o jogador esbarra (fica no estado `Morto` do lado de cá) ou cai (fica no estado `Ar alturaAnterior inclinacaoPecaAnterior 0` do lado de lá, em que `alturaAnterior` é a altura em que o jogador se encontrava e `inclinacaoPecaAnterior` é a inclinação da peça onde o jogador se encontrava). Quando um jogador morre, a mota para instantaneamente e o seu timeout para renascer é inicializado a `1.0`. Note que se um jogador cai quando passa de pista, o seu estado deverá ser alterado adequadamente.
- Quando um jogador se movimento para a esquerda ou direita altera a inclinação da mota em 15º na direção desejada. Só o pode fazer quando está no `Ar`. A inclinação mínima e máxima são -90º e 90º.
- Quando um jogador acelera/desacelera isso reflete-se no seu estado (mas só se estiver no `Chao`).

- Quando um jogador no `Chao` dispara cola, o piso da peça anterior àquela em que se situa fica com cola, e perde uma munição (mas só se ele tiver munições suficientes e se não se encontrar na primeira peça da pista).

Note que os efeitos de acelerar ou desacelerar só alteram o estado do jogador, a velocidade e distância dependem do passo do tempo e não são manifestados na jogada – isto será tratado na Tarefa 4.

Funções a implementar

O objectivo desta tarefa é definir a função

```
jogada :: Int -> Jogada -> Estado -> Estado
```

que efetua uma jogada para um determinado jogador sobre um determinado estado, retornando o novo estado após a jogada ser efetuada. Consulte a documentação online da Tarefa 2 (https://li1.lsd.di.uminho.pt/doc/src/Tarefa2_2019li1g000.html) para mais detalhes.

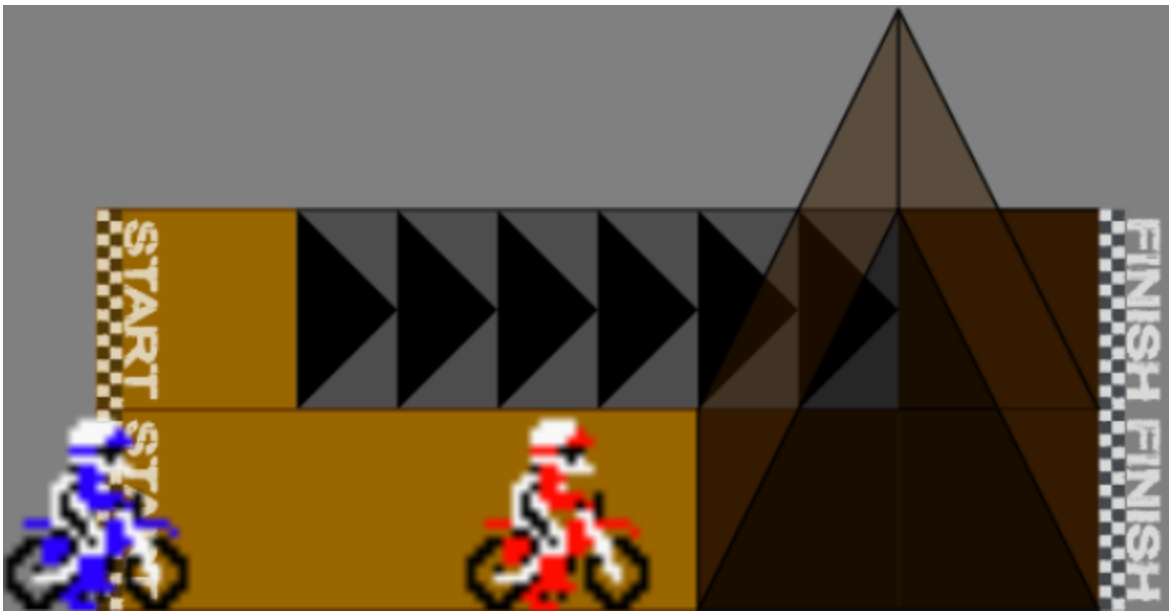
Por exemplo, dado o estado `e` do exemplo anterior, e a jogada `Movimenta B` para o jogador `0`, a função `jogada 0 (Movimenta B) e` deveria retornar o estado `e'`

```
e' = Estado m js'
```

onde `js'` é definido como

```
js' = [jogador1,jogador2]
jogador1 = Jogador 1 2.3 0 5 (Chao True)
jogador2 = Jogador 1 0 0 5 (Chao False)
```

Este novo estado pode ser visualizado na seguinte imagem.



Tarefa 3 - Desconstruir mapas

O objectivo desta tarefa é, dado um mapa cujo comprimento de cada pista é maior que 1[2], convertê-lo numa sequência de instruções a dar a um grupo de bulldozers (um por pista) que avançam da partida para construir o mapa em questão.

Instruções

Uma instrução dada a um bulldozer consiste em: andar em frente colocando um determinado piso; subir/descer uma dada diferença de altura colocando um determinado piso[3]; teleportar um número de posições para a frente (offset positivo) ou para trás (offset negativo); ou repetir um número de vezes um conjunto de instruções. Todas as instruções menos `Repete` podem ser dadas simultaneamente a um grupo de bulldozers. A representação de instruções é dada pelos seguintes tipos de dados:

```
Type Instrucoes = [Instrucao]
data Instrucao
    = Anda [Int] Piso
    | Sobe [Int] Piso Int
    | Desce [Int] Piso Int
    | Teleporta [Int] Int
```

[2] Anteriormente esta assunção não existia. Decidimos acrescentá-la para facilitar a implementação da Tarefa 3 pelos alunos.

[3] Anteriormente esta frase mencionava subir/descer “até uma dada altura”, o que aludia a alturas absolutas. No entanto, uma instrução contém sempre alturas relativas.

```
    | Repete Int Instrucoes
```

Uma instrução válida não pode repetir os identificadores dos bulldozers, por exemplo, a instrução `Anda [0,0] Terra` é considerada inválida.[4]

Padrões

Uma forma simples de resolver esta tarefa é converter cada peça numa instrução (de forma muito similar à Tarefa 1) do tipo `Anda`, `Sobe` ou `Desce`. Por exemplo, o mapa `m` acima definido pode ser desconstruído na seguinte sequência de instruções[5]:

```
[Anda [0] Boost,Anda [0] Boost,Anda [0] Boost,Anda [0] Lama,Anda [1]
Terra,Anda [1] Terra,Sobe [1] Lama 2,Desce [1] Lama 2]
```

Embora correcta, esta solução é óptima, no sentido em que utiliza sempre tantas instruções quantas as dimensões do mapa. Uma forma mais compacta e, portanto, económica no número de instruções, é identificando padrões que se repetem no mapa. É possível identificar três tipos principais de padrões, abaixo descritos.

Padrões horizontais

O padrão mais simples é a ocorrência de peças consecutivas iguais numa mesma pista. Por exemplo, no mapa `m` é possível encontrar dois casos de repetições horizontais: 1) 3 células seguidas da forma `Recta Boost 0` na pista `0`, e 2) 2 células da forma `Recta Terra 0` na pista `1`. Estes padrões podem ser melhor representados mediante a instrução `Repete 3 [Anda [0] Boost]` e `Repete 2 [Anda [1] Terra]`, respectivamente:

```
[Repete 3 [Anda [0] Boost],Anda [0] Lama,Repete 2 [Anda [1]
Terra],Sobe [1] Lama 2,Desce [1] Lama 2]
```

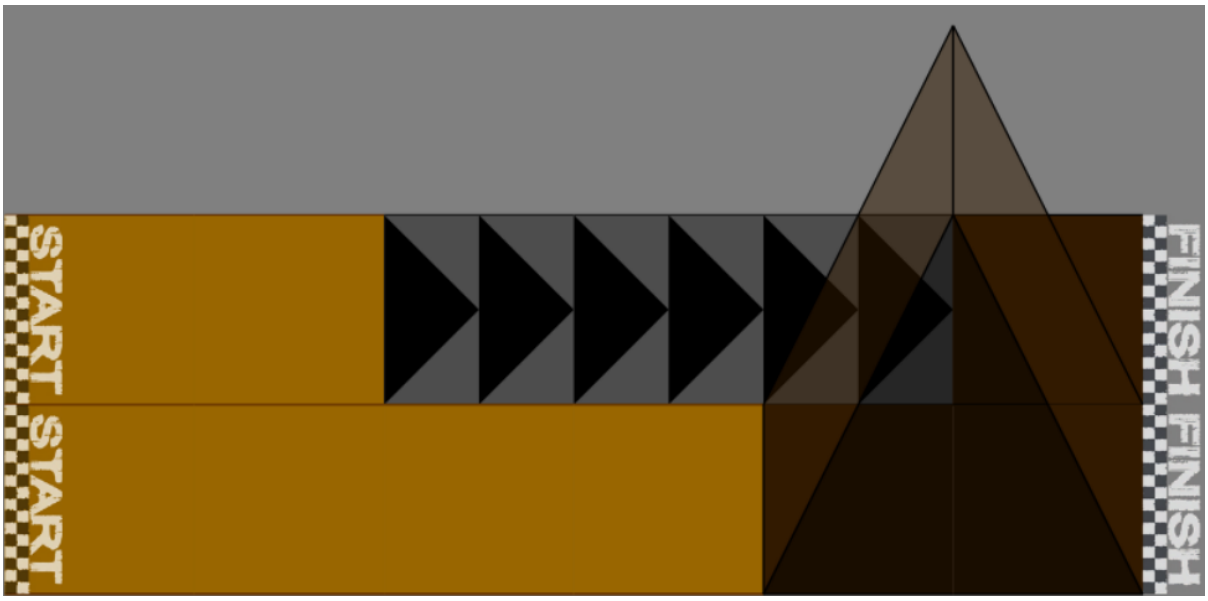
Padrões verticais

Outro padrão fácil de identificar são diferentes pistas com peças iguais em posições iguais. Por exemplo, considere o seguinte mapa:

---

[4] Esta restrição não estava a ser verificada no oráculo da Tarefa 3, o que foi adicionado recentemente.

[5] Note que existiu uma versão anterior do exemplo que assumia ser necessário dar instruções para a primeira célula de cada pista. Isto, no entanto, não é necessário pois a primeira célula de cada pista é sempre do tipo “Recta Terra 0”. Os exemplos seguintes também foram alterados de acordo com este aspecto.

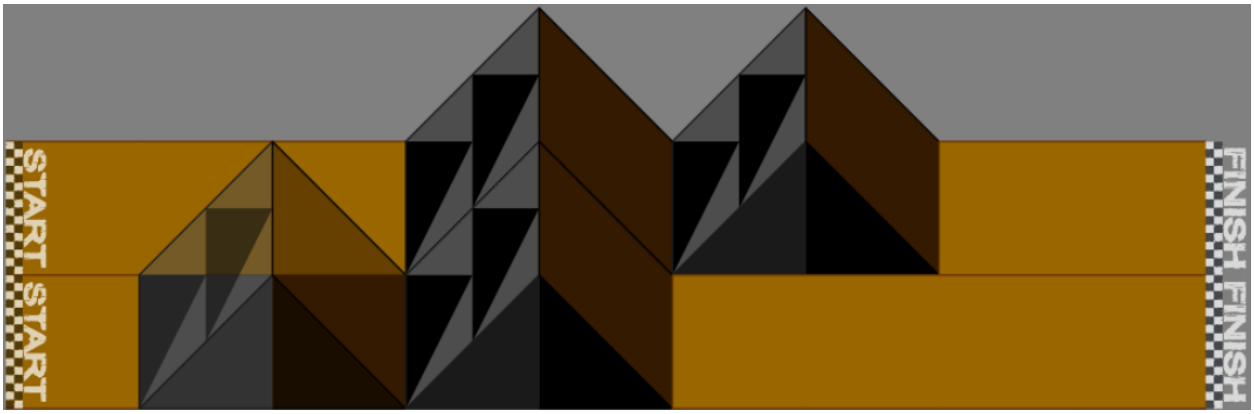


É possível identificar um caso de padrão vertical nas segundas células de ambas as pistas que são da forma `Recta Terra 0`. Neste caso é possível indicar aos bulldozers para avançar em ambas pistas ao mesmo tempo andando em frente e colocando o piso Terra, `[Anda [0,1] Terra]`, atualizando devidamente o resto das instruções (neste caso particular não se reduz o número de instruções):

```
[Anda [0,1] Terra,Repete 3 [Anda [0] Boost],Anda [0] Lama,Repete 2
[Anda [1] Terra],Sobe [1] Lama 2,Desce [1] Lama 2]
```

Padrões verticais desfasados

Um último padrão a salientar, mais avançado, consiste em identificar diferentes pistas com peças iguais mas em posições diferentes. Por exemplo, considere o seguinte mapa:



Podemos notar que tanto na pista 0 e 1, se repete a sequência de células `Rampa Boost 0 1, Rampa Lama 1 0, Rampa Boost 0 1, Rampa Lama 1 0`, a partir das posições 3 e 1 respectivamente. Uma forma de representar este mapa pode ser utilizando a instrução `Teleporta` da seguinte forma:

```
[Repete 8 [Anda [0,1] Terra],Teleporta [1] (-8), Teleporta [0] (-6),
Repete 2 [Sobe [0,1] Boost 1,Desce [0,1] Lama 1]]
```

A instrução `Teleporta`, mais avançada, pode ser utilizada para alinhar diferentes bulldozers nas suas pistas, a fim de aproveitar padrões verticais desfasados. Esta instrução sempre coloca o bulldozer na altura final da peça correspondente. Neste exemplo em concreto, está a instruir-se os bulldozers para a construir toda a pista com piso `Terra`, e teleportar-se para a posição necessária na sua pista de forma a construir o padrão de rampas detectado.

Funções a implementar

O objectivo desta tarefa é definir a função

```
descontroi    :: Mapa -> Instrucoes
```

cujo objectivo é, respectivamente, codificar o mapa para como uma sequência de instruções que, quando executadas, produzem o mesmo mapa. Note que pode visualizar graficamente a construção interativa de um mapa de acordo com uma sequência de instruções no visualizador online disponível em https://li1.lsd.di.uminho.pt/mapviewer/MapViewer.jsexe/run.html.
No entanto, pretende-se uma implementação de uma boa função de desconstrução, de forma a que o número de instruções a dar aos bulldozers seja o mínimo possível. Recomenda-se que construa a sua implementação de forma **incremental**, partindo de uma solução não otimizada e identificando sucessivamente padrões mais complexos. O critério de contabilização de instruções é dado pela função pré-definida `tamanhoInstrucoes :: Instrucoes -> Int`. Consulte a documentação online da Tarefa 3 (https://li1.lsd.di.uminho.pt/doc/src/Tarefa3_2019li1g000.html), para mais detalhes.

Sistema de *Feedback*

O projecto inclui também um Sistema de *Feedback*, alojado em http://li1.lsd.di.uminho.pt/group (o login pode ser feito através da tab Menu) que visa fornecer suporte automatizado e informações personalizadas a cada grupo e simultaneamente incentivar boas práticas no desenvolvimento de software (documentação, teste, controle de versões, etc). Esta página *web* permite aos alunos pedir feedback relativo ao seu trabalho de grupo presente no seu repositório SVN e fornece informação detalhada sobre vários tópicos, nomeadamente:

- Resultados de testes unitários, comparando a solução do grupo com um oráculo (solução ideal) desenvolvido pelos docentes;

- Relatórios de ferramentas automáticas (que se incentiva os alunos a utilizar) que podem conter sugestões úteis para melhorar a qualidade global do trabalho do grupo;
- Visualizadores gráficos de casos de teste utilizando as soluções do grupo e o oráculo.

Note que as credenciais de acesso ao Sistema de *Feedback* são as mesmas que as credenciais de acesso ao SVN.

Para receber *feedback* sobre as tarefas e qualidade dos testes, devem ser declaradas no ficheiro correspondente a cada tarefa as seguintes listas de testes:

```
testesT1 :: [(Int,Int,Int)]
testesT2 :: [(Int,Jogada,Estado)]
testesT3 :: [Mapa]
```

Como exemplo, podem-se utilizar os exemplos acima definidos como casos de teste:

```
testesT1 = [(2,5,1]
testesT2 = [(0,Movimenta B,e)]
testesT3 = [m]
```

Estas definições podem ser então colocadas no ficheiro correspondente a cada tarefa para que sejam consideradas pelo Sistema de *Feedback*. Note que uma maior quantidade e diversidade de testes garante um melhor *feedback* e ajudará a melhorar o código desenvolvido.

Entrega e Avaliação

A data limite para conclusão de todas as tarefas desta primeira fase é **18 de Novembro de 2019 às 23h59m59s (Portugal Continental)** e a respectiva avaliação terá um peso de 50% na nota final do projeto. A submissão será feita automaticamente através do SVN: nesta 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 3 tarefas, será considerada parte integrante do projeto todo o material de suporte à sua realização armazenado no repositório SVN do respectivo grupo (código, documentação, ficheiros de teste, etc.). A utilização das diferentes ferramentas abordadas no curso (como Haddock, SVN, etc.) 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	20%
Avaliação automática da Tarefa 2	20%
Avaliação automática da Tarefa 3	20%
Qualidade do código	15%
Qualidade dos testes	10%
Documentação do código usando o Haddock	10%
Utilização do SVN e estrutura do repositório	5%

A avaliação automática será feita através de um conjunto de testes que não serão revelados aos grupos. No caso da Tarefa 3, a avaliação automática também terá em conta o número de instruções atingido. A avaliação qualitativa incidirá sobre aspectos de qualidade de código (por exemplo, estrutura do código, elegância da solução implementada, etc.), qualidade dos testes (quantidade, diversidade e cobertura dos mesmos), documentação (estrutura e riqueza dos comentários) e bom uso do SVN como sistema de controle de versões.