Pacman

1ª fase do projeto de Laboratórios de Informática 1 - 2020/21

FAQ

P. Porque é que os fantasmas têm PlayerState na sua construção?
R. Eventualmente será possível jogar também com os fantasmas (segunda fase).

P. Se o Pacman comer um fantasma que se encontrava numa peça de comida, também é aplicado o efeito da comida no Pacman?
R. Sim, no caso de haver mais do que uma ação a realizar (morrer, comer, etc.), deve realizá-las todas.

P. Os fantasmas podem ter as mesmas coordenadas?
R. Sim, um fantasmas pode estar na mesma posição de outro, tal como no jogo original.

P. Na tarefa 2, tenho de me preocupar com a movimentação dos fantasmas?
R. Não têm de se preocupar com a movimentação dos fantasmas. No entanto, as funções têm de considerar que mais tarde poderá ser possível que os fantasmas também joguem.

P. Que jogadores devem aparecer no State após realizar uma jogada?
R. Têm que aparecer todos. Mas só um jogador se irá mover, que será aquela cujo id está no tipo Play. Se considerarem uma jogada (Move 1 R) e um estado com a lista
[
(Pacman (PacState (0,(1,2)……) ….)),
(Pacman (PacState (1,(1,3)……) ….))
]
, ambos os jogadores devem aparecer no estado de resultado da função play, mas apenas o jogador com id 1 (o segunda da lista) deve "jogar". Imaginem o jogo neste momento como se fosse por turnos, em que só joga um jogador de cada vez, mas todos estão em jogo. E não esquecer que uma jogada de um jogador pode influenciar também os outros jogadores (comer fantasmas, comer comida grande, etc).

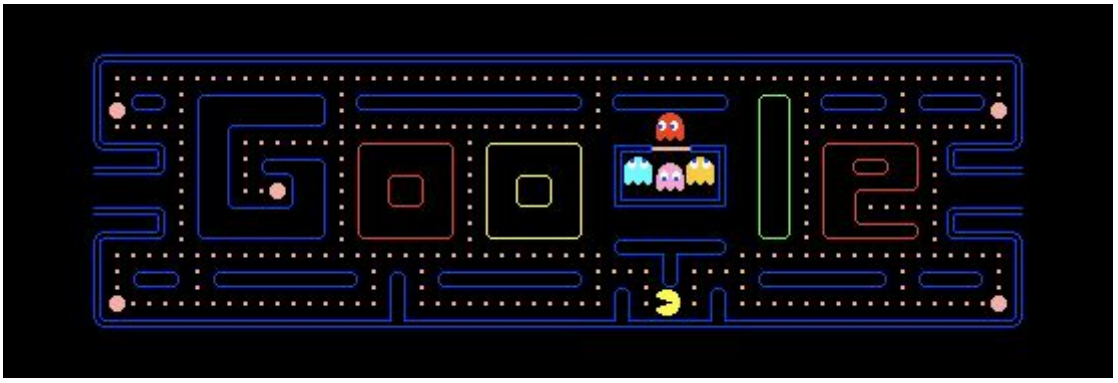
P. Quais as coordenadas da primeira peça do labirinto?
R. A primeira peça do labirinto é a parede que está no topo esquerdo e está nas coordenadas (0, 0).

P. O tipo de dados das peças permite colocar jogadores no labirinto. Devo considerar esse caso?
R. Não. O labirinto não contém jogadores. Os jogadores estão apenas na lista de jogadores.

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 semelhante ao Pacman. No jogo original o jogador é representado por uma cabeça redonda com uma boca que se abre e fecha, posicionado num labirinto repleto de pastilhas e fantasmas que o perseguem. O objetivo é comer todas as pastilhas sem ser apanhado pelos fantasmas, em ritmo progressivo de dificuldade.



Desenvolvimento de projeto

O trabalho será dividido em 6 tarefas, que estarão sujeitas a avaliação no decorrer do semestre. Para cada uma destas tarefas será estabelecido um prazo de entrega, dentro do qual os alunos deverão submeter a sua solução que cumpra os requisitos estipulados por estas.

O desenvolvimento deste projeto terá de ser feito colaborativamente com o auxílio de *GIT* e *GITHUB* (exemplo de tutorial em https://rogerdudler.github.io/git-guide/index.pt_BR.html).
Cada elemento do grupo de trabalho deverá criar uma conta (caso não possua) no *GITHUB* e fornecer o respetivo ID aos docentes, preenchendo um formulário que será disponibilizado para esse efeito. Será criado pelos docentes um repositório para cada grupo de trabalho, que irá conter o trabalho que irão desenvolver ao longo da UC e informação relativa à colaboração de cada elemento. Os docentes serão membros integrantes da equipa de desenvolvimento de cada trabalho e irão acompanhar semanalmente a evolução dos projetos. A entrega do trabalho e das tarefas será efetuada através desta plataforma e os

elementos do grupo serão avaliados individualmente de acordo com a sua contribuição para o repositório. Serão fornecidas algumas regras que os grupos deverão seguir para manter e estruturar o repositório, de forma a que o processo de avaliação e de execução dos trabalhos possa ser uniforme entre os grupos e possa ser efetuada de forma automática.

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

Tarefa 1 - Gerar labirintos

O objetivo desta tarefa é implementar um mecanismo de geração de labirintos. Os inputs serão o número de corredores (horizontais) e o seu comprimento, e um número inteiro positivo para usar como semente num gerador pseudo-aleatório. O output deverá ser um labirinto impresso no formato abaixo descrito.

O gerador pseudo-aleatório a utilizar é dado pela função `generateRandoms :: 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 99. Esta função será providenciada aos alunos.

Labirintos

Um labirinto é uma lista de corredores. Por outras palavras, é uma grelha ou matriz de peças, em que cada peça é uma comida, parede ou chão. Adicionalmente, uma comida pode ser grande (bónus) ou pequena.

```
type Maze = [Corridor] -- sempre horizontal
type Corridor = [Piece]
data Piece = Food FoodType | Wall | Empty
data FoodType = Big | Little
```

Apresenta-se abaixo um exemplo de um labirinto com 4 corredores, cada um com 8 peças:

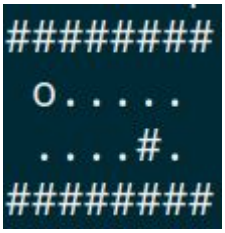
```
maze = [
    [Wall, Wall, Wall, Wall, Wall, Wall, Wall, Wall],
    [Empty, Food Big, Food Little, Food Little, Food Little, Food
Little, Food Little, Empty],
    [Empty, Food Little, Food Little, Food Little, Food Little,
Wall, Food Little, Empty],
    [Wall, Wall, Wall, Wall, Wall, Wall, Wall, Wall]
```

```
        ]
```

De modo a visualizar um labirinto sob a forma de caracteres ASCII impressos no terminal, podemos considerar uma instância de Show definida da seguinte forma:

```
instance Show Piece where
     show (Food Big) = "o"
     show (Food Little) = "."
     show (Wall) = "#"
     show (Empty) = " "
```

Visualmente, o labirinto `maze` corresponde à seguinte figura:



Nota: O labirinto `maze` é um exemplo muito simples e é apenas ilustrativo. Não constitui um labirinto completamente válido.

Labirintos válidos

Para efeitos deste trabalho prático, só serão considerados labirintos válidos. Um labirinto é válido quando:

1. O labirinto é rodeado por peças Wall (exceto túnel);
2. A faixa mais central forma um túnel;
3. Todos os corredores têm o mesmo comprimento;
4. A “casa dos fantasmas” está localizada no centro do labirinto.

Labirintos pseudo-aleatórios

As peças não fixas (i.e. as peças que não sejam os limites do labirinto) devem ser determinadas a partir de uma sequência de números pseudo-aleatórios entre 0 e 99. Num labirinto de dimensão 8x4, o número de células cujo conteúdo tem que ser gerado é 12 (não é 8 x 4 = 32 porque se assume que o labirinto é rodeado por peças `Wall` ou `Empty`), pelo que para determinar o respectivo conteúdo iremos precisar de gerar uma sequência de 12 números aleatórios (1 para cada peça). Se a semente inicial for `1111111111`, essa sequência pode ser gerada da seguinte forma:

```
Tarefa1> generateRandoms 12 1111111111
[3,22,14,12,35,17,8,1,57,18,74,9]
```

Cada peça é determinada a partir do respectivo número aleatório. Estes números aleatórios são distribuídos pelas células sequencialmente, linha por linha. Por exemplo, num labirinto de dimensão 8x4, a sequência anterior iria ser distribuída pelas células (cujo conteúdo não está fixo à partida) da seguinte forma:

-	**-**	**-**	**-**	**-**	**-**	**-**	**-**
-	**3**	**22**	**14**	**12**	**35**	**17**	**-**
-	**8**	**1**	**57**	**18**	**74**	**9**	**-**
-	**-**	**-**	**-**	**-**	**-**	**-**	**-**

A forma como se determina a peça a partir de um número aleatório ***n*** é a seguinte:

Valores (n)	**Peça**
3	Food Big
[0, 70[	Food Little
[70, 99]	Wall

Estruturas do labirinto

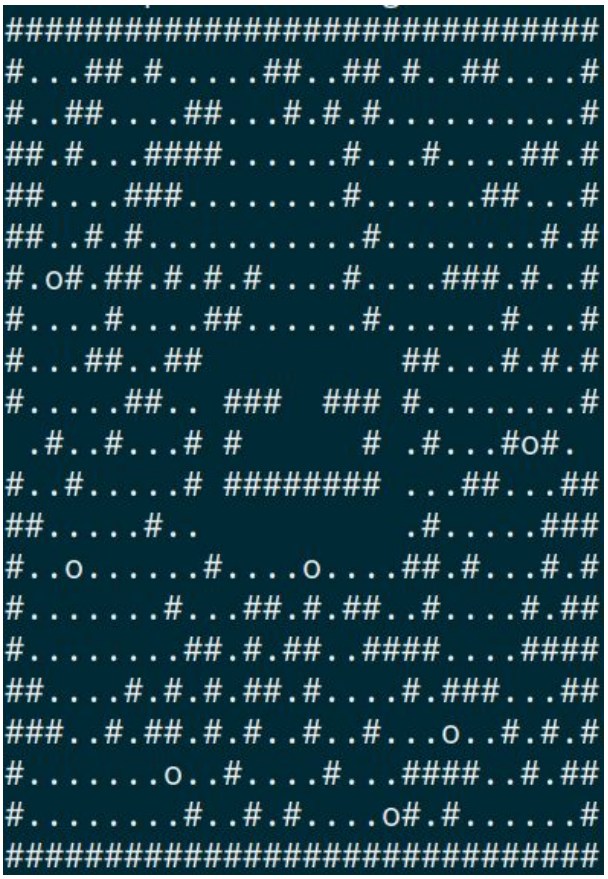
Túnel

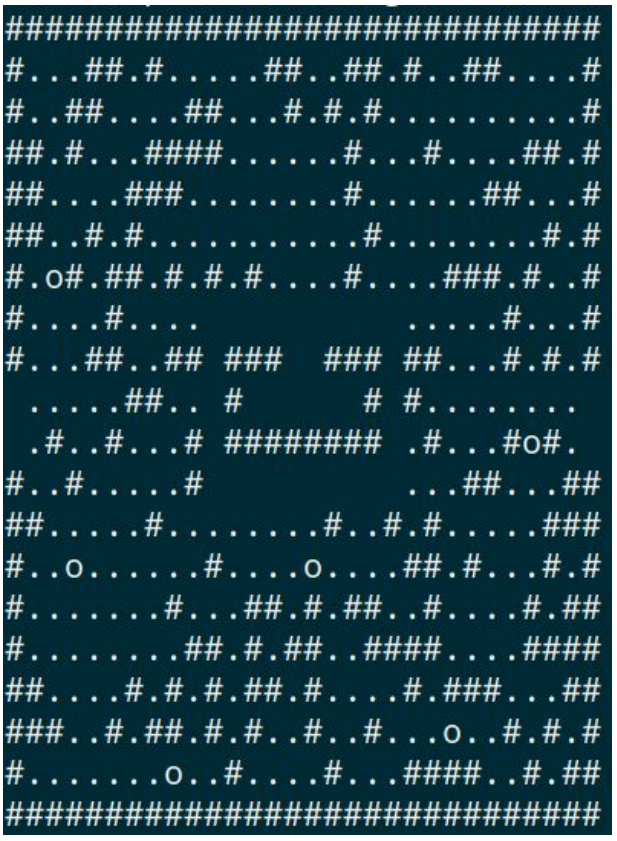
Entende-se por túnel, a parte dos limites do labirinto que não é formada por peças `Wall`. Isto é, um túnel trata-se de uma faixa horizontal cujos limites são peças `Empty`.
Quando um jogador efetua uma jogada em que se movimenta para lá de um destes limites de peça `Empty`, o jogo é responsável por transportar o jogador para o limite oposto, i.e., o jogador deve surgir através da peça `Empty` do lado oposto do labirinto.
Em tarefas seguintes do trabalho prático teremos a preocupação de efetuar jogadas e fazer com que o jogo reaja a essas jogadas. Para já, esta primeira tarefa é estática, isto é, não implica movimento por parte do Pacman nem dos fantasmas.
No entanto, é nesta primeira tarefa que o túnel deve ser colocado no labirinto.
Um dos requisitos acima indicados para um labirinto ser considerado válido é o de que a faixa mais central deve formar uma destas estruturas a que chamamos de túnel.

Para gerar o túnel, a altura do labirinto deve ser tida em consideração. Se esta for:

- ímpar, o túnel será formado por 1 corredor;
- par, o túnel será formado por 2 corredores.

Desta forma, garantimos que o túnel se encontra exatamente a meio do labirinto.



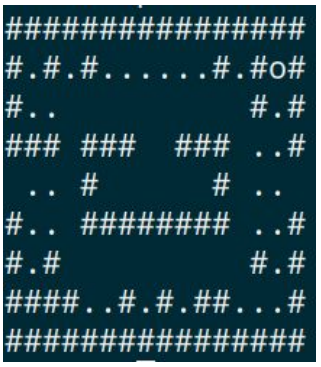


A imagem à esquerda mostra um labirinto com dimensão 30 x 21 e, como tal, o túnel tem altura 1.
Por sua vez, na imagem à direita, o labirinto tem dimensão 30 x 20 e, portanto, o túnel tem altura 2.

Casa dos Fantasmas

A casa dos fantasmas é outro dos elementos que deve ser colocado no labirinto durante esta tarefa e é um dos requisitos para que este seja válido.
Esta estrutura deve obedecer aos seguintes critérios:

- altura é sempre 3;
- comprimento do labirinto ímpar -> casa tem comprimento 9;
- comprimento do labirinto par -> casa tem comprimento 8;
- a área à volta da casa dos fantasmas deve ser gerada ~~OU com `Food` `Little` OU com `Empty` (apenas um destes tipos de peça deve rodear a casa)~~ com peças `Empty`.



À esquerda, está um labirinto com dimensões 15 x 10. Em função do comprimento ser 15, a casa tem comprimento 9 e a entrada é colocada na parte central superior e tem comprimento 3.
O labirinto da imagem à direita tem dimensões 16 x 9. Como tal, o comprimento da casa é 8. Como consequência, a entrada é menor que a anterior e tem comprimento 2.
Se cumprirmos com estes requisitos, garantimos que a casa se encontra centrada horizontalmente para qualquer labirinto gerado.
Como consequência desta estrutura ter altura fixa, esta nunca estará centrada verticalmente para labirintos com altura par. A imagem à esquerda é um exemplo de tal situação. Nestes casos, a casa deve ser colocada a *X* peças de distância da parede inferior e a *X-1* peças de distância da parede superior.

Funções a implementar

O objetivo desta tarefa é definir a função

```
generateMaze :: Int -> Int -> Int -> Maze
```

que gera um labirinto pseudo-aleatório com as dimensões desejadas. Por exemplo, deverá ser verdade que `generateMaze 8 4 1111111111 == maze`, usando os parâmetros `8 4 1111111111` e o labirinto `maze` definidos acima.
Os labirintos gerados por esta função devem ser garantidamente válidos e com dimensões mínimas de 15 x 10.

Tarefa 2 - Efetuar jogadas

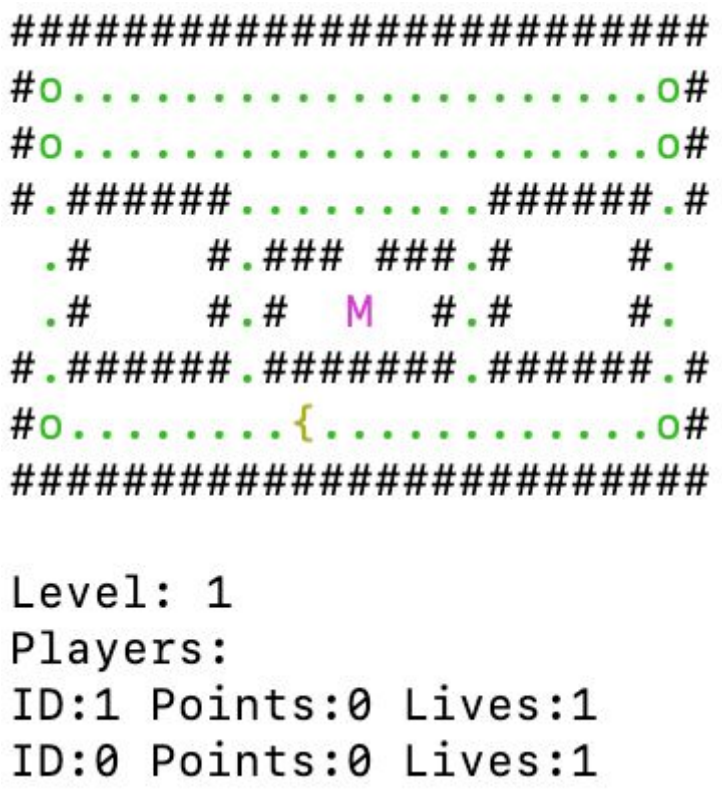
O objetivo desta tarefa é, dada uma descrição do estado do jogo e uma jogada de um dos jogadores, determinar o efeito dessa jogada no estado do jogo. Para esta tarefa, apenas se exige a implementação de realização de jogadas para jogadores do tipo Pacman. A jogada consiste em mover um jogador (identificado pelo seu identificador único ID representado pelo Int no tipo de dados **Play**) numa determinada direção e atualizar o efeito dessa jogada no labirinto atual.

Jogadas

```
data Play = Move Int Orientation
data Orientation = L | R | U | D
```

Estado do jogo

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



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

```
data State = State
     {
          maze :: Maze
      ,   playersState :: [Player]
      ,   level :: Int
     }
```

Um estado de exemplo pode ser representado pelo valor:

```
e = State m ps l
```

O estado `e` contém 2 jogadores (podendo conter vários jogadores, cada um com o respectivo identificador (Id) único):

```
js = [jogador1,jogador2]
jogador1=(Pacman (PacState (1,(7,10),1,R,0,1) 0 Open Normal))
jogador2=(Ghost  (GhoState (0,(5,12),1,R,0,1) Alive ))
```

Jogadores

Um jogador pode ser um fantasma ou Pacman, sendo cada um destes composto pela informação relativa ao seu estado. No caso de ser Pacman, este contém o tipo de dados **PacState**, que contém informação relativa ao estado do jogador (**PlayerState**, que todos tipos Player contêm), o tempo restante para estar em modo **Mega**, o estado da sua boca (aberta ou fechada, para efeitos de animação) e o seu modo (Mega, Normal ou Dying). No caso de ser um fantasma (**Ghost**), tem apenas a informação relativa ao seu estado **PlayerState** e respetivo modo **GhostMode** (**Dead** ou **Alive**).

O estado do jogador **PlayerState** consiste num tuplo cujas posições representam, respetivamente, a seguinte informação:

- O identificador único;
- As coordenadas cartesianas (x,y) do jogador no Labirinto;
- A velocidade;
- A orientação;
- A pontuação;
- O número de vidas restantes.

```
data Piece   =  Food FoodType | PacPlayer Player| Empty | Wall
data Player  =  Pacman PacState | Ghost GhoState
data PacState= PacState
    {
        pacState :: PlayerState
    ,   timeMega :: Double
    ,   openClosed :: Mouth
    ,   pacmanMode :: PacMode

    }

data GhoState= GhoState
    {
        ghostState :: PlayerState
```

```
    ,   ghostMode :: GhostMode
    }

type Coords = (Int,Int)
type PlayerState = (Int, Coords, Double, Orientation, Int, Int)
data Mouth = Open | Closed
data PacMode = Dying | Mega | Normal deriving Show
data GhostMode = Dead | Alive deriving Show
```

O estado do jogador indica o modo do jogador, i.e se ele se encontra vivo, morto ou Mega (no caso do Pacman). Um Pacman poderá então estar em modo Mega, durante um intervalo de tempo definido por uma unidade temporal conveniente especificada por **timeMega**, em cujo caso este jogador poderá engolir fantasmas. Um jogador pode deslocar-se para qualquer posição que não contenha parede, desde que não esteja morto. Nesse caso a jogada não terá nenhum efeito no jogo.

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

- Quando um jogador se movimenta numa direção que não a sua direção atual, este jogador deve permanecer na mesma posição, mudando apenas a sua orientação.
- Quando se movimenta na direção que possui, este deve transitar de posição, caso lhe seja possível.

Quando existe uma mudança de posição por parte do jogador induzida pela jogada efetuada, este pode efetuar uma de várias ações. O tipo de ação depende da peça contida na nova posição:

- Quando um jogador transita para uma posição vazia, nada acontece;
- Quando um jogador transita para uma posição com comida pequena, este deverá atualizar a sua pontuação com +1 ponto;
- Quando um jogador transita para uma posição com comida grande, este deverá atualizar a sua pontuação +5 pontos e mudar o seu estado para Mega. Os fantasmas existentes no labirinto devem mudar o seu estado para Dead e diminuir a velocidade para metade.
- Quando um jogador transita para uma posição com um fantasma vivo (**Alive**), e ainda tem vidas restantes, deve perder uma vida. Caso o seu número de vidas seja == 0 aquando do embate com o fantasma, o jogador deve atualizar o seu estado para Dying.
- Quando um jogador transita para uma posição com um fantasma morto (**Dead**), o jogador deve “comer” o fantasma. O resultado desta ação é um incremento de +10 pontos no jogador e o desaparecimento do fantasma do labirinto, voltando a reaparecer em modo **Alive** na casa dos fantasmas.
- Quando um jogador entrar num túnel (i.e um espaço vazio junto ao limite do labirinto) e transita para fora do labirinto, deve ser transportado para o lado oposto do labirinto.

Qualquer outra situação que não uma das acima contempladas deve resultar na permanência do jogador na posição original de onde pretendeu transitar.

Note que os efeitos no jogador da mudança da sua velocidade e da sua transição para estado Mega só alteram o estado do jogador. O efeito visual dessas alterações, bem como de fazer os jogadores andar sozinhos pelo labirinto sem fazer jogadas, são da responsabilidade da tarefa de passagem de tempo e não são manifestados na jogada – isto será tratado na segunda fase do trabalho.

Funções a implementar

O objectivo desta tarefa é definir a função

```
play :: Play -> State -> State
```

que efetua uma jogada para um determinado jogador sobre um determinado estado, retornando o novo estado após a jogada ser efetuada.

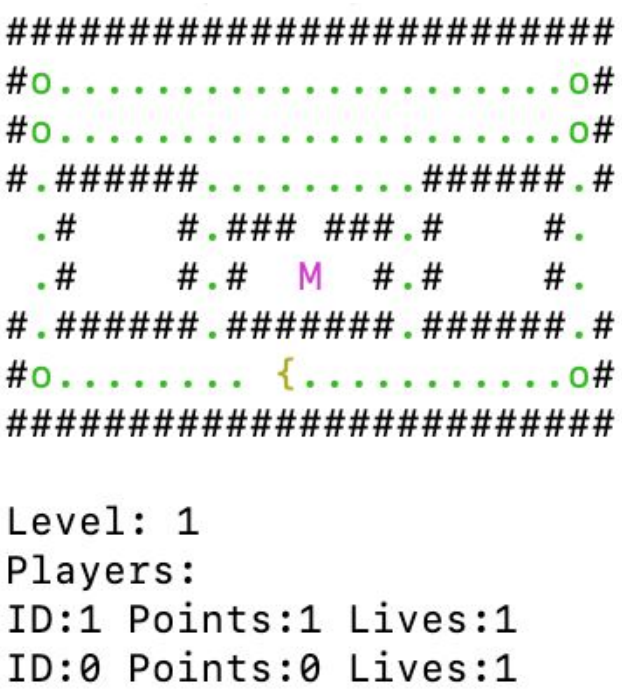
Por exemplo, dado o estado `e` do exemplo anterior, e a jogada `Move 1 R` para o jogador `1`, a função `play (Move 1 R) e` deveria retornar o estado `e'`

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

onde `js'` é definido como

```
js' =  [jogador1,jogador2]
jogador1=(Pacman (PacState (1,(7,11),1,R,1,1) 0 Open Normal))
jogador2=(Ghost  (GhoState (0,(5,12),1,R,0,1) Alive ))
```

Este novo estado pode ser visualizado na seguinte imagem.



Tarefa 3 - Compactar labirinto

O objectivo desta tarefa é, dado um labirinto válido, convertê-lo numa sequência de instruções de modo a recriá-lo num formato mais compacto para leitura.

Instruções

Uma instrução dada a um interpretador consiste em: uma lista de tuplos cujo primeiro valor é um inteiro e o segundo é uma peça; o somatório de todos os inteiros deverá ser igual à largura do labirinto; o comprimento da lista de instruções deve ser igual à altura do labirinto. A representação de instruções é dada pelos seguintes tipos de dados:

```
type Instructions = [Instruction]
data Instruction = Instruct [(Int, Piece)]
                 | Repeat Int
```

Uma instrução válida não pode conter um número maior ou menor de peças do que a largura do labirinto. Por exemplo, a instrução `(15, Wall)` para um labirinto de largura igual a 10 é considerada inválida.

Padrões

Uma forma simples de resolver esta tarefa é converter cada peça numa instrução (de forma muito similar à Tarefa 1) do tipo `Empty`, `Wall` ou `Food`. Por exemplo, a linha `r` de um labirinto `l` acima definido pode ser desconstruído na seguinte sequência de instruções:

```
[(1, Wall),(1, Wall),(1, Wall),(1, Empty),(1, Food),(1, Food),(1, Empty),(1, Wall)]
```

Embora correcta, esta solução não é óptima, no sentido em que utiliza sempre tantas instruções quantas dimensões do labirinto. Uma forma mais compacta e, portanto, económica no número de instruções, é identificando padrões que se repetem no labirinto. 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 exemplo anterior é possível encontrar dois casos de repetições horizontais: 1) 3 células seguidas da forma `(1, Wall)` e 2) 2 células da forma `(1, Food)`. Estes padrões podem ser melhor representados mediante a instrução `(3, Wall)` e `(2, Food)`, respectivamente:

```
[(3, Wall),(1, Empty),(2, Food),(1, Empty),(1, Wall)]
```

Padrões verticais

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

```
[
  [(5, Wall)],
 [(1, Wall), (2, Food Little), (1, Food Big), (1, Wall)],
  [(1, Wall), (3, Food Little), (1, Wall)],
  [(1, Wall), (3, Food Little), (1, Wall)],
 [(1, Wall), (1, Food Little), (1, Food Big), (1, Food Little), (1, Wall)],
  [(5, Wall)],
]
```

É possível identificar casos em que padrões se repetem entre os primeiro e último corredores, bem como entre os terceiro e quarto corredores. Neste caso é possível indicar ao interpretador que aplique ao quarto corredor o mesmo formato aplicado ao terceiro, colocando uma instrução que designa o **índice** de um elemento da lista de instruções a desenhar de novo, `Repeat 2`, atualizando devidamente o resto das instruções (neste caso particular não se reduz o número de instruções):

```
[ [(5, Wall)],
 [(1, Wall), (2, Food Little), (1, Food Big), (1, Wall)],
```

```
  [(1, Wall), (3, Food Little), (1, Wall)],
  Repeat 2,
 [(1, Wall), (1, Food Little), (1, Food Big), (1, Food Little),
(1, Wall)],
  Repeat 0,
]
```

Funções a implementar

O objectivo desta tarefa é definir a função

```
compactMaze    :: Maze -> Instructions
```

cujo objectivo é, respectivamente, codificar o labirinto para como uma sequência de instruções que, quando executadas, produzem o mesmo labirinto. No entanto, pretende-se uma implementação de uma boa função de compactação, de forma a que o número de instruções a dar aos interpretador 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 `sizeInstructions :: Instructions -> Int`.

Entrega e Avaliação

A data limite para conclusão de todas as tarefas desta primeira fase é **5 de dezembro de 2020 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 Git: 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 GIT do respectivo grupo (código, documentação, ficheiros de teste, etc.). A utilização das diferentes ferramentas abordadas no curso (como Haddock, GIT, 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 GIT 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 GIT como sistema de controle de versões.