

# Micro Machines

Enunciado da 1ª fase do projeto de LI1 2017/18

## 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 pequenos programas em Haskell que deverão responder a diferentes tarefas (apresentadas adiante).

O objetivo do projeto deste ano é implementar o clássico jogo de corridas Micro Machines. O principal objetivo do jogo é completar um percurso pré-definido no menor tempo possível. Este percurso pode ter diferentes alturas e é rodeado por lava. O jogador é penalizado sempre que sofre uma queda dentro do percurso ou quando cai na lava.



## Tarefas

**Importante:** Cada tarefa deverá ser desenvolvida num módulo Haskell independente, nomeado `Tarefan_2017li1gxxx.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.

## Tarefa 1 - Construir Mapas

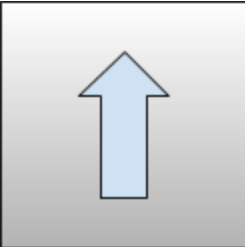
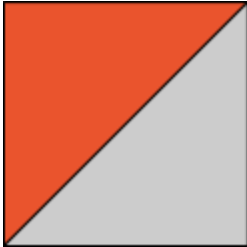
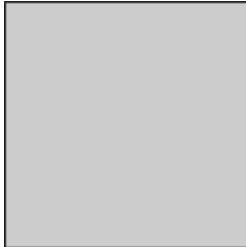
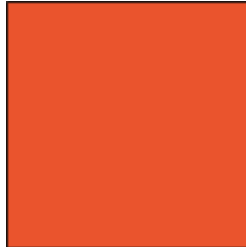
O objetivo desta tarefa é construir um mapa para o jogo seguindo um caminho previamente registado. Um caminho consiste numa lista de passos, onde cada passo é uma das seguintes opções: *avança*, *sobe*, *desce*, *curva à esquerda* ou *curva à direita*. O tipo de dados para representar um caminho e um passo é dado por:

```
type Caminho = [Passo]
data Passo = Avanca | Sobe | Desce | CurvaEsq | CurvaDir
```

Um exemplo de um caminho simples é o seguinte:

```
c :: Caminho
c = [Avanca, CurvaDir, Sobe, CurvaDir, Avanca, CurvaDir, Desce, CurvaDir]
```

Um mapa consiste numa matriz de peças (implementada com uma lista de listas), onde cada peça está a uma determinada altura e pode ser do tipo *lava*, *recta*, *rampa* ou *curva*. Peças do tipo *rampa* e *curva* têm também uma orientação associada, indicando respectivamente a inclinação ou o sentido da curva. Como indicado na imagem em baixo, uma curva com orientação a Norte, tem acessos pelo Sul e Este. Uma rampa com orientação a Norte, tem acessos pelo Norte e Sul, e uma inclinação positiva no sentido Sul-Norte (representada na imagem através de uma seta no sentido da subida).

			
Rampa Norte	Curva Norte	Recta	Lava

As peças são representadas pelos seguintes tipos:

```
type Altura      = Int
data Orientacao  = Norte | Sul | Este | Oeste
data Tipo        = Rampa Orientacao | Curva Orientacao | Recta | Lava
data Peca        = Peca Tipo Altura
```

Note que peças do tipo lava, recta e curva têm uma única altura. Assume-se que a altura atribuída a uma peça do tipo rampa é a do seu ponto *mais baixo*. Todas as peças lava têm uma *altura fixa* de 0.

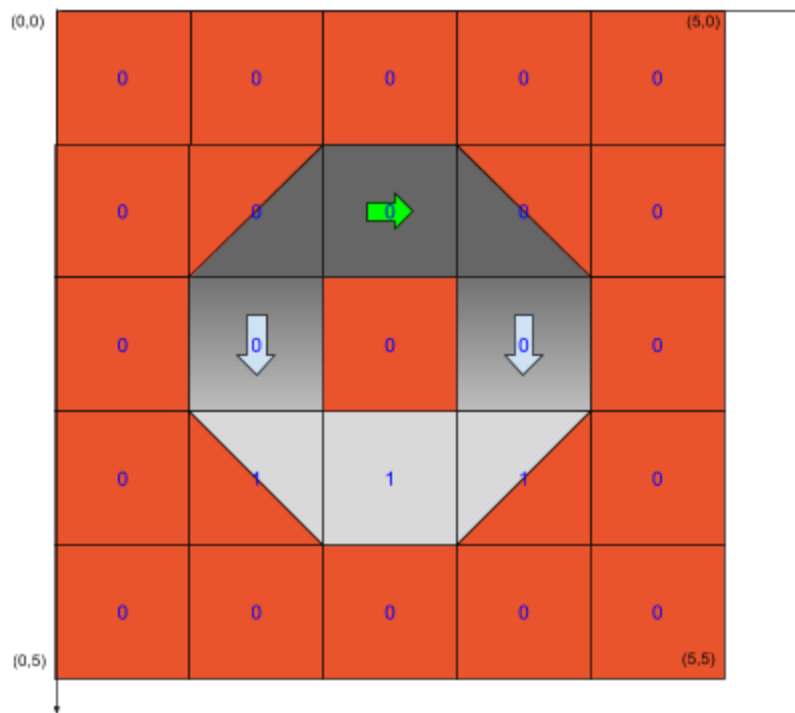
O tipo `Mapa`, definido abaixo, é composto por uma matriz de peças (representada pelo tipo `Tabuleiro`), e uma posição e orientação iniciais. Cada `Posicao` no tabuleiro é representada por um par de inteiros, correspondendo a posição (0,0) ao canto superior esquerdo.

```
type Posicao      = (Int,Int)
type Tabuleiro   = [[Peca]]
data Mapa       = Mapa (Posicao,Orientacao) Tabuleiro
```

A construção de um tabuleiro deve começar a partir da posição inicial dada pela função `partida :: Caminho -> Posicao` (disponibilizada no módulo `LI11718.hs` existente no repositório SVN de cada grupo), na direção `Este` e com a altura 0. Considere também a função `dimensao :: Caminho -> Dimensao` (também disponibilizada) que calcula a dimensão ideal para um tabuleiro correspondente a um caminho (assumindo sempre uma margem de lava ao redor da pista). Caso o caminho passe mais do que uma vez na mesma posição, deve ser considerado o resultado do passo mais recente. Posições não percorridas pelo caminho devem resultar em lava.

O tabuleiro correspondente ao caminho de exemplo `c` definido acima é representado textualmente e visualmente de seguida, considerando que `dimensao c == (5,5)` e `partida c == (2,1)` (a peça de partida é representada pela seta verde). Note que a direção da rampa no tabuleiro é relativa ao eixo do mapa, e independente do caminho que lhe deu origem.

```
t :: Tabuleiro
t =
[[Peca Lava 0,Peca Lava 0,          Peca Lava 0, Peca Lava 0,          Peca Lava 0]
,[Peca Lava 0,Peca (Curva Norte) 0,Peca Recta 0,Peca (Curva Este) 0,Peca Lava 0]
,[Peca Lava 0,Peca (Rampa Sul) 0,   Peca Lava 0, Peca (Rampa Sul) 0, Peca Lava 0]
,[Peca Lava 0,Peca (Curva Oeste) 1,Peca Recta 1,Peca (Curva Sul) 1, Peca Lava 0]
,[Peca Lava 0,Peca Lava 0,          Peca Lava 0, Peca Lava 0,          Peca Lava 0]]
```



O objectivo desta tarefa é definir a função `constroi :: Caminho -> Mapa` que constrói um mapa que corresponde ao caminho recebido. No caso do exemplo acima, deve ser verdade que `constroi c == Mapa (partida c, Este) t`.

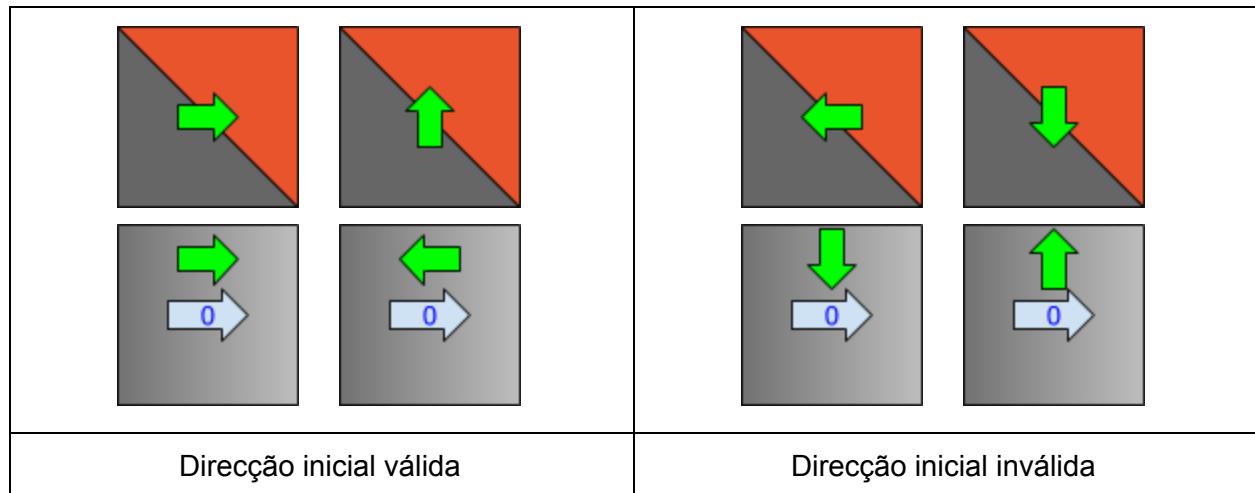
## Tarefa 2 - Validar Mapas

O objetivo desta tarefa é testar se um dado mapa é ou não *válido* de acordo com um conjunto de regras. Considere para cada mapa deve existir um *percurso* de peças de piso que permita ao carro dar voltas à pista. As seguintes regras devem então ser verificadas:

- Existe apenas um percurso e todas as peças fora desse percurso são do tipo lava;
- O percurso deve corresponder a uma trajectória, tal que começando na peça de partida com a orientação inicial volta-se a chegar à peça de partida com a orientação inicial;
- A orientação inicial tem que ser compatível com a peça de partida. Como é sugerido na imagem abaixo, considera-se que a orientação é compatível com a peça se for possível entrar na peça seguindo essa orientação (note que esta questão só é relevante para as peças do tipo curva e rampa);
- As peças do percurso só podem estar ligadas a peças do percurso com alturas compatíveis;
- Todas as peças do tipo lava estão à altura 0;
- O mapa é sempre rectangular e rodeado por lava, ou seja, a primeira e última linha, assim como a primeira e última coluna são constituídas por peças necessariamente do tipo lava.

Note que em termos de percurso, assume-se que as rectas não alteram a orientação da trajectória. Basicamente, os mapas válidos são os que contêm um (e apenas um) percurso definido pelo tipo `Caminho` apresentado na tarefa anterior, com a nuance que o percurso pode começar a uma altura, numa direcção e numa posição diferentes das predefinidas na Tarefa 1.

O objectivo desta tarefa é definir a função `valida :: Mapa -> Bool` que dado um mapa, verifica se este é ou não válido. A imagem seguinte exemplifica peças válidas e inválidas enquanto posições iniciais (setas verdes indicam a direcção e setas cinzentas a inclinação).



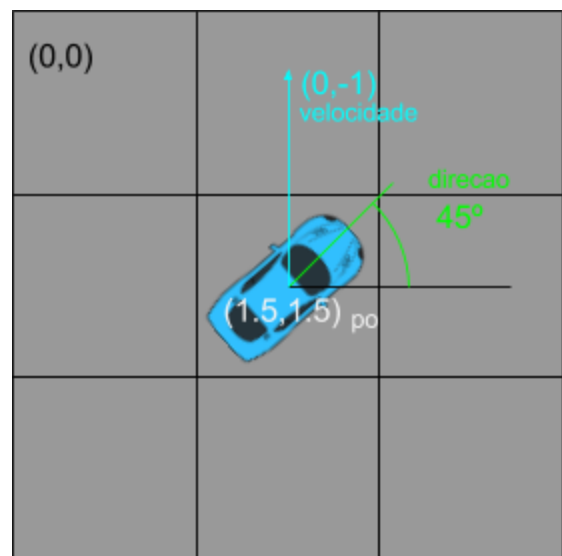
## Tarefa 3 - Movimentar o Carro

O objetivo desta tarefa é começar a implementar a mecânica do jogo, concretamente as movimentações do carro no mapa de jogo e as respetivas colisões com obstáculos. Para isso, começamos por modelar um `Carro` como contendo uma *posição*, uma *direcção* (um ângulo em graus<sup>1</sup>) e um vetor de *velocidade*:

```

type Ponto      = (Double,Double)
type Angulo     = Double
type Velocidade = (Double,Double)
type Tempo      = Double
data Carro      = Carro {
  posicao      :: Ponto,
  direcao     :: Angulo,
  velocidade  :: Velocidade }

```



<sup>1</sup> Relembre que as funções trigonométricas em Haskell assumem ângulos em radianos.

Note que o carro pode estar virado numa certa direcção mas a movimentar-se numa outra. O carro de exemplo desenhado à direita pode ser representado como:

```
carro :: Carro
carro = Carro {ponto = (1.5,1.5),direcao = 45,velocidade = (0,-1)}
```

Nesta tarefa, dado o estado atual dum carro, deve calcular o seu novo estado após um determinado período de tempo, prestando atenção às interações com o mapa que podem ocorrer durante esse período, nomeadamente:

- Se o carro transita para a lava então é “*destruído*”;
- Se o carro transita para uma posição pelo menos uma unidade mais baixa que a atual, então “cai” e é “*destruído*”;
- Se o carro transita para uma posição pelo menos uma unidade mais alta que a atual, então *colide* contra a peça, devendo ser calculado o impacto da colisão, ou seja, qual o respectivo “*ricochete*”.
- Se o carro transita entre duas posições com diferença de alturas menor a uma unidade, então movimenta-se normalmente.

Note que este processo pode envolver várias colisões (ver imagem no sistema de *feedback*). Em todos os passos do processo é assumido que o carro está sempre numa posição válida (isto é, está sempre dentro da “estrada”).

Concretamente, o objectivo desta tarefa é definir a função `movimenta :: Tabuleiro -> Tempo -> Carro -> Maybe Carro` que recebe um tabuleiro, um período de tempo e um carro, e calcula o novo estado do carro após se ter movimentado durante o período de tempo dado. As regras de colisões são as que bem conhece da física, devendo também assumir que:

- A direcção do carro não é alterada;
- O carro desloca-se a velocidade constante e nem “acelera” nem “curva”, inclusive durante colisões. Mais concretamente, o “valor absoluto” da velocidade não é alterado, mas sim a sua direcção de acordo com o ângulo de colisão;
- O resultado é do tipo `Maybe Carro` para contemplar a possibilidade de o carro ser “destruído”, devendo a função devolver `Nothing` quando o carro é “destruído” (em qualquer momento durante a sua movimentação), ou `Just` e em caso contrário, sendo `e` o novo estado do carro.

## Sistema de *Feedback*

O projecto inclui também um Sistema de *Feedback*, alojado em <http://li1.lsd.di.uminho.pt> 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* é actualizada regularmente com o conteúdo de cada 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 :: [Caminho]
testesT2 :: [Tabuleiro]
testesT3 :: [(Tabuleiro, Tempo, Carro)]
```

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

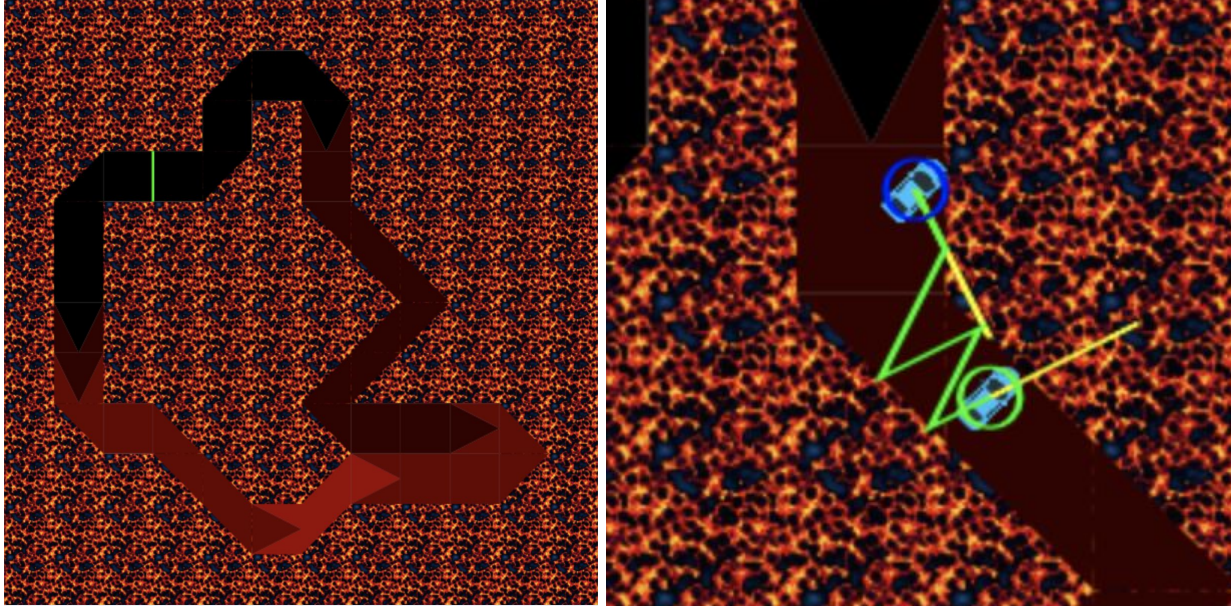
```
testesT1 = [c]
testesT2 = [t]
testesT3 = [(t, 1.6, Carro (1.5, 2.5) 45 (0, -1))]
```

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.

No caso da Tarefa 3, lembre que podem existir aproximações devido à aritmética com números de vírgula flutuante. Por esse motivo, o *feedback* dado compara a precisão da solução dos alunos em relação ao oráculo desenvolvido pelos docentes e, no caso de ambiguidades, diversas soluções são consideradas válidas.

As seguintes imagens demonstram os visualizadores de mapas e colisões, respetivamente:





## Entrega e Avaliação

A data limite para conclusão de todas as tarefas desta primeira fase é **15 de Novembro de 2017 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 nível de precisão do simulador de colisões. 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.

## Anexo - Biblioteca de tipos e funções auxiliares

```
{-|
Module      : LI11718
Description : Módulo auxiliar para LI1 17/18

Tipos de dados e funções auxiliares para a realização do projeto de LI1 em
2017/18.
-}

module LI11718 (
  -- * Tipos de dados
  -- ** Básicos
  Altura, Orientacao(..), Posicao, Ponto, Angulo, Tempo, Velocidade,
  -- ** Caminhos
  Caminho(..), Passo(..),
  -- ** Mapas
  Mapa(..), Tabuleiro, Peca(..), Tipo(..), Dimensao, Carro(..),
  -- * Funções auxiliares fornecidas
  dimensao, partida, dirInit, altInit, altLava
) where

-- | Uma sequência de passos que dá origem a um mapa.
type Caminho = [Passo]
-- | Tipos de passos que podem ocorrer num 'Caminho'.
data Passo
  -- | Segue em frente e mantém o nível
  = Avanca
  -- | Segue em frente e sobe um nível
  | Sobe
  -- | Segue em frente e desce um nível
  | Desce
  -- | Curva à esquerda e mantém o nível
  | CurvaEsq
  -- | Curva à direita e mantém o nível
  | CurvaDir
  deriving (Eq, Read, Show)

-- | Mapa de um jogo, composto por um tabuleiro, uma posição inicial e uma
altura inicial.
data Mapa = Mapa (Posicao, Orientacao) Tabuleiro
  deriving (Eq, Read, Show)

-- | O tabuleiro do mapa, representado por uma matriz de 'Peca'.
type Tabuleiro = [[Peca]]
```

```

-- | Uma peça num 'Tabuleiro'. A altura atribuída é sempre a do ponto mais
/baixo/ da peça.
data Peca = Peca Tipo Altura
  deriving (Eq,Read,Show)

-- | Tipos de peças contidos num 'Tabuleiro'.
data Tipo = Rampa Orientacao | Curva Orientacao | Recta | Lava
  deriving (Eq,Read,Show)

-- | Posições num 'Tabuleiro'.
type Posicao = (Int,Int)
-- | Dimensões de um 'Tabuleiro'.
type Dimensao = (Int,Int)
-- | Altura de uma peça num 'Tabuleiro'.
type Altura = Int

-- | Orientações no mapa.
data Orientacao = Norte | Este | Sul | Oeste
  deriving (Eq,Read,Show,Enum)
-- | Ponto no mapa.
type Ponto = (Double,Double)
-- | Ângulo em graus.
type Angulo = Double
-- | Períodos de tempo.
type Tempo = Double
-- | Vectores de velocidade.
type Velocidade = Ponto

{- |
  O estado de um carro dentro de um 'Mapa'.
  A direção da velocidade (movimento) /não/ é necessariamente a mesma da
  direção do carro.
-}
data Carro = Carro
  { posicao      :: Ponto      -- ^ a posição atual do carro
  , direcao     :: Angulo     -- ^ a direção atual do carro
  , velocidade  :: Velocidade -- ^ a velocidade atual do carro
  }
  deriving (Eq,Read,Show)

{- | Direção inicial na construção de um caminho.

prop> dirInit == Este
-}
dirInit :: Orientacao
dirInit = Este

```

```

{- | Altura inicial na construção de um caminho.

prop> altInit == 0
-}
altInit :: Altura
altInit = 0

{- | Altura da lava.

prop> altLava == 0
-}
altLava :: Altura
altLava = 0

{- | Dado um caminho, calcula a dimensão do tabuleiro correspondente.

>>> dimensao [CurvaDir, CurvaDir, CurvaDir, CurvaDir]
(4,4)
-}
dimensao :: Caminho -> Dimensao
dimensao c = (2+m'+abs m+1, 2+n'+abs n+1)
  where ((m,m'), (n,n')) = bB ((0,0), (0,0)) (0,0) dirInit c

{- |
Dado um caminho, calcula a sua posição inicial no tabuleiro correspondente.

>>> partida [CurvaDir, CurvaDir, CurvaDir, CurvaDir]
(2,1)
-}
partida :: Caminho -> Posicao
partida c = (1+abs m, 1+abs n)
  where ((m,m'), (n,n')) = bB ((0,0), (0,0)) (0,0) dirInit c

bB m i _ [] = up m i
bB m i d (s:c) = bB (up m i) (mx i d') d' c
  where d' = toEnum $ ((fromEnum d) + k) `mod` 4
        k | s == CurvaDir = 1
          | s == CurvaEsq = (-1)
          | otherwise = 0
up ((m,m'), (n,n')) (i,j) = ((min m i, max m' i), (min n j, max n' j))
mx (x,y) d = (x+round (sin a), y-round (cos a))
  where a = ((toEnum.fromEnum) d)*(pi/2)

```