Micro Machines

Enunciado da 2ª fase do projeto de LI1 2017/18

Introdução

Neste enunciado apresentam-se as tarefas referentes à segunda 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 `Tarefa`***`n`***`_2017li1g`***`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. Os módulos auxiliares adicionados ao SVN (ficheiros `LI11718.hs`, atualizado para a segunda fase, e `Mapas.hs`) devem também ficar inalterados.

Tarefa 4 - Atualizar Estado

O objetivo desta tarefa é **atualizar** o **estado do jogo** dadas as **ações** efectuadas por um jogador num período de tempo. Para isso é necessário representar:

- `Jogo` - o estado interno do jogo (que deverá ser atualizado em cada instante);
- `Ação` - algo que vai indicar, por exemplo, se o carro está a acelerar, travar, ou curvar.

Estado do Jogo

Relembre o tipo `Mapa` e o tipo `Carro` introduzidos na primeira fase do projeto. O estado do jogo em cada momento é definido por um *mapa* e as *propriedades* desse mapa (que devem manter-se constantes no decorrer do jogo), e, para cada um dos jogadores, o estado do *carro*, a quantidade de *"nitro" disponível* e o *histórico* de todas as posições visitadas. Estas 3 listas devem ter a mesma dimensão, correspondente ao número de jogadores.

```
data Jogo = Jogo
  { mapa        :: Mapa
  , pista       :: Propriedades
  , carros      :: [Carro]
  , nitros      :: [Tempo]
  , historico   :: [[Posicao]]
  }
```

```
data Propriedades = Propriedades
  { k_atrito    :: Double
  , k_pneus     :: Double
  , k_acel      :: Double
  , k_peso      :: Double
  , k_nitro     :: Double
  , k_roda      :: Double
  }
```

O mapa representa o percurso do jogo, e as suas propriedades as constantes físicas que afetam o movimento do jogo, e que deverão ser tidas em consideração para realizar esta tarefa. Estas propriedades variam de percurso para percurso, e são definidas pelo tipo de dados `Propriedades`.

Estas constantes representam o atrito do piso (`k_atrito`), que faz o carro abrandar naturalmente, o atrito dos pneus (`k_pneus`), que faz o carro abrandar quando o movimento é ortogonal às rodas do carro, a intensidade da aceleração do carro (`k_acel`) e do "nitro" (`k_nitro`), o efeito da gravidade nas rampas (`k_peso`) e a sensibilidade do guiador (`k_roda`), que define quão rápido o carro roda. As propriedades `p = Propriedades 2 3 4 2 15 180` representam um exemplo de propriedades sensatas para um percurso dito normal.

Uma funcionalidade que o jogo deverá implementar é a possibilidade de ativar "nitros" sobre si mesmo ou sobre outro jogador. Cada jogador tem inicialmente disponível uma quantidade limitada de "nitro", que se gasta de cada vez que é ativado pelo jogador, e que está guardada no estado do jogo (`nitros`).

Ações do jogador

As ações do jogador, que têm efeito num dado período de tempo, definem se o carro do jogador está a *acelerar, travar, curvar para a direita* ou *esquerda* ou com o *"nitro" ativado* durante todo esse período de tempo. Estes comandos são codificados pelo seguinte tipo, onde cada Booleano representa se as primeiras quatro ações estão ativas, e o quinto o possível alvo de um "nitro":

```
data Acao = Acao
  { acelerar :: Bool
  , travar   :: Bool
  , esquerda :: Bool
  , direita  :: Bool
  , nitro    :: Maybe Int
  }
```

Note que podem ser dadas várias ações ao mesmo tempo, embora os efeitos de algumas delas se anulem (`acelerar`/`travar` e `esquerda`/`direita`).

Como atualizar o estado

O estado do jogo deve ser atualizado de acordo com o impacto das ações de um dado jogador *j* no dado período de tempo, nomeadamente:

- **Atualizar o vetor velocidade** do carro *j*. Este novo vetor é calculado a partir da soma de todas as forças envolvidas (dadas como vetores), num dado período de tempo. Estas forças são divididas pelas seguintes componentes:
  - **Força de atrito:** obtida como uma percentagem da velocidade inicial (dada por `k_atrito`), com sentido oposto à velocidade;
  - **Força de aceleração (ou travagem):** com norma dada por `k_acel` e com a direção e sentido (ou oposto) do carro;
  - **Força da gravidade** (usada apenas em rampas): de acordo com a constante `k_peso`, na direção do declive;
  - **Força dos pneus:** vetor perpendicular à direção do carro, no sentido oposto à velocidade, com norma dada por
    $$seno(angulo\_da\_direcao\_à\_velocidade\_inicial) \times k\_pneus$$ [1];

- **Atualizar a direção** do carro *j* caso esteja a rodar para a `esquerda` ou `direita` de acordo com a constante de rotação `k_roda`;
- **Atualizar a quantidade de "nitro"** disponível no carro *j* caso o `nitro` esteja ativo diminuindo a quantidade de `Tempo` decorrida, e nesse caso atualizar também o vetor de

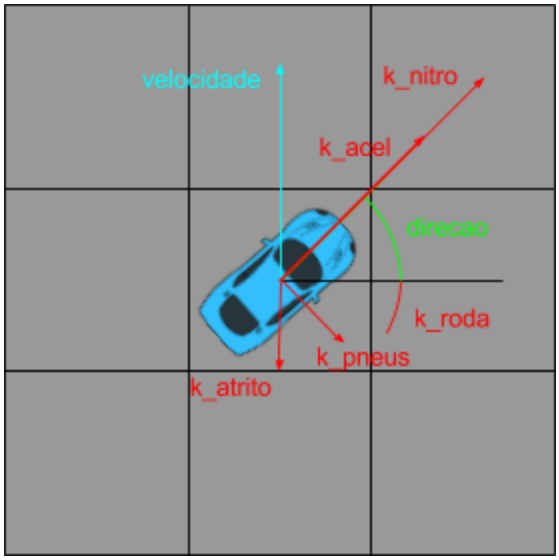
[1] Isto fará com que a norma seja 0 quando a direção do carro for paralela à direção da velocidade, e máxima (`k_pneus`) quando estas sejam ortogonais.

velocidade do carro alvo (não necessariamente o carro *j*) de acordo com a constante `k_nitro` na direção atual do carro alvo. Caso a quantidade de nitro disponível em `nitros` seja menor que o `Tempo` decorrido, o vetor deve ficar apenas ativo durante o tempo disponível (e a quantidade disponível passar a zero);

- **Atualizar o histórico** de posições percorridas pelo carro *j* adicionando-lhe a `Posicao` atual caso tenha mudado de posição desde a última atualização.

Todas estas atualizações devem ser independentes entre si e aplicadas sobre o estado do carro recebido pela função (por exemplo, os vetores de velocidade consideram a direção de entrada e não a direção atualizada, e os vetores de atrito consideram a velocidade de entrada e não a atualizada pela aceleração). Note que a posição do carro, cuja atualização é já tratada pela Tarefa 3, deve ser mantida **inalterada** nesta fase.

Como exemplo, a imagem em baixo representa, a vermelho, os elementos relevantes para atualizar o estado de um carro quando este está numa peça recta, a `acelerar`, a rodar para a `direita` e sob o efeito de `nitro`, e fora de uma rampa. Assumindo que este é o carro **número 1**, a ação que produz estes vetores pode ser codificada pela ação `a` definida também em baixo.




```
a :: Acao
a = Acao
  { acelerar = True
  , travar   = False
  , esquerda = False
  , direita  = True
  , nitro    = Just 1 }
```

Função principal da tarefa

Defina a função `atualiza :: Tempo -> Jogo -> Int -> Acao -> Jogo` que, dado um período de tempo, o estado atual do jogo, o identificador de um jogador, e a ação efetuada por esse jogador, atualiza o estado do jogo.

Tarefa 5 - Implementação do Jogo em `Gloss`

O objectivo desta tarefa é implementar o jogo completo usando a biblioteca `Gloss`. Um breve tutorial de introdução a esta biblioteca do Haskell pode ser encontrado na plataforma *e-learning*

da disciplina. Como ponto de partida deve começar por implementar uma versão com uma visualização gráfica simples e que use sempre um caminho fixo para gerar o mapa inicial. Apesar de dever ser construída inicialmente sobre as tarefas anteriores, esta tarefa trata-se no entanto acima de tudo de uma "tarefa aberta", onde se estimula que os alunos explorem diferentes possibilidades extra para melhorar o aspecto final e jogabilidade do jogo. Sugestões de extras incluem, por exemplo:

- Gráficos visualmente apelativos (como, por exemplo, 2.5D);
- Suportar diferentes câmaras ou perspectivas;
- Menus de início e fim do jogo;
- Mostrar informação sobre o estado do jogo;
- Mostrar o tempo e o número de voltas dadas ao percurso;
- Permitir jogar diferentes mapas e/ou carregar mapas definidos pelo utilizador (combinando as Tarefas 1 e 2);
- Permitir jogar contra outros jogadores e contra *bots* (da Tarefa 6);
- Suportar diferentes percursos com diferentes propriedades;
- Suportar novas funcionalidades dos carros ou das pistas.

No cerne desta tarefa estão as funções da Tarefa 4 (`atualiza`), que atualiza o estado dos carros de acordo com as decisões dos jogadores, e da Tarefa 3 (`movimenta`), que atualiza a posição do carro de acordo com esse estado. Note que apesar do tipo interno do estado do `Jogo` ser usado por estas funções, é provável que necessite de criar um novo tipo que contenha informação adicional relevante para a execução do jogo (denominado `Estado` no guião `Gloss` da disciplina). O tipo `Jogo` deve no entanto permanecer inalterado. Dadas estas duas funções, que atuam num dado período de `Tempo`, um esqueleto que atualiza o estado do jogo pode ser definido como:

```
atualizaMovimenta :: Tempo -> Jogo -> [Acao] -> Jogo
atualizaMovimenta t jogo a = novoJogo
  where jogoAct   = ---> aplica a atualiza a todos os jogadores
                    --   (a partir de "jogo")
        carrosAct = ---> aplica a movimenta a todos os carros
                    --   (usando o mapa e carros da "jogoAct")
        novoJogo =  ---> atualiza os carros em "jogoAct" com o
                    --   resultado de "carrosAct"
                    --   (usando um critério à escolha sobre onde
                    --   posicionar carros que são destruídos)
```

Tarefa 6 - Implementar uma Estratégia de Corrida

O objectivo desta tarefa é implementar um *bot* que jogue Micro Machines automaticamente. A estratégia de jogo a implementar fica ao critério de cada grupo, sendo que a avaliação automática será efectuada colocando o *bot* implementado a combater com diferentes *bots* de variados graus de "inteligência".

Em cada instante, o *bot* tem apenas conhecimento da duração da ação que será tomada, do estado atual do jogo, e pode tomar uma única decisão usando o tipo `Acao` definido em cima. Para definir a estratégia, devem assumir que:

- O objetivo do jogo é **dar uma volta à pista** à frente dos adversários, começando e acabando na posição de partida;
- O jogo **acaba passado 60s** independentemente de algum carro ter conseguido dar a volta. Nesse caso o vencedor será o carro mais avançado no percurso;
- Os **mapas são válidos** de acordo com a Tarefa 2, e a física do jogo é a mesma das Tarefas 3 e 4;
- Cada jogador começa com **5s de "nitro"**;
- Além das mortes naturais, o carro é destruído se passar para uma posição que esteja mais de 4 peças à frente do percurso ideal (i.e., **atalhos são permitidos até 4 peças**);
- Quando o carro é destruído, volta ao **centro da última posição** visitada com velocidade zero e fica **imobilizado durante 1.5s**;
- O percurso será selecionado a partir de um **conjunto fixo de caminhos e propriedades**, que se encontram definidos no módulo auxiliar `Mapas`, disponível no SVN de cada grupo.

Defina a função `bot :: Tempo -> Jogo -> Int -> Acao` que dada a duração da ação, o estado do jogo e o identificador do jogador, devolve a ação a realizar pelo *bot*.

Relatório

Nesta fase deve também ser escrito e submetido um relatório sobre o desenvolvimento do projecto. Este relatório deverá ser escrito em LaTeX, uma ferramenta que será apresentada e explorada nas aulas práticas. Será disponibilizado um template para esse relatório na plataforma de *e-learning* da disciplina.

Sistema de *Feedback*

O projecto inclui um Sistema de *Feedback*, também 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.

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

Feedback da Tarefa 4

Para receber *feedback* sobre a Tarefa 4 e qualidade dos testes, deve ser declarada no ficheiro correspondente à tarefa as seguinte lista de testes:

```
testesT4 :: [(Tempo,Jogo,Acao)]
testesT4 = [...]
```

Os testes definidos por cada grupo podem ser apresentados no *"visualizador da atualização do estado"* do Sistema de *Feedback*. Note que uma maior quantidade e diversidade de testes garante um melhor *feedback* sobre a correção das funções e ajudará a melhorar o código desenvolvido. Para facilitar a escrita dos testes, aconselha-se que os estados do `Jogo` sejam definidos como variantes de um estado predefinido.

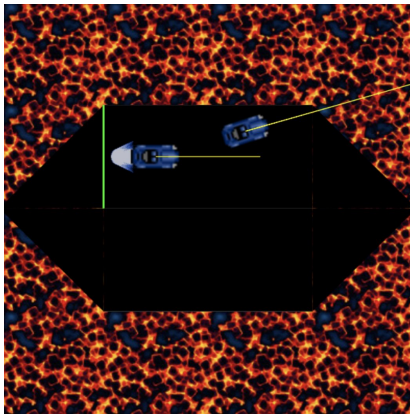
Feedback da Tarefa 5

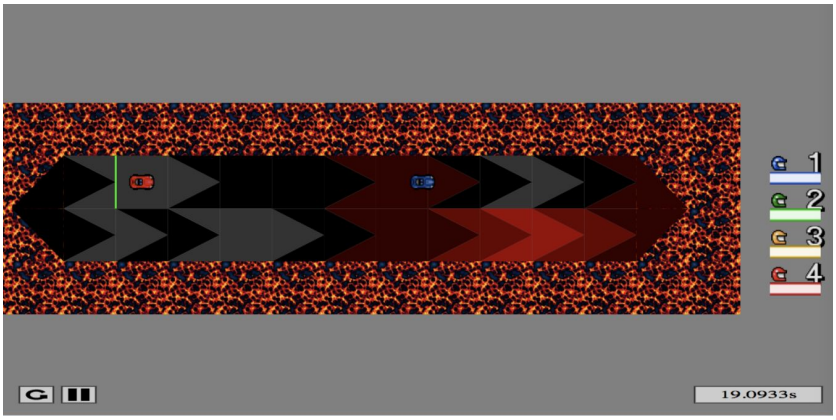
A Tarefa 5, dada a sua natureza aberta, não é contemplada pelo Sistema de *Feedback*.

Feedback da Tarefa 6

No caso da Tarefa 6, o *feedback* é apresentado na forma de um **"torneio"**, em que os vários *bots* dos alunos são postos a jogar uns contra os outros. Os alunos deverão ter atenção ao tempo de execução das soluções visto que *timeouts* resultam no carro não executar nenhuma ação. Esta informação disponibilizada pelo Sistema de *Feedback* é meramente indicativa e não será utilizada para a avaliação final.

As seguintes imagens demonstram os "visualizadores da atualização do estado" e do "simulador de *bots*":





Entrega e Avaliação

A data limite para conclusão de todas as tarefas desta primeira fase é **31 de Dezembro 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, relatório, 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 4	15%
Avaliação qualitativa da Tarefa 5	25%
Avaliação automática e qualitativa da Tarefa 6	25%
Qualidade do código	10%
Utilização do SVN, testes e documentação do código	10%
Relatório e utilização do LaTeX	15%

A avaliação automática será feita através de um conjunto de testes que **não** serão revelados aos grupos. A avaliação automática da Tarefa 6 consiste em correr os *bots* dos alunos contra *bots* de “inteligência” variada. Os alunos deverão ter atenção ao tempo de execução das soluções visto que *timeouts* resultam na derrota nesse percurso. 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.