Bomberman

1ª fase do projeto de LI1 2016/17

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 o clássico jogo Bomberman em *battle mode*. A ideia geral do jogo é colocar bombas estrategicamente de forma a matar inimigos e destruir obstáculos no mapa. A explosão de uma bomba pode desencadear explosões de outras bombas, destruir obstáculos, matar inimigos ou matar o próprio jogador.

Tarefas

Tarefa 1 - Gerar mapas

O objectivo desta tarefa é implementar um mecanismo de geração de mapas. O input será a dimensão do mapa (número ímpar maior ou igual a 5) 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 usar deverá ser o `StdGen` da biblioteca `System.Random`. Para criar o gerador deverá usar a função `mkStdGen` com a semente recebida como input. Depois de criado poderá usar a função `randomRs` para gerar uma sequência infinita de valores pseudo aleatórios dentro de uma determinada gama.

O mapa do jogo Bomberman é constituído por uma grelha quadrada, podendo cada célula ser vazia, conter um bloco de pedra indestrutível ou um bloco de tijolo destrutível por uma bomba. Alguns blocos de tijolo podem esconder *power ups*, que servem para aumentar a capacidade do jogador. Nesta versão iremos apenas usar os *power ups* `Bombs` e `Flames` que servem para, respectivamente, aumentar o número de bombas que um jogador pode lançar simultaneamente e o raio de acção das mesmas. Por omissão estes valores são de 1 unidade. Os blocos de pedra num mapa estão sempre na mesma posição (todo o contorno do mapa e, para cada linha ou coluna, alternadamente) e os quatro cantos do mapa têm sempre 3 células vazias, para permitir ao jogador em cada canto movimentar-se no início do jogo. O conteúdo das restantes células deve ser determinado a partir de uma sequência de números pseudo-aleatórios entre 0 e 99. Num mapa de dimensão 9, o número de células cujo conteúdo tem que ser gerado é 28, pelo que para determinar o respectivo conteúdo iremos precisar de gerar uma sequência de 28 números aleatórios. Se a semente inicial for 0, essa sequência pode ser gerada da seguinte forma:

```
Prelude System.Random> take 28 $ randomRs (0,99) (mkStdGen 0)
[83,93,63,38,0,87,81,1,61,86,13,50,32,80,54,25,90,31,65,92,2,76,70,25
,6,29,10,99]
```

O conteúdo de cada célula será determinado a partir do respectivo número aleatório, sendo estes distribuídos pelas células sequencialmente, linha por linha. Ou seja, num mapa de dimensão 9, a sequência anterior iria ser distribuída pelas células (cujo conteúdo não está fixo à partida) da seguinte forma:

Pedra	Pedra	Pedra	Pedra	Pedra	Pedra	Pedra	Pedra	Pedra
Pedra	Vazio	Vazio	83	93	63	Vazio	Vazio	Pedra
Pedra	Vazio	Pedra	38	Pedra	0	Pedra	Vazio	Pedra
Pedra	87	81	1	61	86	13	50	Pedra

Pedra	32	Pedra	80	Pedra	54	Pedra	25	Pedra
Pedra	90	31	65	92	2	76	70	Pedra
Pedra	Vazio	Pedra	25	Pedra	6	Pedra	Vazio	Pedra
Pedra	Vazio	Vazio	29	10	99	Vazio	Vazip	Pedra
Pedra	Pedra	Pedra	Pedra	Pedra	Pedra	Pedra	Pedra	Pedra

A forma como se determina o conteúdo de uma célula a partir do respectivo número aleatório é a seguinte:

Gama	**Conteúdo**
0 a 1	*Power up* `Bombs` escondido atrás de um tijolo.
2 a 3	*Power up* `Flames` escondido atrás de um tijolo.
4 a 39	Tijolo.
40 a 99	Vazio.

O mapa deve ser impresso no stdout usando o seguinte formato. Primeiro a grelha é descrita usando o caracter `'#'` para representar pedra, `'?'` para representar tijolo, e `' '` para representar uma célula vazia. Depois deverão ser listados os *power ups* escondidos, um por linha. Para cada power up deve ser indicado o seu tipo (o caracter `'+'` representa `Bombs` e o caracter `'!'` representa `Flames`) e as coordenadas onde se encontra (primeiro horizontal e depois vertical). Os power ups Bombs devem ser listados antes dos Flames, e devem ser ordenados aparecendo primeiro os que aparecem mais acima e mais à esquerda. No caso do mapa de dimensão 9 e semente 0 descrito anteriormente o output do programa deverá ser o seguinte:

```
#########
#       #
# #?#?# #
#  ?  ? #
#?# # #?#
```

```
# ?  ?  #
# #?#?# #
#  ??   #
#########
+ 5 2
+ 3 3
! 5 5
```

No repositório SVN, dentro da directoria `src`, encontrará um ficheiro `Tarefa1.hs`, onde falta implementar a função `mapa :: Int -> Int -> [String]` que, dada a dimensão e semente, deverá devolver uma lista de strings com o conteúdo acima. Neste ficheiro encontra-se implementado um programa `main` que pode ser usado para testar esta função: após compilado, o programa aceita como parâmetros a dimensão e a semente e depois invoca a função `mapa` imprimindo o resultado no `stdout`. O nome deste ficheiro e o respectivo `main` não podem ser alterados.

Tarefa 2 - Reagir a comandos

O objectivo desta tarefa é, dada uma descrição do estado do jogo e um comando de um dos jogadores, determinar o efeito desse comando no estado do jogo. Cada jogador é identificado por um dígito entre 0 e 3 e os comandos podem ser os caracteres `'U'` (ir para cima), `'D'` (ir para baixo), `'L'` (ir para a esquerda), `'R'` (ir para a direita) e `'B'` (colocar uma bomba).

O estado do jogo é representado usando o formato descrito na tarefa anterior, mais uma linha a descrever o estado de cada bomba colocada no mapa, e uma linha a descrever o estado de cada jogador. Uma bomba é identificada pelo caracter `'*'` sendo listada depois a sua posição, qual o jogador que a colocou, qual o seu raio de acção e quantos instantes de tempo faltam para explodir. Um jogador é identificado pelo seu dígito sendo listada depois a sua posição e os *power ups* que entretanto acumulou. O jogo suporta um máximo de 4 jogadores. Bombas e jogadores devem aparecer ordenados por posição e identificador, respectivamente. Por exemplo, considere o seguinte estado do jogo.

```
#########
#       #
# #?# # #
#     ? #
#?# # #?#
# ?  ?  #
# #?#?# #
#  ??   #
#########
```

```
+ 3 3
! 5 5
* 7 7 1 1 10
0 4 3 +
1 7 7
```

Neste caso o *power up* `Bombs` na posição 3 3 já se encontra destapado (o tijolo que o escondia foi entretanto explodido), o jogador 1 acabou de colocar uma bomba de raio 1 (cada bomba demora 10 instantes de tempo a explodir), e o jogador 0 já apanhou um *power up* `Bombs`. Neste caso se o jogador 0 efectuar o comando `'L'` o resultado será:

```
#########
#       #
# #?# # #
#     ? #
#?# # #?#
# ?  ?  #
# #?#?# #
#  ??   #
#########
! 5 5
* 7 7 1 1 10
0 3 3 ++
1 7 7
```

Sempre que um comando não possa ser executado o resultado deverá ser o mesmo estado. Note que na mesma célula podem coexistir mais do que um jogador e um jogador pode também estar posicionado numa célula onde se encontra uma bomba. No entanto, na mesma célula não pode estar mais do que uma bomba. Assuma que os comandos têm um efeito imediato: programar o efeito da passagem do tempo (nomeadamente, nos relógios das bombas) será o objetivo da primeira tarefa da 2ª fase do projeto.

No repositório SVN, dentro da directoria `src`, encontrará um ficheiro `Tarefa2.hs`, onde falta implementar a função `move :: [String] -> Int -> Char -> [String]` que, dado o estado actual do jogo no formato acima descrito, o identificador de um jogador e o comando, deverá devolver o novo estado do jogo. Neste ficheiro encontra-se implementado um programa `main` que pode ser usado para testar esta função: após compilado, o programa aceita como parâmetros o identificador de um jogador e um comando, fica à espera do estado do jogo no `stdin` e invoca a função `move`, imprimindo o resultado no `stdout`. O nome deste ficheiro e o respectivo `main` não podem ser alterados.

Tarefa 3 - Comprimir o estado do jogo

O objectivo desta tarefa é, dada uma descrição do estado do jogo (idêntica à utilizada na tarefa anterior) implementar um mecanismo de compressão / descompressão que permita poupar caracteres e, desta forma, poupar espaço em disco quando o estado do jogo for gravado (permitindo, por exemplo, fazer pausa durante o jogo com o objectivo de o retomar mais tarde). Uma sugestão que pode implementar passa por eliminar redundâncias na representação do estado do jogo, nomeadamente na parte do mapa: tal como mencionado na tarefa 1 existem células do mapa cujo conteúdo é fixo, não sendo na prática necessário guardar informação sobre o conteúdo dessas células. Esta é apenas uma sugestão existindo muitas outras técnicas que podem implementar para obter melhores taxas de compressão.

No repositório SVN, dentro da directoria `src`, encontrará um ficheiro `Tarefa3.hs`, onde falta implementar as funções `encode :: [String] -> String` e `decode :: String -> [String]` cujo objectivo é, respectivamente, codificar o estado do jogo para uma `String` e voltar a descodificar. O mecanismo de codificação tem que ser invertível, ou seja, dado um estado do jogo qualquer `s`, tem que ser sempre verdade que `decode (encode s) == s`. Uma forma trivial de satisfazer este critério é não fazer qualquer compressão, usando a seguinte implementação trivial.

```
encode :: [String] -> String
encode l = unlines l

decode :: String -> [String]
decode l = lines l
```

No entanto, pretende-se uma implementação que faça alguma compressão, ou seja uma implementação onde, para um estado do jogo qualquer `s`, `length (encode s)` seja bastante menor do que `length (unlines s)`.

No ficheiro `Tarefa3.hs` encontra-se também implementado um programa `main` que pode ser usado para testar estas funções: após compilado, o programa aceita como parâmetros a opção `-e` (para codificar) e `-d` (para descodificar), fica à espera do estado do jogo (normal ou comprimido) no `stdin`, e invoca a função `encode` (ou `decode`, dependendo da opção), imprimindo o resultado no `stdout`. O nome deste ficheiro e o respectivo `main` não podem ser alterados.

Entrega e Avaliação

A data limite para conclusão de todas as tarefas desta primeira fase é **20 de Novembro de 2017** e a respectiva avaliação terá um peso de 40% na nota final da UC. 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	15%
Avaliação automática da tarefa 2	15%
Avaliação automática da tarefa 3	15%
Avaliação qualitativa da tarefas	20%
Utilização do SVN e estrutura do repositório	10%
Quantidade e qualidade dos testes	15%
Documentação do código usando o Haddock	10%

A nota final é 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. No caso da tarefa 3, a avaliação automática também terá em conta a taxa de compressão obtida. A avaliação qualitativa incidirá sobre aspectos da implementação não passíveis de ser avaliados automaticamente (por exemplo, estrutura do código, elegância da solução implementada, etc).