Haskassonne

Informação geral

Informação geral sobre o jogo *Carcassonne*, sobre o qual o *Haskassonne* se baseia, pode ser consultada nas regras do jogo publicadas na plataforma de elearning ou em http://en.wikipedia.org/wiki/Carcassonne_(board_game)

Regras do Jogo

As regras do jogo *Carcassonne* podem ser encontradas na página oficial do mesmo em http://riograndegames.com/games.html?id=48 ou no vídeo em http://www.youtube.com/watch?v=AcZ6ndkdDCs. O *Haskassonne* é uma variante simplificada do *Carcassonne*, apenas com as seguintes peças de território:


B


C


E


N

Note que a peça C difere ligeiramente da existente no jogo *Carcassonne* (não possui escudo). Como não serão usadas peças com estradas, a peça E deve ser jogada na primeira ronda. Nesta versão alternativa existem 6 peças de tipo B, 2 de tipo C, 18 de tipo E, e 10 de tipo N (total de 36 peças de território). O jogo *Carcassonne* estará disponível às Quartas-feiras à tarde (14h30) na sala DI 1.08, para quem quiser esclarecer as regras e desenvolver estratégias de jogo.

Representação XML de um estado do jogo

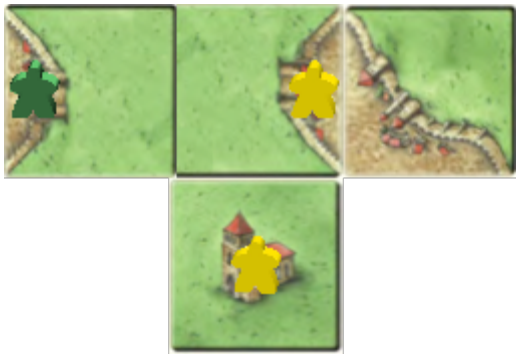
O XML é um formato de codificação de informação muito popular hoje em dia. Mais informações sobre este formato em http://en.wikipedia.org/wiki/XML. Neste projecto vamos usar este formato para representar os diferentes estados do jogo. Um estado do jogo será representado pelo elemento `board`. Dentro este elemento encontram-se dois elementos obrigatórios: `terrain`, que descreve o estado actual do terreno, e `scores`, que descreve as pontuações actuais. O terreno é descrito por uma sequência de elementos `tile`, cada um tendo como atributos obrigatórios o tipo da peça (B, C, E ou N), as coordenadas onde se encontra a peça (a primeira peça colocada no tabuleiro está na origem), e a orientação da mesma (N, S, E, W). Opcionalmente, dentro do elemento `tile` pode aparecer um elemento `follower` (também

conhecido por *meeple*) caso exista algum *follower* nessa peça. Neste caso será indicado nos atributos qual o jogador a que pertence o mesmo e qual o seu tipo (K, de *knight*, quando se encontra na cidade; M, de *monk*, quando se encontra no claustro; ou F, de *farmer*, quando se encontra no campo). Note que para as peças que vamos usar dado o tipo do *follower* nunca é ambíguo onde se encontra. As pontuações são indicadas por uma sequência de elementos `score`, tendo como atributos o identificador de cada jogador e a respectiva pontuação. Caso o jogo ainda não tenha terminado, existirá também um elemento `next` que indica qual o tipo da peça que o próximo jogador terá que jogar.

A seguir se apresenta um possível exemplo de um documento XML, descrevendo um estado válido do jogo.

```
<?xml version="1.0"?>
<board>
     <terrain>
          <tile type=”E” x=”0” y=”0” orientation=”E”>
               <follower player=”1” type=”K”/>
          </tile>
          <tile type=”N” x=”1” y=”0” orientation=”W”/>
          <tile type=”B” x=”0” y=”-1” orientation=”N”>
               <follower player=”1” type=”M”/>
          </tile>
          <tile type=”E” x=”-1” y=”0” orientation=”W”>
               <follower player=”2” type=”K”/>
          </tile>
     </terrain>
     <scores>
          <score player=”1” score=”0”/>
          <score player=”2” score=”0”/>
     </scores>
     <next tile=”E”/>
</board>
```

O terreno descrito neste documento XML corresponde ao seguinte estado de jogo (assumindo que o jogador 1 joga de amarelo e o jogador 2 de verde).



Programas a implementar

Neste projecto devem implementar os seguintes programas:

1. `draw` - dada uma descrição do tabuleiro em XML no `stdin` este programa deve "desenhar" no `stdout` o terreno usando caracteres ASCII. Cada peça será representada por uma matriz de 5x5 caracteres, onde se usa `'*'` para cidade, `´.´` para campo, `'O'` para claustro, e digitos para representar os followers de cada jogador. Mais concretamente, pretende-se a seguinte representação (onde i é o digito que identifica o jogador que colocou o *follower*) quando a peça está orientada para Norte:

Tipo	Empty	Knight	Monk	Farmer

B	`.....`
	`..O..`
	`.OOO.`
	`..O..`
	`.....`
			`.....`
			`..O..`
			`.OiO.`
			`..O..`
			`.....`
				`..i..`
				`..O..`
				`.OOO.`
				`..O..`
				`.....`

C	`*****`
	`*****`
	`*****`
	`*****`
	`*****`
		`*****`
		`*****`
		`**i**`
		`*****`
		`*****`		

E	`*****`
	`.***.`
	`..*..`
	`.....`
	`.....`
		`*****`
		`.*i*.`
		`..*..`
		`.....`
		`.....`
				`*****`
				`.***.`
				`..*..`
				`.....`
				`..i..`

N	`*****`	`*****`		`*****`

	****. ***.. **... *....	*i**. ***.. **... *....		****. ***.. **.i. *....

No caso do tabuleiro acima, o programa deverá imprimir o seguinte resultado

```
*........**....
**......****...
*2*....*1****..
**......******.
*........******
     .....
     ..0..
     .010.
     ..0..
     .....
```

2. `play` - dada uma descrição do tabuleiro em XML no `stdin` este programa deve produzir uma jogada para a peça indicada na tag `next`. Neste caso o output deve ser apenas uma tag do tipo `tile`, por exemplo

```
<tile type=”E” x=”1” y=”-1” orientation=”N”/>
```

ou

```
<tile type=”E” x=”0” y=”-2” orientation=”S”>
     <follower player=”1” type=”F”/>
</tile>
```

Aqui podem implementar várias versões do programa que seguem estratégias diferentes. O aspecto mais importante a ter em consideração é a correcção da estratégia, ou seja garantir que a jogada produzida coloca a peça numa posição adjacente a uma já existente no tabuleiro (ou na origem caso seja a primeira jogada), e que não são colocados mais do que 7 *meeples* de cada jogador, nem em cidades ou campos já controlados por outros *meeples*.

3. `next` - dada uma descrição do tabuleiro em XML no `stdin` este programa deve actualizar as pontuações, retirar os *meeples* que pontuam e, caso o jogo ainda não tenha terminado, sortear uma peça para colocar no elemento `next`. Note que, caso já tenham sido jogadas todas as peças, o jogo termina, e não deve ser gerada o elemento `next` e devem ser usadas as regras de fim de jogo para calcular as pontuações (e

retirados todos os *followers*). Este programa deve devolver no `stdout` uma nova representação do tabuleiro em XML.

Os programas `draw`, `play` e `next` deverão ser implementados em Haskell. A correção destes será avaliada automaticamente usando o Mooshak. A qualidade da implementação (nomeadamente das estratégias implementadas no programa `play`) será avaliada através da leitura do relatório e na discussão oral.

Torneio final

No final do semestre será organizado um torneio virtual para determinar qual a melhor implementação do programa `play`. O resultado deste torneio não será tido em conta na avaliação, mas o grupo vencedor (excluindo os grupos com alunos em regime de melhoria de nota) receberá como prémio o jogo *Carcassonne*. Cada *match* do torneio consistirá numa série de jogos entre 2 grupos, vencendo o grupo que ganhar mais jogos. Se o programa `play` de um dos grupos gerar uma jogada inválida perde imediatamente o jogo.

Durante o semestre será disponibilizado um programa que permite executar um match dadas duas implementações do programa `play`. Esse programa pode ser usado para realizar treinos para o referido torneio e aferir a qualidade da estratégia implementada.

Relatório e documentação

O projecto deve ser documentado num relatório escrito em LaTeX. Nesse relatório devem descrever a estratégia que seguiram para implementar cada um dos programas, destacando os tipos de dados usados e as funções principais implementadas. O relatório deve ser sucinto, mas suficientemente detalhado para que, apenas recorrendo à sua leitura, o docente seja capaz de avaliar a qualidade da implementação. A discussão oral tem apenas por objectivo esclarecer dúvidas sobre a implementação e aferir o conhecimento e destreza dos elementos do grupo com as ferramentas utilizadas no desenvolvimento do projecto.

Todo o código Haskell deve ser documentado usando Haddock (http://www.haskell.org/haddock/). A qualidade e extensão desta documentação será também avaliada.

Testes

Os programas `draw`, `play` e `next` devem ser testados extensivamente da seguinte forma:

- Usando QuickCheck (http://hackage.haskell.org/package/QuickCheck) para especificar

propriedades e testes unitários sobre o maior número possível de funções desenvolvidas em Haskell.

- Usando testes unitários para os diferentes programas: o comportamento esperado para cada um deles deve ser verificado executando-os com um conjunto alargado de entradas (documentos XML descrevendo possíveis estados do jogo) para as quais o resultado é conhecido e contra o qual será comparado. Estes testes devem, na medida do possível, ser automatizados recorrendo a scripts desenvolvidas usando a linha de comando.

A qualidade e extensão dos testes implementados será uma das componentes sujeita a avaliação.

SVN e Makefile

Todo o código fonte (incluindo do relatório) do projecto deve ser desenvolvido com recurso ao servidor SVN disponibilizado ao grupo. Devem fazer commits frequentemente ao longo do semestre. Os logs do servidor serão avaliados no final do semestre para verificar a correcta (e frequente) utilização do mesmo por ambos os elementos do grupo.

Devem também definir uma `Makefile` com, pelo menos, os seguintes *targets*:

- Compilar separadamente todos os programas solicitados no projecto.
- Gerar o pdf do relatório a partir das respectivas fontes em LaTeX.
- Gerar HTML para o código documentado com Haddock.
- Verificar os testes que foram implementados.
- Executar todos os *targets* anteriores.
- Limpar todos os artefactos gerados pelos *targets* anteriores.

Sugestões para a implementação

- Package Haskell para processamento de XML: http://hackage.haskell.org/package/xml. Usando este package o programa `draw` pode ser estruturado da seguinte forma, onde fica a faltar a definição da função `processa`:

```
import Text.XML.Light

main = do entrada <- getContents
          let Just elem = parseXMLDoc entrada
          putStrLn $ processa elem

processa :: Element -> String
```

O programa `play` pode ser estruturado da seguinte forma, onde fica a faltar a definição da função `processa`:

```
import Text.XML.Light

main = do entrada <- getContents
          let Just elem = parseXMLDoc entrada
          putStrLn $ showElement (processa elem)

processa :: Element -> Element
```

O programa `next` pode ser estruturado da seguinte forma, onde fica a faltar a definição da função `processa`, que recebe como primeiro argumento um inteiro gerado aleatoriamente:

```
import Text.XML.Light
import System.Random

main = do entrada <- getContents
          let Just elem = parseXMLDoc entrada
          seed <- randomIO
          putStrLn $ showElement (processa seed elem)

processa :: Int -> Element -> Element
```

- Para verificar se um ficheiro `tabuleiro.xml`, contendo uma descrição de um estado do jogo, está correctamente definido podem usar o comando `xmllint --dtdvalid haskassonne.dtd tabuleiro.xml`, onde o ficheiro `haskassonne.dtd` contém o seguinte DTD com as regras de validação:

```
<!ELEMENT board (terrain, scores, next?)>

<!ELEMENT terrain (tile*)>

<!ELEMENT tile (follower?)>
<!ATTLIST tile type        (B|C|E|N) #REQUIRED
               x           CDATA     #REQUIRED
               y           CDATA     #REQUIRED
               orientation (N|S|E|W) #REQUIRED>

<!ELEMENT follower EMPTY>
<!ATTLIST follower player CDATA #REQUIRED
                   type   (M|K|F) #REQUIRED>
```

```
<!ELEMENT scores (score+)>

<!ELEMENT score EMPTY>
<!ATTLIST score player CDATA #REQUIRED
                score  CDATA #REQUIRED>

<!ELEMENT next EMPTY>
<!ATTLIST next tile (B|C|E|N) #REQUIRED>
```

- Recomenda-se que estruturem o repositório SVN da seguinte forma:
  - `README` (ficheiro com a identificação do grupo)
  - `Makefile` (ficheiro com a makefile do projecto)
  - `src` (directoria com o código fonte)
  - `bin` (directoria para onde são compilados os executáveis)
  - `report` (directoria onde se encontra o relatório em LaTeX)
  - `doc` (directoria para onde é gerada a documentação Haddock)
  - `test` (directoria que contém os ficheiros XML usandos para testar os programas)

Avaliação

O projecto deve ser realizado em grupos de 2 alunos, e será avaliado segundo os seguintes parâmetros:

Programas `draw`, `play` e `next`: qualidade da implementação (avaliação pela leitura do relatório e pela discussão)	30%
Programas `draw`, `play` e `next`: correção da implementação (avaliação automática)	25%
Desenvolvimento de testes para os programas `draw`, `play` e `next` (usando QuickCheck e linha de comando)	10%
Documentação do código dos programas `draw`, `play` e `next` (usando Haddock)	5%
Utilização do sistema de controlo de versões	5%
Definição da makefile	5%
Relatório (conteúdo)	10%
Relatório (execução em LaTeX)	10%

A avaliação dos 2 elementos do grupo pode ser diferente, dependendo da respectiva prestação.

Qualquer cópia que seja detectada leverá à anulação dos projectos de todos os grupos envolvidos, com a consequente reprovação à unidade curricular. Dependendo da gravidade da mesma, pode também ser levantado um processo disciplinar aos envolvidos.