# Haskassone

Bruno Alexandre Alves Ferreira
a61055@alunos.uminho.pt

Nuno Salvador
a?????@alunos.uminho.pt

Grupo 007

26 de Dezembro de 2013

**Resumo**

Esta é uma implementação em Haskell do Carcassone.

# Conteúdo

# 1 Resumo

Neste relatório encontram-se explicitadas de forma detalhada as escolhas do *grupo 007* no que ao projecto da unidade curricular de Laboratórios de Informática I diz respeito. Serão revistas as escolhas em relação aos tipos de dados usados, a forma como se interligam nos vários programas e outras decisões tomadas para resolver o problema que nos foi proposto.

## 2 Introdução

Pretende-se implementar, em Haskell, uma versão simplificada do famoso jogo de tabuleiro Carcassonne. Esta versão simplificada contém apenas 4 tipos distintos de peças e não contempla estradas.

(a) Peça B

(b) Peça C

(c) Peça E

(d) Peça N

De forma resumida, o Carcassonne é um jogo de tabuleiro que se expande à medida que o jogo avança. O jogo começa com uma única peça em jogo. No seu turno, os jogadores devem retirar aleatoriamente uma nova peça e coloca-la adjacente a uma peça do terreno. Esta nova peça deve ser colocada de forma concordante com as peças que lhe são adjacentes.

Depois de colocar a sua peça em jogo, o jogador pode optar por colocar um *meeple* numa parte da peça colocada e assim dominar aquela zona. O jogador não pode colocar um *meeple* numa zona dominada por outro *meeple*, no entanto pode acontecer que zonas dominadas por jogadores diferentes passem a ser partilhadas quando se coloca uma nova peça que junte duas zonas.

O jogo termina quando a ultima peça é colocada, ou quando não é possível colocar as restantes peças. Nesse momento, todas as zonas são pontuadas para o jogador que as domina. O jogador com mais pontos ganha o jogo.

# 3 Desenvolvimento do Projecto

## 3.1 Requisitos

Para implementar o Haskassonne, este foi dividido em 3 fases mais simples. Todas estas fases recebem como input um ficheiro XML representante do estado actual do jogo.

**Draw** Imprimir uma representação do terreno em texto simples.

**Play** Produzir uma jogada válida para o jogador actual.

**Next** Escolher o tipo da próxima peça a ser jogada ou, caso o jogo tenha terminado, calcular as pontuações dos jogadores e remover todos os *meeples*.

Torna-se assim possível executar os programas *Play* e *Next* repetidamente para conseguir sucessivas jogadas. Pode-se a qualquer momento fornecer o estado ao *Draw* e obter uma representação visual das peças colocadas e dos *meeples* em campo.

## 3.2 Principais Tipos de Dados

Ao longo do relatório faremos várias referências aos tipos de dados usados ou aos elementos que eles representam, assim sendo considerou-se essencial a sua explicação numa fase inicial do relatório.

Estes tipos de dados são usados para conseguir uma representação simplificada, em Haskell, da representação fornecida em XML. Existe, no nosso projecto, um módulo responsável por converter um ficheiro XML na representação interna e um segundo módulo que realiza a funcionalidade inversa.

### 3.2.1 Player (módulo Leitor)

Representa um jogador, que é identificado por um número inteiro entre zero e nove, inclusive. O jogador tem também uma pontuação.

**s_player::Int** Identificação do jogador.

**s_score::Int** Pontuação do jogador.

### 3.2.2 Next (módulo Leitor)

Informações obtidas na tag *next* do XML do estado.

**n_tile::Char** A peça que deve ser colocada na próxima jogada.

### 3.2.3 *Meeple* (módulo Leitor)

Um *meeple* é um marcador que um jogador pode optar por colocar numa zona de um *tile* que acabou de jogar. Este *meeple* é que permite aos jogadores pontuar.

Um *meeple*, com informação sobre o seu dono e tipo. O tipo do *meeple* (Farmer, Knight ou Monk) define a zona onde este está colocado num *Tile*.

**m_player::Int** Identificação do jogador.

**m_type::Char** O tipo do *meeple* (F, K ou M).

### 3.2.4 *Tile* (módulo Leitor)

Um *tile* representa uma peça do terreno.

**t_type::Char** O tipo da peça (B, C, E ou N).

**t_x::Int** A posição horizontal da peça.

**t_y::Int** A posição vertical da peça.

**t_orientation::** A orientação da peça (N, S, W ou E).

**t_meeple::Maybe Meeple** Um *meeple*, caso exista.

### 3.2.5 *Board* (módulo Leitor)

Representa um estado do tabuleiro, com os *tiles*, *meeples*, jogadores e informações sobre a próxima jogada.

**b_terrain::[Tile]** As peças em jogo.

**b_scores::[Player]** Os jogadores.

**b_next::Next** Informações sobre a próxima jogada.

### 3.2.6 *Map* (módulo Leitor)

Uma matriz de *Maybe Tile* para representar um mapa. Nesta representação as peças estão organizadas de acordo com os seus campos *t_x* e *t_y*.

Existe a possibilidade de não existir um *tile* numa posição (x,y), daí a necessidade de ter a matriz de *Maybe Tile*.

**[[Maybe Tile]]** Matriz com os *tiles* em jogo.

## 3.3 Tipos de Dados Auxiliares

Estes tipos de dados têm um papel menos significativo. No entanto facilitam a execução de alguma tarefas.

### 3.3.1 Limits (módulo Leitor)

Representa os limites exteriores do mapa.

**l_Xmin::Int** Valor mínimo para o X.

**l_Xmax::Int** Valor máximo para o X.

**l_Ymin::Int** Valor mínimo para o Y.

**l_Ymax::Int** Valor máximo para o Y.

### 3.3.2 Location (módulo Leitor)

**(Int, Int)** Localização (x,y) de um *tile*.

### 3.3.3 ScoredTile (módulo Pontuar)

Um par constituído por (Zone, Int). Com este par torna-se mais fácil relacionar uma zona com o número de pontos que lhe está associado.

**::Zone** Representa uma zona.

**::Int** A pontuação atribuída a essa zona.

### 3.3.4 Zone (módulo Tabuleiro)

Representa uma zona do mapa: uma cidade, um campo ou um claustro.

O primeiro *tile* identifica a peça por onde se começou a zona. Os *tiles* seguintes identificam os *tiles* por onde se estende a mesma zona (neste conjunto deve estar incluído o *tile* onde a zona começou).

No caso do claustro considera-se a zona como os 8 *tiles* à volta do claustro e o próprio claustro.

Definem-se da seguinte forma:

**City Tile [Tile]** Representa uma cidade.

**Field Tile [Tile]** Representa um campo.

**Cloister Tile [Tile]** Representa um claustro.

### 3.3.5 Side (módulo Tabuleiro)

Representa uma lateral de uma peça.
Isto facilita o reconhecimento de todos os *tiles* de uma zona.
Define-se da seguinte forma:

**SideField** Representa uma lateral de campo.

**SideCity** Representa um campo.

**SideBoth** Representa um claustro.

### 3.3.6 Sides (módulo Tabuleiro)

Representa as laterais de uma peça.

**(Side, Side, Side, Side)** Representa as laterais de um *tile*.

### 3.3.7 Art (módulo ArtASCII)

**[String]** Representa um tile em formato de texto simples.

## 3.4 Módulos

### 3.4.1 Leitor

Utiliza o módulo *Text.XML.Light* para converter os dados XML para os tipos de dados adaptados ao problema.

### 3.4.2 Tabuleiro

Contém uma série de utilitários para manipulação do tabuleiro.

### 3.4.3 ArtASCII

Permite o desenho do terreno em formato de texto simples.

### 3.4.4 Escritor

Tem a funcionalidade inversa ao módulo *Leitor*: converter os dados (organizados em tipos de dados adaptados ao problema) para um tipo de dados intermédio que depois o módulo *Text.XML.Light* consegue converter para XML.

### 3.4.5 Pontuar

Contém utilitários para contagem e atribuição de pontuação.

### 3.4.6 FakePrettyShow

Define uma função *ppShow* que faz apenas um *show*. Este módulo é importado em vez do *Text.Show.Pretty* quando se quer submeter programas no Mooshak (o Mooshak não suporta *Text.Show.Pretty*).

### 3.5 Programa Draw

Descreve-se agora o funcionamento geral do programa Draw.

**Algorithm 1:** Funcionamento geral do Draw

**input**: XML com o estado atual

importa o XML para um Board
obtém os limites exteriores do Board
constroi o Map

Matriz de Art←**forall the** *maybeTiles of Map* **do**
| `convertMaybeTileToArt`
**end**
imprime a Matriz de Art

### 3.6 Programa Play

Descreve-se agora o funcionamento geral do programa Play.

**Algorithm 2:** Funcionamento geral do Play

**input**: XML com o estado atual

importa o XML para um Board
tipoDefinido ←tipo da próxima peça a ser jogada.

**if** *não existem peças em jogo* **then**
  tileEscolhido ←`tileAleatorioDoTipo('E')` na posição (0,0)
**else** `/* existem peças em jogo */`
  tilesPossiveis ←todos os tiles que podem ser jogados
  **if** *jogador pode colocar Meeple* **then**
    tilesPossiveis ←tilesPossiveis ++ tiles possíveis com meeples
  **end**
  tileEscolhido ←seleccionar aleatoriamente um dos tilesPossiveis
**end**

**if** *não recebeu nenhum argumento* **then**
  converte o tileEscolhido para uma tag XML e imprime-a
**else**
  adiciona o tileEscolhido ao Board
  converte o Board num Element
  imprime o Element em formato XML
**end**

A nossa implementação do programa Play tem uma funcionalidade extra: quando lhe é passado qualquer argumento, imprime um ficheiro de estado XML completo, em vez de apenas a tag com o *Tile* jogado. Esta alteração permite-nos encaminhar o output do programa Play para o programa Next e assim jogar uma "ronda".

### 3.7 Programa Next

Descreve-se agora o funcionamento geral do programa Next.

**Algorithm 3:** Funcionamento geral do Next

```
input: XML com o estado atual

importa o XML para um Board
tipoEscolhido ←tipo da próxima peça a ser jogada.

if podem ser jogadas mais peças then
    tilesPossiveis ←todos os tiles que podem ser jogados
    tile ←seleccionar aleatoriamente um dos tilesPossiveis
    tipoEscolhido ←tipo do tile
    alterar o Next do Board com o tipoEscolhido
    pontuar as cidades completas
    pontuar os claustros completos
    converte o Board num Element
else                                          /* jogo terminou */
    colocar no Next do Board o tipo '-'
    pontuar as cidades
    pontuar os claustros
    pontuar os campos
    converte o Board num Element
end

if não recebeu nenhum argumento then
    imprime o Element em formato XML (numa linha)
else
    imprime o Element em formato XML (legível)
end
```

A nossa implementação do programa Next tem uma funcionalidade extra: quando lhe é passado qualquer argumento, imprime um ficheiro de estado XML legível (com várias linhas e tabulações), em vez de um XML *"one-liner"*. Esta alteração permite-nos ler e interpretar manualmente o ficheiro XML de forma mais rápida.

Colocar um '-' no *Next* do *Board* faz com que o módulo *Escrever* o ignore, resultando num formato XML sem a tag next.

### 3.8 Algoritmos das zonas

Quando precisamos de calcular uma *Zone* (cidade ou campo) fazemos uma pesquisa breadth-first que começa no *Tile* que tem o *meeple* e percorre todos os *tiles* adjacentes que tenham laterais concordantes.

A *Zone* de um claustro são os 8 tiles adjacentes ao claustro e o próprio claustro.

Este tipo de dados e forma de resolver o problema permitem-nos depois dar diversos usos à *Zone*:

- Contar o número de *tiles* numa zona consiste em contar o número de elementos na lista de *tiles*. No caso do claustro, deve-se subtrair um ao número de elementos nessa lista.
- Todas as zonas com os mesmos elementos na lista de *tiles* são dominadas por mais que um meeple. Isto permite-nos saber quantos *meeples* de cada jogador dominam uma zona e atribuir correctamente os pontos.
- Remover o *meeple* que domina uma zona consiste em obter a localização do tile inicial da *Zone*, pesquisar o *tile* com essa localização no Board, remover o *meeple* e adicionar uma unidade ao número de *meeples* disponíveis desse jogador.

O algoritmo que verifica se uma cidade já está completa (no programa Next) é bastante semelhante, utilizando uma pesquisa breadth-first que percorre apenas os *tiles* de cidade e termina caso a cidade não esteja completa ou quando não houver mais *tiles* na orla (que significa que a cidade está completa).

# 4 Makefile

A *Makefile* contém várias regras principais:

| | |
|---|---|
| **all** | Compila os três programas. |
| **clean** | Apaga ficheiros binários e temporários. |
| **draw** | Compila o programa Draw. |
| **play** | Compila o programa Play. |
| **next** | Compila o programa Next. |
| **doc** | Gera a documentação para as classes do projecto. |

# 5 Testes

## 5.1 Ficheiro TestDraw.hs

Foi feita uma tentativa de utilizar o QuickCheck no nosso projecto. Não nos ocorreram formas intuitivas de testar funcionamento complicado usando a geração de dados aleatórios, por isso os nossos testes usando QuickCheck resumem-se a testar se a rotação de Art e a remoção de duplicados em listas funciona.

Por outro lado a nossa script de jogar uma ronda providenciou-nos todo o tipo de situações para testar o nosso projecto.

## 5.2 Ficheiros XML de teste

Foram criados vários ficheiros XML para testes na directoria test. Estes servem para testar algumas funcionalidades básicas do programa, sendo que as mais avançadas foram testadas usando a script de jogar uma ronda.

## 5.3 Script: Jogar uma ronda

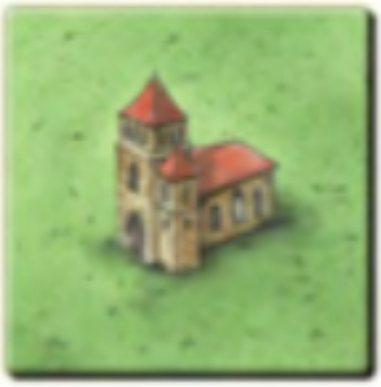
Foi criada uma script que "joga uma ronda". Caso não exista nenhum XML de estado, esta script começa por usar o ficheiro de testes com um tabuleiro vazio.

A script executa em sequência os programas Play e Next, executando antes e depois o programa Draw. Utilizando as funcionalidades extra que foram adicionadas aos programas Play e Next (que são activadas quando se fornece um argumento a estes programas) e gravando os vários XML que são produzidos pelos programas, conseguimos ter todo o tipo de testes e situações para por a nossa solução à prova.

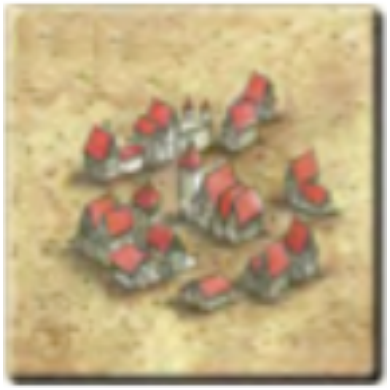
Antes de terminar, a script coloca lado a lado o output do programa Draw antes e depois de jogar a ronda. Assim consegue-se ter uma perspectiva geral rápida sobre a evolução do jogo.

O verdadeiro potencial desta script é realizado quando ela é executada várias vezes. Desta forma consegue-se, partindo de um tabuleiro vazio chegar a um tabuleiro de fim de jogo, jogando ronda após ronda.

# 6 Elementos do Grupo



(e) Bruno Ferreira (a61055)



(f) Nuno Salvador (a?????