Le projet : jouons !!!

L'objectif du projet est d'écrire un module `Game`, permettant de jouer un jeu arbitraire à deux personnes (à l'aide de l'ordinateur). L'utilisateur jouera le rôle de `Adam`, l'ordinateur celui de `Eve`. La machine jouera en disposant d'une stratégie optimale (pour `Eve`), c'est-à-dire gagnante si il y en existe une, ou une stratégie quelconque (tirer des coups au hasard) sinon.

Modélisation d'un jeu

Un jeu sera un type de données (quelconque), dont les valeurs (ses éléments) sont les positions. Étant donnée une position, on aura une méthode pour connaître le joueur qui doit jouer, et quelles sont les positions suivantes parmi lesquelles il devra choisir. On connaîtra aussi la position initiale du jeu.
Nous pouvons attendre cet objectif en définissant les types suivants

```
data Player = Adam | Eve
type Moves a = [a]
```

et en définissant la collection de tous les jeux comme l'ensemble des types munis d'une position initiale, d'une méthode pour connaître, en fonction de la position courante, le joueur qui doit jouer et, enfin, d'une méthode pour connaître les mouvements possibles. C'est-à-dire, nous définissons la classe `Game` comme suit :

```
class Game a where
    initial :: a
    player :: a -> Player
    moves :: a -> Moves a
```

Remarque. Une position `pos` d'un jeu `game` sera considérée finale si `moves pos` s'évalue à la liste vide. Dans ce cas, on considère la position `pos` gagnante pour `Eve` si `player pos` est `Adam` ; autrement, `pos` est une position gagnante pour `Adam`. La raison en est que si un joueur est sensé jouer, et il n'a à disposition aucun choix, alors il est perdant.

Exemple. Voici comment le jeu du nim peut se représenter comme une instance de cette classe :

```
import Game

data Nim = Nim Player [Int] deriving (Eq,Show)

startPosition =  Nim Adam [5,4,3,2,1]

choices (Nim thisplayer heaps) =
  [ Nim nextplayer (update heaps indexHeap nbTokenLeft)
  | indexHeap <- [1..nbHeap],
    nbTokenLeft <- [0.. (heaps !! (indexHeap-1) - 1)]
  ]
    where
      nbHeap = length heaps
      nextplayer = flip thisplayer
      update values index newValue =
        take (index-1) values ++ [newValue] ++ drop index values

instance Game Nim where
    initial = startPosition
    player (Nim pl _) = pl
    moves = choices
```

Objectif. En suivant l'exemple du jeu Nim, écrivez deux modules similaires, pour le jeu de soustraction (partant d'un entier, chaque joueur à tour de rôle soustrait 1, 2, ou 3, le joueur qui atteint 0 a gagné) et le jeu du Morpion (voir http://fr.wikipedia.org/wiki/Morpion_jeu). Pour contraindre le jeu du Morpion à un jeu sans matchs nuls, on pourra déclarer une partie nulle comme gagnante pour le deuxième joueur.

Résoudre un jeu

Une solution d'un jeu est une fonction associant à chaque position le jouer qui possède une stratégie gagnante à partir de cette position. Nous considérerons des solutions partielles—donc des fonctions partielles—car l'ensemble des positions peut, en principe, être infini ; une fonction partielle pourra se représenter, de façon usuelle, par des listes associatives (voir http://en.wikipedia.org/wiki/Association_list). Ce ne sera pas suffisant pour résoudre un jeu avec plusieurs milliers de positions différentes : dans ce cas elle pourra aussi se représenter par des structures de données plus efficaces disponibles comme modules en Haskell (voir par exemple http://www.haskell.org/ghc/docs/latest/html/libraries/containers/Data-Map.html). On définira donc un type des solutions :

```
type Solution a = ...
```

On peut calculer une solution partielle incluant une position de la façon suivante : on calculera d'abord une solution incluant toutes les positions suivantes ; puis, la position actuelle sera déclarée gagnante pour `Eve` ssi

1. le joueur de cette position est `Adam` et toutes les positions suivantes sont des positions gagnantes pour `Eve` ;
2. le joueur de cette position est `Eve` et au moins une des positions suivantes est gagnante pour `Eve`.

Le type de cette fonction sera le suivant :

```
solve :: (Game a,Eq a) => a -> Solution a
```

(La contrainte `Eq a` est due au fait que l'on souhaite utiliser des listes associatives pour les solutions, mais on peut aussi prendre `Ord a` si on souhaite utiliser le module `Map` pour coder les solutions partielles).

Jouer un jeu

Nous allons écrire ensuite une petite interface permettant de jouer un jeu à deux personnes arbitraire :

```
play :: (Game a,Eq a,Show a) => a -> IO ()
```

Dans le type de cette fonction, la contrainte `Show a` vient du fait que l'on souhaite représenter une position via une chaîne de caractères, pour affichage dans une interface utilisateur.
Il faudra donc instancier chaque jeu à la classe `Show`, ou même déclarer la classe `Game` comme une extension de la classe `Show`. Définissez les méthodes `show` pour les jeux considérés de façon appropriées (évitez l'instanciation automatique avec `deriving` dès que possible).
Les coups de `Adam` seront choisis par un utilisateur ; les coups de `Eve` seront choisis par la machine de cette façon :

1. si la position est gagnante pour `Eve`, alors on sait qu'il y a au moins un coup possible vers une autre position gagnante ; on choisira donc un de ces coups,
2. sinon, on choisira un coup à hasard.

Évidemment, pour pouvoir jouer au jeu, il faudra pré-calculer une solution.

Développements ultérieurs

1. Modifiez les module `Game`, de façon qu'il soit possible prendre en compte les matchs nuls.
2. Paramétrez la fonction `play` de facon à associer à chaque joueur une facon de procéder parmi :
   (a) jouer en utilisant un humain,
   (b) jouer en tirant des mouvements à hasard,
   (c) jouer en suivant une stratégie optimale (comme pour `Eve` dans le paragraphe précédent),
   (d) jouer avec un taux donné de maîtrise : par exemple taux 0 équivaut à dire que tout coup est choisi à hasard, taux 5 vaut dire que 50 pourcent des coups sont choisis à l'aide d'une stratégie gagnante, taux 10 veut dire que tous les coups sont choisis en utilisant une stratégie gagnante.
3. Certains jeux peuvent commencer avec diverses positions initiales, comme le jeu de Nim. Modifier `Game` pour rendre ceci possible.
4. Pour le morpion, le nombre de positions possibles du jeu est très élevé, et le calcul d'une stratégie gagnante prend un peu de temps. On peut alors remarquer que le plateau étant carré, il existe de nombreuses symétries. En codant le prédicat d'égalité (ou la fonction de comparaison) de sorte que deux positions symétriques soient égales, l'algorithme de résolution aura beaucoup moins de positions à calculer. Codez les modifications nécessaires. Pour le jeu de Nim, c'est pareil, on peut donc ordonner les tas par ordre décroissant et supprimer les tas vides pour se ramener à une position canonique.
5. Les jeux de *misère* sont les jeux pour lesquels le dernier joueur à effectuer un mouvement a perdu. Pour le jeu de Nim, celui prenant le dernier jeton perd. Modifier votre code pour pouvoir choisir entre jouer à la version normale ou à la version misère.

Remarques

— La description des types et de la classe `Game` ne doit pas être contraignante : elle sert pour vous donner un intuition assez concrète des objectifs du projet. Vous êtes invités à la suivre, mais pouvez prendre des libertés.
— Pour intégrer votre code avec d'autres bibliothèques Haskell, vous devez utiliser l'outil `cabal`. Si cet outil n'est pas installé dans votre machine, vous pouvez le télé-charger depuis la page
`http://www.haskell.org/cabal/download.html`
— Votre code doit être propre et justement commenté (ni trop ni trop peu) et testé autant que possible. Vous utiliserez les idiomes fonctionnels autant que possible (et vous éviterez donc de faire du C déguisé en Haskell). La notation prend ces critères en compte !
— Le projet se fait impérativement en binôme.
— **Rendu du projet.** La date envisagée pour rendre le projet est le **dimanche 16 novembre, 23h59 dernier délai**. Envoyez votre projet par courriel (vos sources seront annexes en forme d'un archive `zip` ou `tgz`) à votre encadrant de TP. Si retard, des points de pénalités seront appliqués à la notation.
— Dates envisagés pour la soutenance : la semaine du 17/11 au 21/11.