Nom :

Prénom :

RICM3 - 2015/2016

Algorithmique et Programmation Fonctionnelle

Devoir à la Maison
À rendre pour le 11/12/2015

L'objectif de ce devoir est de concevoir un générateur (simple) d'histoires, et surtout une hiérarchie de types permettant de représenter et de générer les différents éléments qui composent cette histoire.

Cette situation est librement inspirée d'un exercice du MOOC OCaml de la plateforme FUN.

Les réponses devront être données sur un fichier .ml envoyé à votre enseignant de TP.

1 La situation

Nos histoires vont concerner un certain nombre de personnages ; chaque personnage est identifié par son *nom*, et dispose d'une certaine *fortune* (représentée par un nombre entier, on ne se préoccupera pas ici des centimes) :

type *nom* = *Tinkywinky* | *Laalaa* | *Po* | *Dipsy*
type *fortune* = *int*

On peut bien entendu étendre le type *nom* à volonté pour enrichir les histoires.
À tout moment, on connaîtra deux autres informations à propos de chaque personnage : son *état* (un seul à la fois parmi : bonheur, faim ou sommeil), et le *lieu* où il se trouve (voir plus bas les différents lieux).
Enfin, les personnages pourront réaliser certaines *actions*, en fonction de leur état et de l'endroit où ils se trouvent. Là encore, les actions possibles sont décrites dans le paragraphe ci-dessous.

Actions et lieux La liste ci-dessous décrit toutes les actions possibles dans les différents lieux et leurs conséquences. En aucune autre circonstance les actions ne sont autorisées.
- Les personnages peuvent manger au restaurant à condition d'avoir faim. Un repas coûte 20€ et rend heureux.
- Chaque personnage dispose d'une maison, dans laquelle il peut dormir gratuitement s'il a sommeil (mais seulement lui, pas les autres personnages).
- On peut également dormir à l'hôtel, pour la somme de 50€. Lorsqu'on dort, on n'a plus sommeil mais ça donne faim !
- Enfin, les personnages qui n'ont pas sommeil peuvent passer entre 1 et 10 heures à travailler à l'usine. Travailler rapporte 10€ de l'heure, mais à partir de 5 heures de travail les personnages sont fatigués.

Bien sûr quand une action coûte de l'argent, il faut en avoir suffisamment pour la réaliser.

Enfin, une action permet aux personnages d'aller dans n'importe quel lieu (sauf celui où ils se trouvent déjà bien sûr), sans condition ni conséquence particulière.

2 Modélisation

1. Proposer des types permettant de représenter respectivement :
   – les différents états des personnages
   – les différents lieux
   – un personnage avec toutes les informations qui lui sont associées
   – les différentes actions

Corrigé

```
type etat = Bonheur | Faim | Sommeil
type lieu = Maison of nom | Resto | Hotel | Usine
type personnage = nom × etat × fortune × lieu
type action = Aller of lieu | Manger | Dormir | Travailler of int
type personnages = personnage list
```

De plus, à certains moments, on manipulera une *liste* de personnages qui représente la situation courante, c'est-à-dire l'ensemble des protagonistes, leurs états, leurs fortunes et leurs lieux.

2. Définir en OCaml les constantes suivantes :
   (a) *tw* : un personnage fatigué nommé Tinkywinky, qui possède 10€ et se trouve chez Po
   (b) *dp* : un personnage heureux nommée Dipsy, qui possède 100€ et est à l'usine
   (c) *la* : un personnage nommé Laalaa, qui est au restaurant et a faim mais n'a pas d'argent

Corrigé

```
let tw = (Tinkywinky, Sommeil, 10, Maison(Po))
let dp = (Dipsy, Bonheur, 100, Usine)
let la = (Laalaa, Faim, 0, Resto)
```

3. Écrire une fonction *happy_end* qui prend une liste de personnages en paramètre et renvoie *true* s'ils sont tous heureux.

Corrigé

```
let rec happy_end lp = match lp with
    [] → true
  | (_, Bonheur, _, _) :: lps → happy_end lps
  | _ → false
```

3 Analyse d'une situation

1. Écrire une fonction *invite_resto* qui prend une liste de personnages en paramètre et renvoie la liste des personnages qui ont faim.

Corrigé

```
let rec invite_resto lp = match lp with
    [] → []
  | (n, Faim, _, _) :: lps → n :: (invite_resto lps)
  | _ :: lps → invite_resto lps
```

2. Écrire une fonction *rencontre* qui prend en paramètre une liste de personnages et qui renvoie *true* si au moins deux personnages de la liste sont dans le même lieu.

Corrigé

```
let rec rencontre lp =
  let rec occupe l lp' = match lp' with
      [] → false
    | (_, _, _, l') :: lps' → l = l' ∨ occupe l lps'
  in
  match lp with
    [] → false
  | (_, _, _, l) :: lps → occupe l lps ∨ rencontre lps
```

3. Écrire une fonction *rencontre2* similaire, mais qui renvoie *un des lieux* dans lequel au moins deux personnages sont présents. Vous ferez bon usage du type *'a option* pour traiter le cas où aucune rencontre n'a lieu.

Corrigé

```
let rec rencontre2 lp =
  let rec occupe l lp' = match lp' with
      [] → false
    | (_, _, _, l') :: lps' → l = l' ∨ occupe l lps'
  in
  match lp with
    [] → None
  | (_, _, _, l) :: lps → if occupe l lps then Some l else rencontre2 lps
```

4 Déroulement de l'histoire

1. Définir une fonction *action_possible* qui prend en paramètre une action et un personnage, et renvoie *true* si ce personnage peut effectuer cette action (voir plus haut dans quelles circonstances les actions sont possibles).

Corrigé

```
let action_possible a (n, e, f, l) = match a with
    Aller(l') → l ≠ l'
  | Manger → e = Faim ∧ l = Resto ∧ f ≥ 20
  | Dormir → e = Sommeil ∧ (l = Maison(n) ∨ l = Hotel ∧ f ≥ 50)
  | Travailler(h) → h > 0 ∧ h ≤ 10 ∧ l = Usine ∧ e ≠ Sommeil
```

2. Définir une fonction *consequence* qui prend en paramètre une action et un personnage qui peut effectuer cette action, et renvoie le personnage modifié par l'action.

Corrigé

```
let consequence a (n, e, f, l) = match a with
    Aller(l') → (n, e, f, l')
  | Manger → (n, Bonheur, f − 20, l)
  | Dormir → (n, Faim, (if l = Hotel then f − 50 else f), l)
  | Travailler(h) → (n, (if h < 5 then e else Sommeil), f + 10 × h, l)
```

3. On définit un *événement* comme un couple (action, nom). Écrire une fonction *chapitre* qui prend en paramètre un événement et une liste de personnages, qui cherche le personnage concerné dans la liste, vérifie s'il peut effectuer l'action, et renvoie la liste avec le personnage modifié le cas échéant.

Corrigé

```
let rec chapitre (a, n) lp = match lp with
    [] → []
  | ((n', _, _, _) as p) :: lps when n = n' →
 if (action_possible a p) then (consequence a p) :: lps else p :: lps
  | p :: lps → p :: (chapitre (a, n) lps)
```