NFP 119 : Programmation Fonctionnelle

Session du 27 juin 2011

Il vous est recommandé de rédiger de façon concise et précise, tout en écrivant lisiblement. Il sera tenu compte de la présentation dans la note de la copie. Tous les documents sont autorisés. Le barème n'est donné qu'à titre indicatif

Exercice 1 (7.5 points):

Dans cet exercice nous modélisons la trace des *performances* d'un joueur lors d'une *partie* de jeu vidéo. Un jeu vidéo est composé de plusieurs niveaux, où chaque *niveau* correspond à une suite de *quêtes* à réussir. Chaque quête possède un nombre *minimal de points* à acquerir pour être réussie, et une *performance*, correspondant au nombre de points réellement acquis par le joueur lors de la partie. Le nombre de points acquis dans un niveau est la somme des points de performance réalisés dans les quêtes de ce niveau ; et un niveau est réussi, si toutes ses quêtes sont réussies. On utilise les types suivants pour ces représentations :

```
type quete = {id: string; min: int; perf: int}

and niveau = {num: int; traceQ: quete list}
and tracePartie = {pseudo: string; traceN: niveau list}
```

1. Définissez une trace de partie avec deux niveaux. Le niveau 1 possède une seule quête : "tuer-monstre", avec un minimum de 100 points, et 170 de performance. Le niveau 2 possède deux quêtes : "trouver-porte", avec 50 points minimum et 70 de performance, et "echaper-aux-zombies", avec minimum de 200 points et seument 50 de performance. Utilisez des variables intermédiares q_1, ..., n_1, ..., pour définir les quêtes, niveaux, etc.
2. Ecrivez une fonction `queteReussie` qui renvoie true si une quête est réussie.
3. Ecrivez une fonction `pointsLQ` qui prend en paramètre une liste de quêtes, et qui renvoie le cumul de leurs points de performance. Utilisez cette fonction pour écrire la fonction `pointNiveau` qui renvoie le nombre de points cumulés d'un niveau.
4. Ecrivez une fonction `afficheNiveau` qui prend un niveau et qui affiche son numéro, ainsi que la trace de toutes ses quêtes. *Indice : utilisez une fonction auxiliaire récursive locale qui parcourt une liste de quêtes en les affichant.*
5. La fonctionnelle `List.for_all: ('a → bool) → 'a list → bool)` teste si tous les éléments d'une liste satisfont la condition passée en premier argument sous forme de fonction. Par exemple, l'appel suivant retourne false car 9 n'est pas paire :

   ```
   # List.for_all (fun x -> (x mod 2 = 0)) [6; 9; 18; 4];;
   - : bool = false
   ```

   A l'aide de cette fonctionnelle et de la fonction `queteReussie` de la question 2, écrivez une fonction `niveauReussi` qui prend en paramètre un niveau et qui renvoie true si le niveau est réussi. Rappel : un niveau est réussi si toutes ses quêtes sont réussies.

6. On souhaite calculer le bilan d'une partie sous forme d'un enregistrement de type `bilan`, avec un champ qui identifie la partie, et un champ `resultats` contenant la liste de résultats par niveau. Le résultat d'un niveau est modélisé par le type `resultatN` donné plus bas. Il enregistre le numéro du niveau, si le niveau a ou non été réussit et le nombre de points de performance cumulés

```
type resultatN = {niv: int; reussi: bool; points: int}
type bilan = {idPartie: string; resultats : resultatN list}
```

Ecrivez une fonction `bilanP` qui prend en argument une trace de partie et qui renvoie en résultat son bilan sous forme d'enregistrement de type `bilan`. *Indice : utilisez une fonction locale afin de calculer l'enregistrement de type `resultatN` correspondant à un niveau. Cette fonction pourra employer les fonctions `pointsNiveau` et `niveauReussi`. Utilisez ensuite List.map afin d'appliquer cette fonction locale sur tous les niveaux d'une partie.*

□

Exercice 2 (4.5 points)**:**

Dans cet exercice nous modélisons la gestion de dépendances lors de l'installation de nouveaux logiciels sur un ordinateur. Un logiciel ou *paquet* est décrit par son nom, son *numéro de version*, et par ses *dépendances*, qui expriment quels autres logiciels, et dans quels versions, doivent être installés au préalable pour le bon fonctionnement du logiciel à installer. Les dépendances sont des combinaisons par *et* et par *ou* sur des contraintes de base de la forme : *le logiciel x dans une version supérieure ou égale à n*, ou encore aucune dépendance. Par exemple, voici ce que l'on trouve dans un fichier qui décrit le paquet *tuareg* version 1.8 :

```
Name: tuareg
Version: 1.8
Depends: ocaml (>= 3.1) & (emacs (>= 20.5.1) | xemacs (>= 21.3))
```

Ce paquet a besoin d'*ocaml* au moins en version 3.1, et soit d'*emacs* au moins en 20.5.1, soit d'*xemacs* au moins en 21.3. Les *numéros de version* seront modélisés par une liste d'entiers non négatifs. Par exemple, 3.1.0 sera modélisé par [3 ;1 ;0]. Nous utilisons les types suivants pour modéliser les paquets à installer :

```
type version = int list
and dependance = Vide
               | SupEgal of string * version
               | Et of dependance * dependance
               | Ou of dependance * dependance
and paquet = {nom: string; vers: version; dep: dependance}
```

Exemple : on peut répresenter la contrainte *ocaml au moins en version 3.1* par :

```
# let dep_ocaml = SupEgal("ocaml",[3;1]);;
val dep_ocaml : dependance = SupEgal ("ocaml", [3; 1])
```

Enfin, on utilisera une *liste d'association* composé de couples *(nom, version)*, afin de représenter la liste de paquest déjà installés sur un ordinateur. Par exemple, selon la liste suivante *ocaml* 3.3 et *xemacs* 21.3.2 sont déjà installés.

```
let i2 = [("ocaml",[3;3]); ("xemacs",[21;3;2])];;
```

Les deux fonctions suivantes pré-définies sur les listes d'association vous seront utiles :

- `List.mem_assoc a l`, teste si un couple (a,b), est présent dans l.
- `List.assoc a l`, renvoie la deuxième composante b du premier couple (a,b) présent dans l.

1. Donnez la définition complète du paquet *tuareg 1.8* décrit précédemment.Utilisez des variables intermédiaires pour faciliter la lecture de votre définition.
2. On utilise l'ordre lexicographique pour comparer deux numéros de version. Par exemple, 3.1.3 est plus petit que 3.2. Ces numéros étant modélisés par des listes, on procèdera de la manière suivante. Une liste vide est toujours plus petite qu'une liste non vide. Sinon, on compare les deux premiers éléments des deux séquences. S'ils sont égaux on continue avec les deux restes. Sinon, la séquence qui commence par l'entier le plus petit est la plus petite. Ecrivez la fonction `infEgalVersions` qui prend deux numéros de version et qui renvoie true si le premier est plus petit ou égal au deuxième.
3. Ecrivez une fonction `installPossible` qui prend une un paquet p et une liste d'association de paquets installés li et répond true si les dépendances de p sont satisfaites par les paquets dans li.

□

Exercice 3 (2 points)**:**
Soit `somme` la fonction suivante :

```
let rec somme x =
  if x <= 0 then 0 else x + somme (x-1) ;;
```

1. (2pt) Démontrez la propriété suivante : $\forall x \in \mathbb{N}$, `somme` $x = (\sum_{i=0}^{x} i)$, c'est-à-dire que si `x` est positif `somme x` retourne la somme des entiers compris entre 0 et `x`.

□

Exercice 4 (3 points)**:**
La fonction suivante `somme2` utilise un argument supplémentaire comme « accumulateur » :

```
let rec somme2 x res =
  if x <= 0 then res
  else somme2 (x-1) (x+res);;
```

1. (2,5pt) Démontrez la propriété suivante : $\forall x \in \mathbb{N}, \forall res$ `somme2` $x\ res = (\sum_{i=0}^{x} i) + res$
2. (0,5pt) En déduire que $\forall$ `x`$\in \mathbb{N}$, `somme2 x 0` = $(\sum_{i=0}^{\mathtt{x}} i)$

□

Exercice 5 (3 points)**:**

```
let rec f x l =
  if x = 0 then 0
  else
    match l with
     | [] -> x
     | e::l2 ->
          e
        + f x l2
        + f (x-1) (e::e::l2)
```

1. (1,5pt) Démontrez que cette fonction termine pour tout entier `x` positif et toute liste `l` d'entiers positifs.

2. (1,5pt) Démontrez que pour tout entier `x` positif et toute liste `l` d'entiers positifs, `f` $x\ l$ est supérieur à la somme des éléments de `l`. Autrement dit :

$$\forall x, \forall l, x \geq 0 \wedge (\forall n \in l, n \geq 0) \rightarrow \texttt{f}\ x\ l \geq \sum_{n \in l} n.$$

□