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.
   Réponse: 1.5 – 0.25 par variable

```
let q1 = {id="tuer-monstre"; min=100; perf=170};;
let q2 = {id="trouver-porte"; min=50; perf=70};;
let q3 = {id="echaper-zombies"; min=200; perf=50};;

let n1 = {num=1; traceQ=[q1]}
let n2 = {num=2; traceQ = [q2; q3]};;
let tp = {pseudo="cap. toto"; traceN = [n1;n2]};;
```

2. Ecrivez une fonction `queteReussie` qui renvoie true si une quête est réussie.
   Réponse: Réponse (0.5 pts)

```
let queteReussie {min=m; perf=p} = p>=m;;
```

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.
   Réponse: Réponse (1.5 pts)

```
let rec pointsLQ lq=
 match lq
 with [] -> 0
  | {perf=p}::r -> p + pointsLQ r;;

let pointsNiveau n =
 pointsLQ n.traceQ;;

# pointsNiveau n2;;
- : int = 120
```

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.*
Réponse: Réponse (1.5 pts)

```
let afficheNiveau n =
 let rec afficheQuetes lq=
 match lq
 with [] -> ()
  | {id=c; min=m; perf=p}::r ->
        print_string("␣␣␣"^c^"␣"); print_int p; print_string "␣/␣";
        print_int m; print_newline(); afficheQuetes r
 in print_string"Niveau:␣"; print_int(n.num);
   print_string"\n␣Quetes:␣\n";
   afficheQuetes n.traceQ;;

# afficheNiveau n2;;
Niveau: 2
 Quetes:
  trouver-porte 70 / 50
  echaper-zombies 50 / 200
- : unit = ()
```

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.
Réponse: Réponse (1 pts)

```
let niveauReussi n =
 List.for_all queteReussie n.traceQ;;

# niveauReussi n2;;
- : bool = false
```

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.*
Réponse: Réponse (1.5 pts)

```
let bilanP p =
  let resN n = {niv = n.num; reussi= niveauReussi n; points = pointsLQ n.traceQ}
  in {idPartie= p.pseudo ; resultats = List.map resN p.traceN};;

# bilanP tp;;
- : bilanPartie =
{idPartie = "cap. toto";
 resultats =
  [{niv = 1; reussi = true; points = 170};
   {niv = 2; reussi = false; points = 120}]}
```

□

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.
   Réponse: 1.5 pt

```
let dep_ocaml = SupEgal("ocaml",[3;1])
let dep_emacs = SupEgal("emacs",[20;5;1])
let dep_xemacs = SupEgal("xemacs",[21;3])

let tuareg = {nom="tuareg"; vers=[1;8];
  dep= Et (dep_ocaml, Ou ( dep_emacs, dep_xemacs))}
```

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.
   Réponse: 1.5 pt –¿ 0.5 par cas correct

```
let rec infEgalVersions v1 v2 =
 match (v1,v2)
 with [],_   -> true
  |  _::_,[] -> false
  |  n1::r1, n2::r2 ->
        if n1=n2 then infEgalVersions r1 r2
        else n1 < n2;;
```

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.
   Réponse: 1.5 pt –¿ 0.5 par cas correct (sauf pour Vide)

```
let installPossible p pInst =
 let rec iPos dep =
 match dep
 with Vide     -> true
  |   SupEgal(n,v) -> if not(List.mem_assoc n pInst) then false
                 else let v1 = List.assoc n pInst in
                     infEgalVersions v v1
  |   Et (d1, d2) -> iPos d1 && iPos d2
```

```
    |   Ou (d1, d2) -> iPos d1 || iPos d2
  in iPos p.dep;;

installPossible tuareg i;;

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

installPossible tuareg i2;;
```

□

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.

   Réponse: Par récurrence sur x.
   - *Base :* $x = 0$*, OK.*
   - *Réc : Supposons que pour un x quelconque* somme $x = (\sum_{i=0}^{x} i)$*. Montrons qu'alors* somme $(x+1) = (\sum_{i=0}^{x+1} i)$*. Par définition* somme $(x+1) = (x+1)+$ somme x *car* $x > 0$*. Or par hypothèse de récurrence* somme $x = (\sum_{i=0}^{x} i)$*. Donc* somme $(x+1) = x+1+(\sum_{i=0}^{x} i) = \sum_{i=0}^{x+1} i$*. OK.*

□

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 $x\ res = (\sum_{i=0}^{x} i) + res$

   Réponse: Par récurrence sur x.
   - *Base :* $x = 0$*, OK.*
   - *Réc : Supposons que pour un x quelconque* $\forall res$, somme2 $x\ res = (\sum_{i=0}^{x} i) + res$*. Montrons qu'alors* $\forall res$, somme2 $(x+1)\ res = (\sum_{i=0}^{x+1} i) + res$*. Soit res un entier positif quelconque, par définition* somme2 $(x+1)\ res =$ somme2 $x\ (x+1+res)$*. Or par hypothèse de récurrence* somme2 $x\ (x+1+res) = (\sum_{i=0}^{x} i) + (x+1+res) = (\sum_{i=0}^{x+1} i) + res$*. OK.*
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.

   Réponse: On utilise l'ordre suivant : $(x_1, l_1) < (x_2, l_2)$ *ssi* $\begin{cases} x_1 <_{\mathbb{N}} x_2 \\ x_1 = x_2 \text{ et } |l_1| <_{\mathbb{N}} |l_2| \end{cases}$

   Cet ordre est bien fondé car il s'agit de la combinaison lexicographique de deux ordres bien fondés (le deuxième par image inverse de la fonction $|.|$ *des listes vers* $\mathbb{N}$*). Les deux appels récursifs se font sur les couples* `(x,l2)` *et* `(x-1,e::e::l2)`*, or* `(x,l2)` $<$ `(x,l)` *car* `|l2|` $<_{\mathbb{N}}$ `|l|`*, et* `(x-1,e::e::l2)` $<_{\mathbb{N}}$ `(x,l)` *car* `(x-1)` $<_{\mathbb{N}}$ `x`*. Donc la fonction termine.*

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 :

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

   Réponse: Par récurrence bien fondée en utilisant l'ordre ci-dessus. Soit $x \in \mathbb{N}$ *et* l *une liste d'entiers positifs. Supposons que* $\forall y \in \mathbb{N}$ *t.q.* $y < x$ *et* $\forall l', |l'| < |l|$*,* `f` $y\ l' \geq \sum_{n \in l'} n$*. On distingue 3 cas :*

   (a) $x = 0$*, alors* $f\ x\ l = 0 \geq 0$*. OK.*

   (b) $x > 0$ *et* $l =$ `[]`*, alors* $f\ x$ `[]` $= x \geq 0$ *car* $x > 0$*. OK.*

   (c) $x > 0$ *et* $l = e$ `::` l_2*, alors* $f\ x\ l = e + f\ x\ l2\ +\ f\ (x-1)\ (e :: e :: l2)$.
   L'hypothèse de récurrence s'applique à $(x, l2)$ *car* $|l2| < |l|$*. Donc* $f\ x\ l2 \geq \sum_{n \in l2}$.
   L'hypothèse de récurrence s'applique également sur $((x-1), (e :: e :: l2))$ *car* $x - 1 < x$*. Donc on a : que* $f\ (x-1)\ (e :: e :: l2) \geq \sum_{n \in e::e::l2} n$*, donc* $f\ (x-1)\ (e :: e :: l2) \geq 0$*.*
   Donc finalement $e + f\ x\ l2\ +\ f\ (x-1)\ (e :: e :: l2) \geq e + \sum_{n \in l2} + 0 = \sum_{n \in e::l2} n = \sum_{n \in l} n$*. OK.*

□