TP2 - Itérateurs

(*UPS - L3 Informatique - UE TAPFA*)

Prélude

Ce TP s'appuie sur le "prélude" suivant, qui est automatiquement ajouté avant d'effectuer la notation de votre code, mais que vous pourriez devoir (*temporairement*) ajouter à la main à votre code, **si vous utilisez un *toplevel* en dehors de PFITAXEL**:

```
type elem = Obstacle | Objet | Rien
type plateau = (int * int) -> elem
type etat_pince = Vide | Pleine
```

Contexte

On considère un robot pouvant détecter la présence d'obstacles ou d'objets et se déplacer selon les 4 directions sur un plateau dont les cases contiennent soit un obstacle (case inaccessible au robot), soit un objet (que le robot peut prendre), soit rien (le robot peut poser un objet). On définit le type `elem` :

```
type elem = Obstacle | Objet | Rien
```

1) Propriétés

Les propriétés, définies par le type `prop`, et pouvant être testées par le robot sont : la case occupée contient un élément donné, la pince est vide, la conjonction de deux propriétés, la négation d'une propriété.

a) Définir le type `prop`.

b) Définir les fonctions de construction (ou constantes) suivantes :

```
contient : elem -> prop
pince_est_vide : prop
andp : prop -> prop -> prop
notp : prop -> prop
```

c) Exprimer par un terme `some_prop` de type `prop` la propriété suivante : « la case occupée par le robot contient un objet et la pince est vide ».

Note: la question (d) ci-dessous (et la partie (3) « Évaluation des propriétés ») ne pourra pas être traitée avant la partie du cours « itérateur le plus général ».

d) Définir l'itérateur `it_prop` « le plus général » sur le type `prop` et donner son type (on traitera les cas dans le même ordre que ci-dessus).

---

2) Plateau et état

Le contenu du plateau est défini par une fonction qui associe à chaque couple de coordonnées le contenu de la case associée :

```
type plateau = (int * int) -> elem
```

Par exemple, un plateau contenant un objet en position (0,0) et des cases vides par ailleurs sera défini par :

```
function
| 0,0 -> Objet
| _ -> Rien
```

L'état du système est composé du plateau, de la position du robot, de l'état de la pince (vide ou non) et du vecteur de direction du robot. Le vecteur de direction est défini par 2 entiers pris parmi {-1,0,1} codant les 4 directions (`(1, 0)` pour droite, `(0, 1)` pour haut, `(-1, 0)` pour gauche et `(0, -1)` pour bas). On supposera sans le vérifier, que les "vecteurs de direction" fournis seront parmi ces 4 possibilités.

a) Définir le type `etat`.

Ce type utilisateur est-il un type somme ? (justifiez votre réponse)

b) Définir la fonction de construction `mk_etat : plateau -> (int * int) -> etat_pince -> (int * int) -> etat` où les deux couples d'entiers sont respectivement la position du robot et son vecteur de direction.

c) Définir les fonctions suivantes, permettant d'extraire les informations précédentes d'un `etat` :

```
get_plateau : etat -> plateau
get_position : etat -> (int * int)
get_pince : etat -> etat_pince
get_direction : etat -> (int * int)
```

3) Évaluation des propriétés

Ecrire avec l'itérateur `it_prop` la fonction `eval` prenant en argument un `etat` et une propriété (de type `prop`) et renvoyant true si la propriété est vérifiée par l'état.

4) Actions du robot (partie 1)

a) Définir la fonction `prendreCase` prenant en argument un element (contenu d'une case) et retournant l'élément `Rien` si le contenu est un objet, ou utilisant `failwith` pour lever une exception sinon.

b) En déduire la fonction `prendreA` prenant en argument un `plateau`, une position `(x,y)` et retournant le plateau où l'objet supposé présent en `(x,y)` a été retiré (remplacé par `Rien`). Si la position `(x,y)` ne contient pas d'objet, une exception sera levée *au moment propice* (sans renvoyer de plateau).

Autrement dit, `prendreA (fun _ -> Rien) (0, 0)` (par exemple) devra lever une exception même s'il s'agit d'une application partielle de la fonction `prendreA : plateau -> int * int -> int * int -> elem` (qui prend 3 arguments).

c) Définir la fonction `etatPrendre : etat -> etat` retournant l'état obtenu après la prise dans la pince du robot de l'objet se trouvant dans la position courante. Une exception sera levée si l'exécution de la commande n'est pas possible.

5) Actions du robot (partie 2) (même chose !)

Similairement à la question 4 :

a) Définir la fonction `poserCase` prenant en argument un element (contenu d'une case) et retournant un `Objet` si la case était vide, ou utilisant `failwith` pour lever une exception sinon.

b) En déduire la fonction `poserA` prenant en argument un `plateau`, une position `(x,y)` et retournant le plateau où la case `(x,y)` supposée vide a été remplacée par `Objet`. Si la position `(x,y)` n'était pas vide, une exception sera levée *au moment propice* (sans renvoyer de plateau).

c) Définir la fonction `etatPoser : etat -> etat` retournant l'état obtenu après la dépose par le robot de l'objet à la position courante. Une exception sera levée si l'exécution de la commande n'est pas possible.

6) Actions du robot (partie 3) (avancer/tourner)

a) Définir la fonction `etatTourner` retournant l'état obtenu en faisant tourner le robot d'un quart-de-tour vers la gauche (sens antihoraire). Rappel :

> le vecteur de direction est défini par 2 entiers pris parmi {-1,0,1} codant les 4 directions (`(1, 0)` pour droite, `(0, 1)` pour haut, `(-1, 0)` pour gauche et `(0, -1)` pour bas). On supposera sans le vérifier, que les "vecteurs de direction" fournis seront parmi ces 4 possibilités.

b) Définir la fonction `etatAvancer` retournant l'état obtenu en faisant avancer le robot du déplacement indiqué par le vecteur de direction se trouvant dans son état. Une exception sera levée si l'avancée n'est pas possible (la prochaine case contient un obstacle). Le plateau est supposé non borné.

7) Commandes

Les commandes pouvant être exécutées par le robot sont : Prendre, Poser, Tourner, Avancer, Exécuter une commande si une condition est vérifiée, Exécuter une liste de commandes.

a) Définir le type `commande`.

b) Définir les fonctions de construction (ou constantes) suivantes :

```
prendre : commande
poser : commande
tourner : commande
avancer : commande
if_prop : commande -> prop -> commande
```

```
seq : commande list -> commande
```

c) Définir la fonction `executer` prenant en argument un `etat` et une `commande`, et retournant l'état obtenu après exécution de la commande. Vous écrirez deux versions qui différeront de l'implémentation du cas "liste de commandes" :

- l'une (fonction `executer`) écrite directement avec un filtrage sur les listes ;
- l'autre (fonction `executer2`) écrite en utilisant la fonction `List.fold_left`.