Université de Bordeaux
Année 2016-2017

Mastère 2
T.A.P

TD n°7 - Programmation orientée par les données
Représentation des données par des fonctions

Exercice 1: Type de données pour graphes

Pour représenter les graphes à l'aide d'un type de données, partons de la représentation liant chaque sommet à ses voisins, en utilisant des *listes d'adjacence*. Considérons ainsi le type suivant :

```
type 'a vertex = V of 'a
and 'a graph_full = Graph of ('a vertex * 'a vertex list) list;;
```

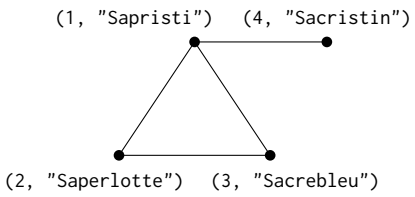
1. Créer un graphe dans lequel le type `'a` est une chaîne de caractères. Comment doit-on tester de l'égalité de deux sommets ?
2. Comment pourrait-on écrire une fonction `graph_map` modifiant juste les valeurs sur les sommets du graphe (et surtout pas sa structure) ? Quel problème se pose ici ?

Pour tenter de remédier à ce problème, on considère maintenant le type suivant :

```
type 'k key = K of 'k
and ('k, 'a) vertex = 'k key * 'a
and ('k, 'a) graph = Graph of (('k, 'a) vertex * 'k key list) list;;
```

Ainsi, un exemple de graphe simple sur lequel les étiquettes sont des chaînes de caractères correspondrait à :

```
Graph ([
  ((K 1, "Sapristi"), [K 2;K 3;K 4]);
  ((K 2, "Saperlotte"), [K 1;K 3]);
  ((K 3, "Sacrebleu"), [K 1;K 2]);
  ((K 4, "Sacristin"), [K 1]);
      ])
```



3. Quelle est la différence avec le type précédent ?

4. Écrire la fonction `graph_nexts` qui, étant donné un graphe et un sommet, fournit le liste de ses voisins.

5. Quel serait le type d'une fonction `graph_map` agissant sur les données de type `graph` ?

6. Écrire une fonction `component` permettant, à partir d'un sommet donné, d'obtenir l'ensemble des sommets dans la même composante connexe.

Exercice 2: Méthode du gradient

Reprenons la définition de type sur les graphes vue précédemment :

```
type 'k key = K of 'k
and ('k, 'a) vertex = 'k key * 'a
and ('k, 'a) graph = Graph of (('k, 'a) vertex * 'k key list) list;;
```

1. Écrire une fonction énumérant les sommets d'un `graph` par un parcours en profondeur.

2. Écrire une fonction énumérant les sommets d'un `graph` selon un parcours en largeur.

Les deux fonctions précédentes ont en pratique beaucoup de code en commun. On se propose de factoriser les parcours sur les graphes en utilisant une *stratégie*, sous la forme d'une fonction décidant, étant donné le sommet courant, la liste de ses voisins et les sommets restant à parcourir, la façon dont ces voisins vont s'adjoindre à la liste restant à parcourir.

```
type 'a strategy = Strategy of ('a → 'a list → 'a list → 'a list)
```

3. Écrire un parcours de graphe qui soit générique.

Pour illustrer ces parcours de graphe, on s'intéresse à une méthode apparentée à la *méthode du gradient*, qui consiste à rechercher un élément localement minimal dans un graphe.

4. Écrire une fonction `until_reached` qui applique une stratégie `s` lors du parcours d'un graphe, jusqu'à ce qu'une fonction d'arrêt (dépendant uniquement du sommet et du graphe) décide d'arrêter le parcours.

5. Écrire la fonction d'arrêt permettant de s'arrêter lors du parcours d'un graphe lorsque l'élément courant est plus petit que tous ses voisins.

Exercice 3: Représentation de données par des fonctions

Un ensemble peut toujours être représenté par sa *fonction caractéristique*, une fonction qui prend en argument un élément et renvoie vrai si cet élément appartient à l'ensemble, et faux sinon. Concrètement, considérons le type :

```
type 'a set = Set of ('a → bool)
```

Le type `'a set` est un type inductif avec un unique constructeur, ce qui fait qu'on peut l'utiliser de la manière suivante :

```
let set_even = Set (fun x → (x mod 2) == 0);;
let set_contains_0 (Set s) = s 0;;  (* pattern-matching *)
```

Pour travailler sur ce type, nous allons mettre en place l'outillage habituel de la théorie des ensembles :

1. Écrire l'ensemble vide (`set_empty`) et l'ensemble plein (`set_all`). Écrire une fonction qui construit à partir d'un élément x l'ensemble correspondant au singleton $\{x\}$.
2. Écrire la fonction `set_mem` qui teste l'appartenance d'un élément à un ensemble.
3. Écrire la fonction `set_add` qui permet d'ajouter un élément à un ensemble, et la fonction `set_out` qui permette d'obtenir le complémentaire d'un ensemble.

Les opérateurs suivants permettent de faire respectivement l'union et l'intersection de deux ensembles :

```
let ( +| ) (Set s1) (Set s2) = Set (fun y → s1 y || s2 y);;
let ( *| ) (Set s1) (Set s2) = Set (fun y → s1 y && s2 y);;
```

4. Dans le même ordre d'idées, écrire la fonction `(+-|)` qui permet de faire la différence symétrique de deux ensembles E_1 et E_2, c'est-à-dire l'ensemble des éléments qui sont soit dans E_1, soit dans E_2.

$$E_1 \Delta E_2 = \left(E_1 \cap (E_2)^c\right) \cup \left((E_1)^c \cap E_2\right)$$

   On écrira cette fonction de deux manières différentes : premièrement, avec la définition mathématique ci-dessus, et deuxièmement avec une fonction `xor`.

Exercice 4: Ensembles de points du plan

Supposons vouloir représenter avec le type précédent des ensembles de points du plan. Pour ce faire, relancer l'interpréteur avec la commande `ocaml graphics.cma` et utiliser la représentation suivante afin de pouvoir afficher les ensembles utilisés :

```
type point2D = Point2D of float * float;;
```

Pour commencer, il faut récupérer dans les sources la fonction `set_draw` qui permet de dessiner un ensemble de points du plan dans une fenêtre graphique.

1. Écrire une fonction `disk` permettant de représenter un disque dans le plan.

2. Écrire des fonctions `halfplane` permettant de représenter des demi-plans horizontaux et verticaux. Comment représenter un demi-disque ?

Le code source fourni contient un certain nombre de fonctions de transformation géométrique simple pour un `Point2D`, comme des homothéties, translations et rotations.

3. Écrire des fonctions `set_scale`, `set_translate` et `set_rotate` permettant d'appliquer respectivement une homothétie, une translation et une rotation à un `point2D set`.

Considérons maintenant le code de la fonction `wheel` fourni avec les sources.

4. Expliquer ce que fait la fonction `wheel`, et vérifiez son fonctionnement avec les deux fonctions `( +-| )` définies précédemment.

   Pourquoi y en a t'il une plus rapide que l'autre ?

Exercice 5: Introduction à la congélation

Nous étudions ici quelques aspects de la *congélation*. Une manière d'implémenter cette technique consiste à préfixer les expressions par un niveau de fonctionnelle `fun () → ...` de manière à ne pas évaluer directement l'expression. Celle-ci sera évaluée lors de l'opération de *dégel*, qui consiste à forcer l'évaluation de ce niveau de fonctionnelle. La congélation permet ainsi de maîtriser l'exécution des calculs.

Considérons le type suivant pour les valeurs :

```
type 'a frozen_flow =
| End of 'a
| Step of (unit → 'a frozen_flow);;
```

Une valeur est donc soit calculée, soit congelée et en attente de décongélation.

1. Écrire la fonction `thaw :'a frozen flow → 'a frozen flow` qui permet de décongeler une étape de calcul (si elle est congelée).

Le code suivant permet de calculer de manière réursive terminale le ppcm de deux nombres entiers positifs :

```
let ppcm x y =
  let rec ppcm_rec x y mul =
    if (y > x) then (ppcm_rec y x mul) else (* Ensures x >= y *)
    if (x = 0) then 0 else                  (* Ensures both are positive *)
      let r = (x mod y) in
        if (r = 0) then (mul/y) else ppcm_rec y r mul
  in ppcm_rec x y (x*y);;
```

2. Réécrire cette fonction en utilisant le type `frozen_flow` pour effectuer ce calcul en congelant *chaque* appel récursif.

   Indice : son type doit devenir `int → int → int frozen_flow`.

Enrichissons le type `frozen_flow` pour permettre de manipuler en même temps que le flot d'exécution des données présentes dans l'environnement sous fome d'une liste d'associations (associant un nom de variable et une valeur, dans le cas présent quelque chose comme `[("f",6);("n",4)]`)

```
type ('key,'data) frozen_data_flow =
| End of 'data
| Step of (unit → (('key*'data) list) * ('key,'data) frozen_data_flow);;
```

3. Réécrire la fonction précédente en utilisant ce nouveau type, de façon à obtenir une forme primitive de débogueur.