TD n°6 - Types inductifs
Programmation orientée par les données

Exercice 1: Files d'attentes fonctionnelles
Considérons l'implémentation suivante de files d'attente fonctionnelles[1]. Un type associé à une file d'attente est défini par :

```
type 'a queue = Q of 'a list * 'a list;;
```

Une *file d'attente* est une structure de donnée dans laquelle on peut insérer des éléments, et les faire sortir dans l'ordre dans lequel ils ont été insérés. L'implémentation utilise deux listes : une liste d'entrée dans laquelle on insère les éléments et une liste de sortie à partir de laquelle on les sort. Lorsqu'il n'y a plus d'éléments dans la liste de sortie, on remplace la liste de sortie par la liste d'entrée *retournée*. Par exemple :

État de la file	Liste d'entrée	Liste de sortie
File vide	`[]`	`[]`
Insertion de 1	`[1]`	`[]`
Insertion de 2	`[2;1]`	`[]`
Retournement	`[]`	`[1;2]`
Ejection	`[]`	`[2]`
Insertion de 3	`[3]`	`[2]`

Idéalement, l'opération de retournement est effectuée uniquement lorsque l'on doit éjecter un élément, et que la liste de sortie est vide. Les opérations d'insertion et d'éjection sont naturellement en temps constant.

1. Écrire une fonction `q_insert` qui insère un élément dans une file d'attente fonctionnelle.

   Remarque : vu la façon dont les files d'attente sont définies, on peut écrire ces fonctions en utilisant la reconnaissance de motifs de la manière suivante :

```
let q_fun  (Q (l1, l2)) = l1@l2
```

[1] Cf. *"An efficient functional implementation of FIFO queues"* de F. W. Burton (1982).

2. Écrire une fonction `q_transfer` qui effectue l'opération de retournement, c'est-à-dire retourne la première liste dans la seconde, et ceci uniquement lorsque la seconde liste est vide.

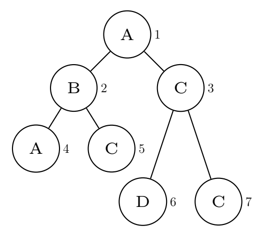
3. Écrire la fonction `q_pop` qui éjecte un élément de la file d'attente. La fonction générera une exception avec la commande `failwith` si la liste est vide.

Exercice 2: Méthode du gradient

Considérons la définition de type suivante sur les graphes, utilisant une représentation par liste d'adjacence :

```
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;;
```

```
Graph ([
        ((K 1, "A"), [K 2; K 3]);
        ((K 2, "B"), [K 4; K 5]);
        ((K 3, "C"), [K 6; K 7]);
        ((K 4, "A"), []);
        ((K 5, "C"), []);
        ((K 6, "D"), []);
        ((K 7, "C"), []);
      ])
```



1. Écrire une fonction qui, étant donné un graphe et une clé, renvoie la liste des clés des voisins.

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

3. 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)
```

4. 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.

5. 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.

6. 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.

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

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

Exercice 4: Interprète BASIC
Une utilisation habituelle des types inductifs consiste à représenter des grammaires, qui à leur tour permettent d'écrire des compilateurs. Sur la page http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/book-ora059.html se trouve un exemple d'interpréteur du langage BASIC.

Pour nous simplifier la vie, nous allons alléger la grammaire utilisée. Dans un premier temps, les expressions du langage seront de la forme :

```
type op_bin = PLUS | MOINS | EQUAL | MULT | DIV;;
type expression =
| ExpInt of int
| ExpVar of string
| ExpBin of expression * op_bin * expression  ;;
```

On ne gère que des expressions numériques ou booléennes. Les valeurs gérées par l'environnement seront du type :

```
type valeur = Vint of int | VBool of bool;;
```

1. Comment gérer l'environnement dans lequel sont évaluées les expressions ?

2. Écrire une fonction qui prend en argument une expression et un environnement et qui renvoie la valeur correspondant à cette expression (en remplaçant les variables par les valeurs qu'elles possèdent dans l'environnement).

La syntaxe des expressions du langage se résume à :

```
type instruction =
| Goto of int
| Print of expression
| Input of string
| If of expression * int
| Let of string * expression  ;;
type ligne = { num : int ; inst : instruction }  ;;
type program = ligne list  ;;
```

Nous allons tenter d'écrire un mini-interpréteur de ce langage. Pour cela, nous allons utiliser les définitions suivantes pour les valeurs et l'environnement. Remarquer que les instructions conditionnelles IF sont obligatoirement suivies d'un GOTO. De plus, les lignes sont numérotées de 10 en 10.

3. Écrire une fonction qui prend en argument un programme, une ligne et un environnement, et renvoie le numéro de ligne correspondant à l'instruction suivante, ainsi que l'environnement modifié par le traitement de cette instruction.

Voilà un petit programme BASIC qui calcule la factorielle d'un entier :

```
10 INPUT N
20 LET I = 1
30 LET S = 1
40 LET I = I + 1
50 LET S = S * I
60 IF (I = N) THEN GOTO 80
70 GOTO 40
80 PRINT S

let prog = [
  { num = 10; inst = Input "n" };
  { num = 20; inst = Let ("i",ExpInt 1) };
  { num = 30; inst = Let ("s",ExpInt 1) };
  { num = 40; inst = Let ("i",(ExpBin ((ExpVar "i"),PLUS,(ExpInt 1)))) };
  { num = 50; inst = Let ("s",(ExpBin ((ExpVar "s"),MULT,(ExpVar "i")))) };
  { num = 60; inst = If ((ExpBin ((ExpVar "i"),EQUAL,(ExpVar "n"))),80) };
  { num = 70; inst = Goto 40 };
  { num = 80; inst = Print (ExpVar "s") };
];;
```

4. Écrire une fonction permettant de simuler le fonctionnement d'un programme et l'appliquer à ce programme.

Les types inductifs du langage OCaml permettent de faciliter l'écriture de cet interpréteur. La question se pose alors de savoir quels problèmes se posent dans un langage différent.

5. Reprendre les 4 premières questions en langage C#, en adaptant les types utilisés pour qu'ils correspondent au langage.

6. Quelles sont les avantages et inconvénients de la programmation utilisant des types inductifs en C# par rapport à OCaml ?