Devoir en temps libre n° 1

Ce devoir est à rendre pour le 6 mars 2017. La partie I est à rédiger sur papier et à rendre au début du cours. Le code CAML de la partie II devra être envoyé à `nicolas.pecheux@cpge.info` avant 07h59 le 6 mars.

I Exercices CAML à faire sur papier

Les exercices de cette partie sont à rédiger proprement, en couleur, et en mettant en valeur l'indentation. Comme dans toute question qui demande d'implémenter une fonction, vous veillerez à introduire votre programme par quelques lignes explicatives, à préciser les types de vos fonctions et à indiquer la complexité de votre programme (en utilisant la notation $O(\cdot)$). Le premier exercice est résolu pour montrer ce qui est attendu.

EXERCICE 1 *Exercice corrigé*

Écrire une fonction `avant_dernier` qui donne l'avant-dernier élément d'une liste s'il existe et qui lève l'exception `Failure "Liste trop courte"` sinon.

Solution. Si la liste contient 0 ou 1 élément, alors il n'y a pas d'avant-dernier et on lève l'exception demandée (deux premiers cas du filtrage). Si la liste contient exactement deux éléments, on renvoie directement le premier, qui est bien l'avant-dernier (troisième cas du filtrage). Sinon, l'avant-dernier élément de la liste est le même que l'avant-dernier de sa queue, ce qui justifie l'appel récursif du quatrième et dernier cas.

```
let rec avant_dernier liste =
  match liste with
   | [] -> failwith "Liste trop courte"
   | _ :: [] -> failwith "Liste trop courte"
   | tete :: _ :: [] -> tete
   | _ :: queue -> avant_dernier queue
;;
```

Cette fonction est de type `'a list -> 'a`. Pour une liste de moins de deux éléments, l'appel termine par l'un des cas de base, en temps constant $O(1)$. Sinon, un appel récursif est effectué sur la queue de la liste. Il y a donc au plus n appels récursifs où n est la taille de la liste. Ainsi, cette fonction termine toujours et est de complexité $O(n)$, linéaire en la taille de la liste. □

EXERCICE 2 *Existence d'un élément satisfaisant à un prédicat*

Écrire une fonction `existe`, telle que `existe predicat liste` renvoie vrai s'il existe un élément d'une liste satisfaisant un prédicat de type `'a -> bool` et faux sinon. Pour le calcul de complexité, on supposera que la complexité du prédicat est en $\Theta(1)$.

EXERCICE 3 *Minimum et maximum d'une liste*

Écrire une fonction `min_et_max` prenant en entrée une liste et renvoyant le couple formé par le plus petit et le plus grand élément, ou levant l'exception `Failure "Liste vide"` si la liste est vide.

EXERCICE 4 *Supprimer l'élément d'indice n d'une liste*

Écrire une fonction `supprime`, telle que l'appel `supprime n liste` renvoie une liste sans l'élément d'indice n s'il y en a un et renvoie une liste identique sinon.

EXERCICE 5 *X 2010*

Écrire une fonction `log2` prenant en argument un entier $n \in \mathbb{N}^*$, et qui calcule le plus grand entier k tel que $2^k \leq n$.

II Exercices CAML à faire sur machine

Les programmes de cette partie devront être envoyés à `nicolas.pecheux@cpge.info` avant 07h59 le 6 mars 2017. Vous enverrez un unique fichier `.ml` dont le nom sera obligatoirement et exactement `<login>.ml` où `<login>` est votre nom d'utilisateur du laboratoire d'informatique (tout en minuscules avec les tirets).

Votre programme devra respecter scrupuleusement les noms et les types indiqués (ou un type plus général ou équivalent bien entendu) car il sera en partie corrigé automatiquement. Le code fourni devra compiler sans erreurs ni aucun message d'avertissement. L'encodage du fichier devra être UTF-8 et vous n'utiliserez que des caractères ASCII pour les noms de variables. Il est impératif de veiller au strict respect de ces consignes.

La clarté et la simplicité du code et l'utilisation judicieuse de commentaires seront pris en compte dans l'évaluation. On prendra bien soin de tester toutes les fonctions sur des cas triviaux, puis un peu plus complexes.

Lorsque l'on pose des hypothèses sur les arguments (par exemple qu'un argument est un entier naturel), il n'est pas nécessaire dans vos fonctions de le vérifier

ni de prévoir le comportement si l'hypothèse est violée (par exemple si l'argument est négatif). On pourra cependant lever des exceptions (`failwith "Argument non valable"` par exemple) si on le souhaite.

EXERCICE 6 *Réduction d'éléments consécutifs égaux*

Écrire une fonction `reduction : 'a list -> 'a list` qui réduit à un seul exemplaire les cas éventuels d'éléments consécutifs égaux dans une liste. Par exemple :

```
reduction [1; 1; 1; 2; 3; 3; 4; 5; 5; 5; 5; 6; 6; 7; 8; 8; 9];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9]
```

EXERCICE 7 *Aplatissement d'une liste de listes*

Écrire une fonction `aplatir : 'a list list -> 'a list` aplatissant une liste de listes, sans utiliser l'opérateur `@`. Par exemple :

```
aplatir [[3; 4]; [5; 7; 9]; [0]; [6; 8]]
- : int list = [3; 4; 5; 7; 9; 0; 6; 8]
```

EXERCICE 8 *Cet exercice plus difficile est facultatif*

Écrire une fonction `plateau : 'a list -> int * int * 'a` identifiant le plus long plateau formé d'éléments consécutifs identiques dans une liste non vide quelconque (pas nécessairement triée).

Le résultat sera le triplet `(debut, longueur, element)` formé par la position du début du plateau, par sa longueur et par la valeur de l'élément ainsi répété. Par exemple :

```
plateau ["e"; "x"; "x"; "c"; "c"; "x"; "c"; "c"; "c"; "c"];;
- : int * int * string = 6, 4, "c"
```

PROBLÈME 1 *Représentation des polynômes*

On représente un polynôme par la liste de ses coefficients, rangés dans l'ordre *croissant* des degrés. Par exemple, le polynôme $X^3 + 2X - 1$ est représenté par la liste `[-1.; 2.; 0.; 1.]`. Le polynôme nul est représenté par la liste vide.

Une telle liste est dite *propre* quand son dernier terme est non nul ou quand elle est vide. Un polynôme a une et une seule représentation par une liste propre, mais a une infinité de représentations impropres : la liste `[-1.; 2.; 0.; 1.; 0.; 0.]` représente aussi le polynôme $X^3 + 2X - 1$.

1. Écrire une fonction `evaluation : float list -> float -> float` qui calcule l'évaluation d'un polynôme [1] en un point.

   On remarquera que si $P = \sum_{k=0}^{n} a_k X^k$ on a

   $$P(x) = a_0 + x(a_1 + x(\cdots + x(a_{n-1} + x(a_{n-1} + xa_n))\dots))$$

2. Écrire une fonction qui calcule la liste associée au produit λP d'un polynôme P par un scalaire λ :
   `produit_externe : float list -> float -> float list`
3. Écrire une fonction qui calcule une liste représentant la somme $P + Q$ de deux polynômes :
   `somme : float list -> float list -> float list`
4. Écrire une fonction qui calcule une liste représentant le produit $P \times Q$ de deux polynômes :
   `produit : float list -> float list -> float list`
5. Écrire une fonction qui calcule une liste représentant le polynôme P^k :
   `puissance : float list -> int -> float list`
6. Écrire une fonction qui calcule une liste représentant la composition $P(Q(X))$ de deux polynômes :
   `compose : float list -> float list -> float list`
7. Les fonctions précédentes peuvent éventuellement construire des représentations impropres. Écrire une fonction
   `nettoyage : float list -> float list`
   qui construit la représentation propre d'un polynôme. On pourra librement utiliser la fonction prédéfinie `rev` qui construit la liste miroir d'une liste :
   `rev [1; 5; 4] = [4; 5; 1]`
8. Écrire une fonction `coefficient_dominant : float list -> float` qui calcule le coefficient dominant d'un polynôme P supposé non nul.

1. Attention : on utilise la notation P comme en mathématiques, mais en CAML, vos noms de variables doivent impérativement commencer par une minuscule. On choisira par exemple `polynome`.