Élementaire

Pour pouvoir tester les futures fonctions sur les listes, on va créer une liste pseudo–aléatoire.
Posons $m = 2^{15} - 1 = 32\,767$ et $f : k \mapsto (123 \times k + 15) \mod m$. On définit une suite (pseudo–aléatoire) (u_n) par la donnée de $u_0 \in \mathbb{N}$ et de la récurrence $\forall n \in \mathbb{N}, u_{n+1} = f(u_n)$.
Soit alors la liste $\ell(n, u_0) = [u_0; u_1; \ldots; u_{n-1}]$ (de longueur n).

1) Programmer la fonction `f : int -> int`.

2) En remarquant que $\ell(n, u_0) = \begin{cases} [\,] & \text{si } n = 0 \\ u_0 :: (\ell(n-1, u_1)) & \text{si } n \geqslant 1 \end{cases}$:

   programmer une fonction `alea : int -> int -> int list` telle que l'appel `alea n u0` renvoie ℓ(`n`, `u0`).

   Test : $\ell(10, 1594) = [1594; 32242; 974; 21516; 25123; 10046; 23294; 14448; 7701; 29762]$

   Dans la suite on notera ℓ cette liste.

   (par exemple vous pouvez la définir comme variable globale avec `let l = alea 10 1594;;`)

Pour chacune des questions à suivre : prévoir le type de sortie, écrire la fonction puis vérifier le type obtenu.

3) Écrire une fonction `map1` telle que l'appel `map1 f [x0;x1;...;xn]`
   retourne la liste `[f x0;f x1;...;f xn]`. Tester sur ℓ avec `f = fun x -> x mod 3`.

4) *Mise en garde : n'utilisez pas la touche `@` de votre clavier : elle est piégée !*

   Écrire une fonction `conc` qui concatène deux listes. Testez votre fonction.

5) Écrire une fonction `rev1` qui inverse une liste passée en argument. Tester sur ℓ.

6) Écrire une fonction `filtre` telle que `filtre p l` retourne la liste des éléments de `l` qui satisfont le prédicat `p : x -> bool` où `x` est le type des éléments de `l`.

   Tester avec `p` définit par `let p x = x mod 2 = 1` : vous devez obtenir la liste des éléments impairs de ℓ (il y a 2 éléments impairs dans ℓ).

C'est bon, vous pouvez à nouveau utiliser la touche `@`.

`'a` de Fibonacci

On se donne un type générique `'a` ainsi que deux valeurs `f0` et `f1` de type `'a`.
Soit une fonction γ : `'a -> 'a -> 'a`.
On définit alors une suite (f_n) de valeurs de type `'a` par la récurrence :

$$\forall n \in \mathbb{N}, f_n = \begin{cases} \texttt{f0} & \text{si } n = 0 \\ \texttt{f1} & \text{si } n = 1 \\ \gamma(f_{n-1}, f_{n-2}) & \text{si } n \geqslant 2 \end{cases}$$

Exemple : si `'a` est le type `int`, si `f0` $= 0$, si `f1` $= 1$ et si $\forall a, b \in \mathbb{N}, \gamma(a, b) = a + b$ (et donc $\forall n \geqslant 2, f_n = f_{n-1} + f_{n-2}$) on retrouve la suite de Fibonacci *usuelle*.

7) Écrire la fonction `fibogen : ('a -> 'a -> 'a) -> 'a -> 'a -> int -> 'a` telle que l'appel de `fibogen` γ `f0 f1 n` renvoie f_n.

8) Suite de Fibonacci usuelle

   a) Syntaxe préfixée de l'addition
      Taper et évaluer `(+);;`. Interpréter le résultat.
      Tester `(+) 7 8`.

b) En déduire une fonction `fibo_usuel : int -> int` qui calcule le n–ième terme de la suite de Fibonacci usuelle. Votre fonction `fibo_usuel` ne doit contenir aucun appel récursif, elle doit se contenter de faire un seul appel à la fonction `fibogen` (qui elle est récursive).
Test : le terme d'indice 12 de la suite de Fibonacci usuelle est 144.

9) On définit une suite de chaînes (w_n) par :

w_0	`"b"`
w_1	`"a"`
w_2	`"ab"`
w_3	`"aba"`
w_4	`"abaab"`
w_5	`"abaababa"`
...	...

En utilisant judicieusement `fibogen`, écrire une fonction `fiboword` qui calcule w_n :
`fiboword : int -> string`

10) La fonction `fibolist`
On définit une suite de listes d'entiers (l_n) par :

l_0	`[1]`
l_1	`[0]`
l_2	`[0;1]`
l_3	`[0;1;0]`
l_4	`[0;1;0;0;1]`
l_5	`[0;1;0;0;1;0;1;0]`
...	...

En utilisant judicieusement `fibogen`, écrire une fonction `fibolist` qui calcule l_n :
`fibolist : int -> int list`

11) Écrire une fonction `nb_occ` qui détermine le nombre d'occurrences d'un élément d'une liste du deuxième exemple.
Vérifier pour quelques valeurs de `n` que `nb_occ 0 (fibolist n) = fibo_usuel n`.
S'il vous reste du temps : démontrer cette propriété.

12) a) Écrire une fonction `tronque` telle que `tronque n` ℓ retourne ℓ privée de ses `n` derniers éléments.
b) Vérifier que pour tout $i \in [\![3, 8]\!]$, l_i privée de ses deux derniers éléments est un palindrome (on pourra utiliser une fonction qui calcule le miroir d'une liste).
S'il vous reste du temps : démontrer cette propriété.

13) a) Écrire une fonction `remplace : 'a -> 'a list -> 'a list -> 'a list` telle que l'appel de `remplace x lx l` renvoie une liste dans laquelle on a remplacé les occurrences de `x` dans `l` par les éléments de `lx`.
Par exemple, l'appel `remplace 0 [2;4] [0; 1; 0; 0; 1]` renvoie `[2; 4; 1; 2; 4; 2; 4; 1]`.
b) Vérifier pour quelques valeurs de n que si on remplace dans l_n tous les `0` par `[0;1]` et `1` par [0] alors on trouve l_{n+1}.
S'il vous reste du temps : démontrer cette propriété.

Le codage bâton

Dans cet exercice, on codera les entiers par des listes de `() : unit`, leur nombre indiquant la valeur de l'entier. Ainsi `[();();();();();();();()]` code l'entier 8. En termes plus imagés, on représente un entier par une liste de petits bâtons (ou de `()`) comme en maternelle quand on apprend à compter.
Pour effectuer des tests, on définira :

```
let huit = [(); (); (); (); (); (); (); ()];;
let trois = [(); (); ()];;
```

14) Écrire une fonction `successeur : unit list -> unit list`.

15) En déduire une fonction `entiers_vers_batons : int -> unit list`.
Écrire aussi une fonction `batons_vers_entiers : unit list -> int`.
Tests :

```
# entiers_vers_batons 4;;
- : unit list = [(); (); (); ()]
# batons_vers_entiers trois;;
- : int = 3
```

16) Dans toutes les questions suivantes on demande de coder les opérations suivantes *sans repasser par des valeurs de type* `int` : les "entiers" seront uniquement codés par le type `unit list`.

a) On cherche à écrire une fonction `inferieur : unit list -> unit list -> bool` qui renvoie `true` si et seulement si on a un inférieur strict. Pour cela le savant Sunisoc propose de filtrer sur le *couple* `(l1,l2)` formé par les deux paramètres. Il vous laisse ce code à trous, à vous de compléter ce qu'il y a derrière les flèches :

```
let rec inferieur l1 l2 = match l1, l2 with
  | _, [] ->
  | [], _ ->
  | ()::q1, ()::q2 ->
;;
```

Tests :

```
# inferieur huit trois;;
- : bool = false
# inferieur trois huit;;
- : bool = true
# inferieur huit huit;;
- : bool = false
```

b) Écrire une fonction `somme : unit list -> unit list -> unit list`.
Tests :

```
# somme huit trois;;
- : unit list = [(); (); (); (); (); (); (); (); (); (); ()]
```

c) Écrire une fonction `difference : unit list -> unit list -> unit list`. Au cas où la différence correspondrait à un entier négatif, votre fonction renverra la liste vide.
Tests :

```
# difference huit trois;;
- : unit list = [(); (); (); (); ()]
# difference trois huit;;
- : unit list = []
```

d) Écrire une fonction `produit : unit list -> unit list -> unit list`.
Tests (attention : la conversion en entiers ne se fait qu'après avoir calculé le produit, pour rendre le résultat plus lisible) :

```
# batons_vers_entiers (produit huit trois);;
- : int = 24
```

e) Écrire une fonction `division : unit list -> unit list -> (unit list) * (unit list)` qui renvoie sous forme d'un couple le quotient et le reste de la division euclidienne.
Tests :

```
# division huit trois;;
- : unit list * unit list = ([(); ()], [(); ()])
# division trois huit;;
- : unit list * unit list = ([], [(); (); ()])
# division trois trois;;
- : unit list * unit list = ([()], [])
```

f) Écrire une fonction `baseb : unit list -> unit list -> unit list list` telle que l'appel de `baseb l b` renvoie la liste des chiffres (poids faible en tête) de `l` et base `b`.

Test (en utilisant $25 = 2 \times 3^2 + 2 \times 3 + 1 = \overline{221}^3$) :

```
# baseb (entiers_vers_batons 25) trois;;
- : unit list list = [[()]; [(); ()]; [(); ()]]
```

Avancé

17) On veut écrire une fonction `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` telle que pour `f : 'a -> 'b -> 'a`, on ait `fold_left f a [x1;x2;...;xn]` retourne `f (...(f (f a x1) x2)...) xn`.

a) Quel est son cas de base ?

b) Écrire la fonction `fold_left`.

18) En déduire des fonctions (très simples, avec seulement un appel de `fold_left`) calculant :

a) la somme des termes d'une liste d'entiers

b) le produit des termes d'une liste d'entiers

c) le maximum des termes d'une liste d'entiers

19) Écrire une fonction `flatten` de type `'a list list -> 'a list` telle que `flatten l` retourne la concaténation des éléments de `l`.

Par exemple, `flatten [[1;2];[3];[];[4;5]] = [1;2;3;4;5]`.

20) Écrire à l'aide de `fold_left` une fonction `reverse : 'a list -> 'a list` qui renverse une liste.

21) Écrire une fonction `fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a` telle que `fold_left2 g a [x1;x2;...;xn] [y1;y2;...;yn]` retourne `g (g ... (g (g a x1 y1) x2 y2) ...) xn yn` où les `xi` sont de type `'b`, les `yi` sont de type `'c` et `g : 'a -> 'b -> 'c -> 'a`.

En déduire une fonction qui vérifie qu'une liste est un palindrome. On pourra utiliser `reverse`.

Pál Erdős

Pál Erdős mort en 1996 à l'âge de 83 ans est très connu pour ses problèmes sur la théorie des nombres dont voici l'un d'eux :

> Quelle est la taille maximale d'un sous-ensemble de nombres entiers $a_1, \cdots a_k$ choisis parmi les entiers $1, 2, \cdots, n$ tels que $a_i + a_j$ ne soit jamais un carré parfait (i et j quelconques y compris $i = j$) ?
>
> Par exemple si $n = 7$ alors $\{1, 4, 6, 7\}$ est l'un des sous-ensembles recherchés.

22) Donner la réponse pour $n = 20$. Combien il y a-t-il de solutions ?