université de BORDEAUX	**Année universitaire:** 2019/2020 **Parcours:** Licence Informatique 2e année UE 4TINA01U **Épreuve:** Examen de Programmation fonctionnelle Session 1 **Date:** Vendredi 20 décembre **Durée:** 1h20	**Collège Sciences et Technologies**

Les documents sont **interdits**. On pourra utiliser (si besoin) les fonctions de la bibliothèque `List` de OCaml.
On pourra réutiliser une fonction demandée à une question dans la suite (même si la question n'a pas été traitée)

Exercice 1 : Itérations — 5 Points

Étant donnée une liste `l` et un nombre entier `n`, on associe une nouvelle liste qu'on appelle *la `n`-itération de `l`*. Cette liste est obtenue en concaténant `l` avec elle-même `n` fois. La `0`-itération de toute liste est la liste vide.

Exemple

On considère la liste `[1;4;5]`. Sa `0`-itération est la liste vide `[]`, sa `1`-itération est `[1;4;5]`, sa `2`-itération est `[1;4;5;1;4;5]`, sa `3`-itération est `[1;4;5;1;4;5;1;4;5]`, ...

1) Écrire une fonction `iterate` de type `'a list -> int -> 'a list`. L'évaluation de `iterate l n` doit retourner la `n`-itération de `l` (on pourra utiliser la concaténation de listes).

2) Écrire une fonction `cut_prefix` de type `'a list -> 'a list -> bool * 'a list`. Étant donnés deux listes `p` et `l`, l'évaluation de cette fonction doit retourner la paire suivante :

- si il existe une troisième liste `s` telle que `l = p @ s` (c'est-à-dire que `l` est la concaténation de `p` avec `s`), `cut_prefix p l` doit retourner la paire `(true,s)`.
- sinon `cut_prefix p l` doit retourner la paire `(false,[])`.

3) Écrire une fonction `is_iter` de type `'a list -> 'a list -> bool`. L'évaluation de `is_iter l1 l2` doit retourner `true` si il existe un entier `n` tel que `l2` est la `n`-itération de `l1`, et `false` sinon.

Exercice 2 : Polynômes — 8 Points

On représente un polynôme à coefficients entiers par la liste (`int list`) de ses coefficients. Plus précisément, la liste `[c0 ; c1 ; c2 ; ... ; cn]` représente le polynôme $P(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n$. Par convention, la liste vide `[]` représente le polynôme constant égal à zéro : $P(x) = 0$.

Remarque

Attention, deux listes distinctes peuvent représenter le **même** polynôme. Par exemple, `[2 ; 5 ; 0 ; 7]` et `[2 ; 5 ; 0 ; 7 ; 0 ; 0 ; 0]` représentent toutes deux le polynôme $P(x) = 2 + 5x + 7x^3$. De même, les listes `[0 ; 0 ; 0]` et `[]` représentent toutes deux le polynôme $P(x) = 0$.

1) Écrire une fonction `poly_sum` de type `int list -> int list -> int list`. L'appel de `poly_sum l1 l2` retourne une liste représentant $P_1(x) + P_2(x)$ où $P_1(x)$ et $P_2(x)$ sont les polynômes représentés par les listes `l1` et `l2`. Par exemple, `poly_sum [2; 5; 0] [0; 1; 2]` retourne une liste représentant $2 + 6x + 2x^2$.

2) Écrire une fonction `multdeg` de type `int -> int list-> int list`. L'évaluation de `multdeg k l` retourne une liste représentant le polynôme $x^k \times P(x)$ où $P(x)$ est le polynôme représenté par la liste `l` passée en paramètre. Par exemple, sur `multdeg 2 [3; 1; 0; 4; 0]`, la liste retournée représente $3x^2 + x^3 + 4x^5$.

3) Écrire une fonction `multcons` de type `int -> int list -> int list`. `multcons c l` retourne une liste représentant le polynôme $c \times P(x)$ où $P(x)$ est le polynôme représenté par la liste `l` passée en paramètre. Par exemple, `multcons 2 [2; 5; 0; 7; 0; 0]` retourne une liste représentant le polynôme $4x + 10x + 14x^3$.

4) Écrire une fonction `poly_mult` ayant pour type `int list -> int list -> int list`. L'évaluation de `poly_mult l1 l2` retourne une liste représentant le polynôme $P_1(x) \times P_2(x)$ où $P_1(x)$ et $P_2(x)$ sont les polynômes représentés par les listes `l1` et `l2` passées en paramètre.

5) Écrire une fonction `degree` de type `int list -> int`. L'évaluation de `degree l` retourne le degré du polynôme représenté par la liste `l` passée en paramètre.

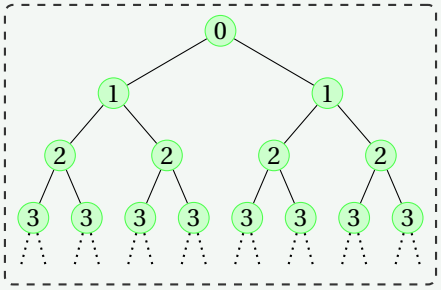
6) Écrire une fonction `evaluate` de type `int list -> int -> int`. L'évaluation de `evaluate l n` retourne l'entier $P(n)$ où P est le polynôme représenté par la liste `l` passée en paramètre.
Par exemple, `evaluate [2; 5; 0; 1; 0] 2` retourne l'entier $20 = 2 + 5 \times 2 + 1 \times 2^3$.

Exercice 3 : Arbres infinis — 7 Points

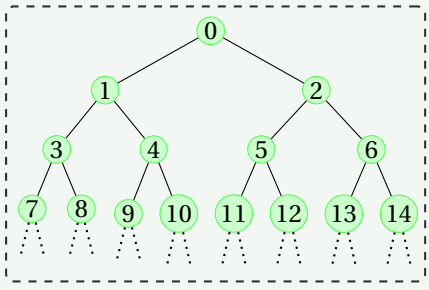
On considère des arbres binaires parfaits qui contiennent une *infinité de nœuds*. Dans un tel arbre, tout nœud a *exactement deux fils* (il n'y a pas de feuilles) et porte une *étiquette*.

Deux exemples d'arbres infinis étiquetés par des entiers

Dans le premier, `t_depth`, chaque nœud est étiqueté par sa profondeur (c'est à dire le nombre d'arêtes le reliant à la racine). Le second, `t_bread_first`, contient *tous* les nombres entiers positifs *exactement une fois*.



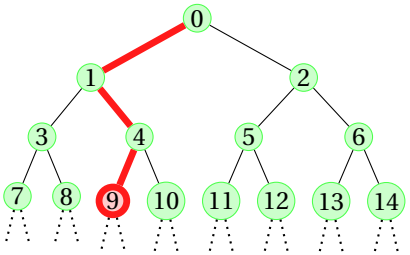
Arbre `t_depth`



Arbre `t_naturals`

On explique comment représenter ces arbres en *Ocaml*. Par définition, deux arbres binaires infinis distincts sont *structurellement identiques* : ils ne diffèrent que par les étiquettes de leur nœuds. On va donc représenter un arbre par une *fonction* qui associe une étiquette à chaque nœud de l'arbre infini. Pour cela, il nous faut au préalable un moyen de désigner chaque nœud d'un arbre infini. On utilise le type suivant, `type dir = G | D`.

On se sert de listes de type `dir list` pour désigner les nœuds d'un arbre binaire infini. Intuitivement, un élément de type `dir` est une *direction* (**G**auche ou **D**roite). Ainsi, une liste `l` de type `dir list` spécifie un chemin dans l'arbre binaire infini : celui-ci part de la racine et suit les directions données par la liste. La liste `l` désigne le nœud sur lequel termine le chemin. Par exemple la liste `[]` désigne la racine et la liste `[G ;D ;G]` désigne le nœud obtenu en partant de la racine et en prenant successivement l'enfant gauche, puis l'enfant droit, puis l'enfant gauche (voir la figure ci-contre).



Nœud désigné par `[G ;D ;G]`
(sur l'arbre `t_naturals`)

On représente les arbres infinis avec le type suivant `type 'a itree = dir list -> 'a`. Un arbre donné est représenté par la fonction qui, à toute liste de directions, associe l'étiquette du nœud qu'elle désigne dans l'arbre.

1) On considère l'expression `let t_one = fun _ -> 1`. Quel est l'arbre représenté par `t_one` ?

2) Définir une variable `t_depth` qui représente l'arbre du même nom décrit dans l'exemple.

3) Définir une variable `t_right` qui représente un arbre binaire infini étiqueté par des Booléens et dans lequel chaque nœud à pour étiquette `true` si il est le fils droit d'un autre nœud et `false` sinon.

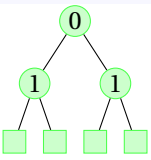
4) Écrire une fonction `sub_right` de type `'a itree -> 'a itree`. L'évaluation de `sub_right t` doit retourner l'arbre binaire infini qui est le sous-arbre droit de l'arbre `t` passé en paramètre.

5) On réutilise le type `'a btree` vu en cours pour représenter les arbres binaires **finis** :

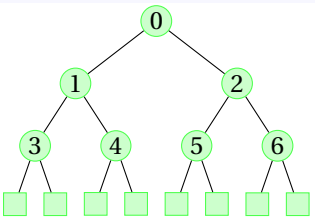
```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```

Écrire une fonction `itree_to_btree` de type `'a itree -> int -> 'a btree` telle que l'évaluation de `itree_to_btree t n` retourne l'arbre *fini* obtenu à partir de l'arbre *infini* `t` en gardant les nœuds de profondeur *au plus* `n` et en remplaçant les autres par des feuilles. On donne deux exemples d'évaluation de la fonction pour les arbres `t_naturals` et `t_depth` présentés au début :

`itree_to_btree t_depth 1`



`itree_to_btree t_naturals 2`



6) Définir une variable `t_naturals` qui représente l'arbre du même nom décrit dans l'exemple.