DM de révisions pour les grandes vacances

Lorsque l'on pose des hypothèses sur les arguments (par exemple qu'un argument est un entier naturel), il n'est pas nécessaire dans les 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 l'on souhaite.

La plupart des exercices sont très simples (quelques lignes maximum) et sont là pour vous aider à vous entraîner. Se cachent cependant sans prévenir quelques exercices plus difficiles ou plus longs. Certains des exercices proposés vous feront réviser le chapitre sur la programmation dynamique (lesquels ?).

1 Quelques exercices sur les entiers

EXERCICE 1 *Nombre de chiffres d'un entier*

Écrire une fonction `ndigits : int -> int` donnant le nombre de chiffres d'un entier naturel.

EXERCICE 2 *Test de primalité*

Écrire une fonction `is_prime: int -> bool` testant si un entier strictement positif est premier.

EXERCICE 3 *Factorisation en nombres premiers*

Écrire une fonction `prime_factors : int -> (int * int) list` qui renvoie la décomposition en facteurs premiers d'un entier $n > 0$ sous la forme d'une liste $[(p_i, m_i)]_{i=1}^{k}$, où les p_i sont les facteurs premiers de n dans l'ordre croissant et $m_i \geq 1$ leur multiplicité (*i.e* $n = \prod_{i=1}^{k} p_i^{m_i}$).

EXERCICE 4 *Coefficients de Bézout d'une famille de n entiers*

Écrire une fonction `bezout : int list -> int list` qui prend en argument une liste d'entiers $[a_1; a_2; \ldots; a_n]$ et qui retourne une liste $[b_1; b_2; \ldots; b_n]$ d'entiers telle que $\sum_{i=1}^{n} a_i b_i = \text{pgcd}(a_1, a_2, \ldots, a_n)$.

2 Quelques exercices sur les chaînes de caractères

EXERCICE 5

Écrire une fonction `chars : string -> char list` transformant une chaîne de caractères en la liste de ses caractères.

EXERCICE 6 *Palindromes*

Écrire une fonction `palindrome : string -> bool` testant si une chaîne de caractères (dans '[a-z]') est identique qu'elle soit lue de gauche à droite ou de droite à gauche, en ignorant le caractère espace (caractère ASCII 32). Par exemple *esope reste ici et se repose* est un palindrome.

EXERCICE 7

Écrire une fonction `longueur_min_surchaine : string -> string -> int` qui, en fonction de deux chaînes de caractères u et v, renvoie la longueur minimale d'une chaîne w de caractères dont u et v sont des sous-chaînes, c'est-à-dire que les lettres de u et de v apparaissent dans w dans le bon ordre, mais de manière non nécessairement consécutive). Par exemple la longueur minimale pour les chaînes `u = "atga"` et `v = "gtaaag"` est 8 et une chaîne minimale est `w = "agtaaaga"` (on a `u = "a--t---ga"` et `v = "-gtaaag-"`).

3 Quelques exercices sur les vecteurs

EXERCICE 8

Écrire une fonction `subvect : 'a vect -> int -> int -> 'a vect` renvoyant le sous-vecteur formé des éléments dont les positions sont comprises entre des valeurs entières a et b, large pour a et strict pour b. Si la taille du vecteur est n, on suppose que $a \leq b$ et que $0 \leq a < n$ et $0 \leq b \leq n$. Ceci suppose donc que vecteur n'est pas vide.

EXERCICE 9 *Miroir d'un vecteur*

Écrire une fonction `rev_vect : 'a vect -> unit` qui inverse l'ordre des éléments d'un vecteur.

EXERCICE 10

Écrire une fonction `map_vect : ('a -> 'b) -> 'a vect -> 'b vect` qui applique une fonction `f` à tous les éléments d'un vecteur `v`.

EXERCICE 11

Écrire une fonction `map_op_vect : ('a -> 'b -> 'a) -> 'a -> 'b vect -> 'a` itérant un même opérateur binaire (préfixe) f à tous les éléments d'un vecteur v, avec une valeur initiale a. Ainsi `map_op_v f a [|v0; v1; v2|]` renvoie `f (f ( f a v0) v1) v2)`.

En déduire et écrire les fonctions `sum_vect : int vect -> int` et `prod_vect : int vect -> int` calculant respectivement la somme et le produit des éléments d'un vecteur d'entiers.

EXERCICE 12

Écrire une fonction `rot_vect : 'a vect -> int -> 'a vect` effectuant une rotation de $|p|$ positions, vers la gauche si $p > 0$ et vers la droite si $p < 0$, d'un vecteur v dans un nouveau vecteur en sortie, sans modifier v.

Par exemple :

```
rot_vect [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10|] 3;;
- : int vect = [|4; 5; 6; 7; 8; 9; 10; 1; 2; 3|]
rot_vect [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10|] (-13);;
- : int vect = [|8; 9; 10; 1; 2; 3; 4; 5; 6; 7|]
```

EXERCICE 13

Écrire une fonction `plateau : 'a vect -> int * int` identifiant le plus long plateau formé d'éléments consécutifs identiques dans un vecteur non vide. Le résultat est le couple formé par la position du début du plateau et par la longueur de celui-ci. On suppose que le vecteur n'est pas vide.

Par exemple :

```
plateau [|"e"; "c"; "c"; "z"; "c"; "c"; "x"; "c"; "c"; "c"; "c"; "u"; "c"|];;
- : int * int = 7, 4
```

EXERCICE 14

Écrire une fonction `plus_longue_sous_suite_croissante : int vect -> int` qui renvoie la longueur maximale d'une sous-suite croissante (au sens large). Par exemple si le tableau est `[|1; 2; 2; 4; 5; 11; 9; 12; 9; 3|]` la longueur recherchée est 7 et est atteinte pour trois sous-suites différentes.

EXERCICE 15

Écrire une fonction `plus_grande_somme : int vect -> int -> int` qui, étant donné un vecteur v d'entiers strictement positifs et une constante $M > 0$, renvoie la plus grande somme inférieure ou égale à M d'une partie formée d'éléments de v, si elle existe, et 0 sinon.

4 Quelques exercices sur les matrices

EXERCICE 16 *Produit "naïf" de deux matrices de flottants*

Écrire une fonction `mult_matrix : float vect vect -> float vect vect -> float vect vect` réalisant le produit de deux matrices de flottants quelconques (en supposant cependant les dimensions compatibles et non nulles).

EXERCICE 17 *Organisation du calcul d'un produit de matrices*

Pour $k \geq 2$, on se donne des matrices $M_1, M_2, \dots, M_k$ avec $\forall i \in [\![1,k]\!], M_i \in \mathcal{M}_{m_i m_{i+1}}(\mathbb{R})$, et on veut calculer le produit $M_1 M_2 \dots M_k$. Pour $i \in [\![1,k-1]\!]$, le produit $M_i M_{i+1}$ a un coût $m_i m_{i+1} m_{i+2}$ et l'ordre dans lequel on effectue les différents produits peut avoir un impact sur la complexité totale.

1. Se convaincre sur un exemple.
2. Écrire une fonction `calcul_minimal : int list -> int` qui calcul le coût minimal du produit de matrices $M_1 M_2 \dots M_k$ en fonction de $m_1, m_2, \dots, m_{k+1}$ donnés en entrée.

5 Quelques exercices sur les listes

EXERCICE 18 *Avant-dernier élément*

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

EXERCICE 19

Écrire une fonction `nth : int -> 'a list -> 'a` renvoyant l'élément d'indice $n \in \mathbb{N}$ dans une liste (l'élément en tête ayant pour indice 0) et `Failure "Liste trop courte"` s'il n'existe pas.

EXERCICE 20

Écrire une fonction `sublist : 'a list -> int -> int -> 'a list` renvoyant la sous-liste formée des éléments dont les positions sont comprises entre les valeurs entières a et b, large pour a et strict pour b, avec les mêmes hypothèses sur a et b qu'à l'exercice 8.

EXERCICE 21 *Minimum et maximum d'une liste*

Écrire une fonction `min_max_list : int list -> int * int` qui renvoie le couple formé par le plus petit et le plus grand élément d'une liste non vide.

EXERCICE 22 *Existence d'un élément satisfaisant à un prédicat*

Écrire une fonction `exists : ('a -> bool) -> 'a list -> bool` qui renvoie vrai s'il existe un élément satisfaisant un prédicat passé en entrée et faux sinon.

EXERCICE 23 *Supprimer le n-ième élément d'une liste*

Écrire une fonction `del_list : int -> 'a list -> 'a list` qui supprime le n-ième élément d'une liste s'il existe et ne fait rien sinon.

EXERCICE 24 *Réduction d'éléments consécutifs égaux*

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

Par exemple :

```
reduc_list [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 25 *Aplatissement d'une liste de listes*

Écrire une fonction `flatten : 'a list list -> 'a list` aplatissant une liste de listes.

Par exemple :

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

EXERCICE 26 *Plus grand intervalle d'éléments égaux dans une liste quelconque*

Écrire une fonction `plateau_list : '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 `(d, l, x)` formé par la position `d` du début du plateau, par sa longueur `l` et par la valeur `x` de l'élément ainsi répété. Par exemple :

```
plateau_list ["e"; "x"; "x"; "x"; "c"; "c"; "x"; "c"; "c"; "c"; "c"; "u"; "c"];;
- : int * int * string = 7, 4, "c"
```

EXERCICE 27 *Les 2^n listes de n éléments de $\{0,1\}$*

Écrire une fonction `listes_binaires : int -> int list list` qui prend en argument un entier naturel n et qui retourne l'ensemble des suites des 2^n listes de n éléments pris dans l'ensemble $\{0,1\}$ classées par ordre lexicographique croissant. Le code des éventuelles fonctions intermédiaires doit être récursif.

6 Quelques exercices sur les arbres

Dans toute cette partie on considère des arbres binaires définis par le type

```
type arbre = Feuille of int | Noeud of arbre * arbre * int;;
```

EXERCICE 28

Écrire une fonction `max_min_arbre : arbre -> int * int` qui renvoie le minimum et le maximum des étiquettes d'un arbre.

EXERCICE 29

Écrire une fonction `somme_feuilles : arbre -> int` qui renvoie la somme des étiquettes de toutes les feuilles.

EXERCICE 30

Une branche est un chemin de la racine vers une feuille. Le poids d'une branche est la somme des étiquettes des nœuds qui la composent. Écrire une fonction `max_somme_branche : arbre -> int` qui renvoie le poids de la branche de poids maximal.

EXERCICE 31

Écrire une fonction `parcours_profondeur_infixe : arbre -> int list` qui renvoie la liste des étiquettes dans l'ordre d'un parcours en profondeur infixe (de gauche à droite).

EXERCICE 32

Écrire une fonction `parcours_largeur : arbre -> int list` qui renvoie la liste des étiquettes dans l'ordre d'un parcours en largeur (de gauche à droite).

7 Quelques exercices sur les tris

On suppose que l'on dispose d'une fonction `compare : 'a -> 'a -> int` qui est strictement négative si le premier argument est strictement inférieur au deuxième, strictement positive s'il est strictement supérieur et qui vaut 0 si les deux arguments sont égaux.

EXERCICE 33

On demande dans cette partie d'écrire le tri insertion, le tri rapide, et le tri fusion qsur des listes et sur des vecteurs (tris par ordre croissant). Les fonctions sur les listes auront pour type `('a -> 'a -> int) -> 'a list -> 'a list` et pour suffixe `_list` ; et celles sur les vecteurs `('a -> 'a -> int) -> 'a vect -> unit` et pour suffixe `_vect`. Ainsi on demande d'écrire les fonctions :

- `tri_insertion_list`
- `tri_insertion_vect`
- `tri_rapide_list`
- `tri_rapide_vect`
- `tri_fusion_list`
- `tri_fusion_vect`

Si vous n'êtes pas rassasiés après ces exercices et que vous en souhaitez davantage, vous pouvez consulter la page CAML du site `http://mathprepa.fr/caml-option-informatique/` de Jean-Michel Ferrard qui contient une base d'exercices très complète (dont sont d'ailleurs tirés un certain nombre des exercices proposés ici).