TP13.5: OCaml impératif: exercices

Exercice 1: Tableaux

Q1. Se renseigner sur les fonctions suivantes du module `Array` d'OCaml, et les réimplémenter par vos propres moyens:

1. `map`
2. `for_all`
3. `append` puis `concat`
4. `blit`

Exercice 2: Remplissage de matrice

En OCaml, on peut représenter une matrice par un tableau de tableaux:

```
type 'a matrix = 'a array array

let (m: int matrix) = [|
  [|1; 2; 4|];
  [|0; 4; 3|];
  [|5; 5; 0|]
|]
```

On accède aux éléments de `m` avec `m.(i).(j)`.

Q1. Écrire une fonction `make_matrix: int -> int -> 'a -> 'a matrix` telle que `make_matrix n m x` renvoie une matrice $n \times m$ dont chaque case vaut x. **ATTENTION:** vérifiez que modifier une case de la matrice obtenue ne modifie pas les autres cases.

On considère une matrice M entière de dimensions $n \times n$. La matrice des sommes de M, notée $S(M)$, est une matrice entière de dimensions $n+1 \times n+1$ telle que pour tout $i, j \in [\![0, n]\!]$, $S(M)_{i,j}$ est la somme de toutes les cases de M d'indices i', j' avec $i' < i$ et $j' < j$.

Q2. Chercher une formule de récurrence liant $S(M)_{i+1,j+1}$ à $S(M)_{i,j+1}$, $S(M)_{i+1,j}$ et $S(M)_{i,j}$.

Q3. Implémenter une fonction `sommes: int array array -> int array array` renvoyant la matrice des sommes d'une matrice M donnée. Quelle est sa complexité ? Comparer à la complexité de l'algorithme naïf.

Exercice 3: Parcours d'arbre, logique

On définit le type suivant pour les arbres binaires:

```
type 'a ab = V | N of 'a * 'a ab * 'a ab
```

On rappelle le pseudo-code du parcours en profondeur itératif, que l'on note parfois DFS (comme Depth-First Search):

Algorithme 1 : `DFS`(a)

Entrée(s) : a un arbre binaire non-vide
Sortie(s) : L liste des étiquettes de a dans l'ordre postfixe

```
P ← pile_vide();
r ← racine de a;
L ← [r];
empiler(P, r);
tant que P non vide faire
    u ← depiler(P);
    si u est de la forme N(x, g, d) alors
        L ← r :: L;
        empiler(P, d);
        empiler(P, g);
retourner L
```

Q1. Implémenter cet algorithme en une fonction `dfs: 'a ab -> 'a list`

On considère maintenant des formules propositionnelles, où l'ensemble des variables est numéroté: $X_0, X_1, \dots$ On représente ces formules en OCaml par le type:

```
type formule =
  | Top | Bot | Var of int
  | And of formule * formule
  | Or  of formule * formule
  | Not of formule
```

Par exemple, la formule $X_1 \wedge \neg(X_2 \vee \bot)$ sera représentée par l'expression OCaml `And (Var 1, Not (Or (Var 2, Bot)))`.

Q2. Adapter l'algorithme DFS pour implémenter la fonction suivante:

```
(* nombre_var f n renvoie un tableau donnant le nombre d'occurrences de chaque
   variable dans f, en supposant que les indices des variables
   sont tous dans {0, ..., n-1}. *)
nombre_var : formule -> int -> int array
```