TP13: OCaml impératif, tas binaire

MP2I Lycée Pierre de Fermat

Jusqu'ici, nous avons programmé en OCaml exclusivement à l'aide de fonctions récursives, et en raisonnant sur des objets eux-même récursifs comme les listes.
En réalité, OCaml dispose également d'éléments de programmation impérative, que nous allons étudier dans ce TP.

Références

Une **référence** OCaml est une case mémoire pouvant stocker un objet, et pouvant être modifiée. C'est un concept proche des pointeurs C.

Une référence contenant un objet de type `'a` est de type `'a ref`. La fonction `ref: 'a -> 'a ref` permet de créer une référence et de lui donner une valeur initiale:

```
let r = ref 32
```

On peut imaginer que le code ci-dessus s'exécute de manière analogue au code C suivant:

```
int* r = malloc(sizeof(int));
*r = 32;
```

Étant donné une référence `r`, on peut accéder à son contenu avec `!r`: C'est comme si l'on écrivait `*r` en C: on déréférence.
On peut aussi modifier le contenu d'une référence grâce à l'opérateur binaire `:=`, qui s'utilise comme suit:

```
let r = ref 0 ;;
r := 5 ;; (* change la valeur stockée dans r en 5 *)
r := !r + 1 ;; (* change la valeur stockée dans r en 6 *)
```

L'opérateur `:=` prend en paramètres une référence (à gauche) et une valeur (à droite). Il modifie le contenu de la référence, et renvoie `unit`. On peut donc enchaîner les `:=` comme on le ferait avec des print:

```
let r = ref 2 in
r := !r + 3;    (* r contient 5  *)
r := !r * !r;   (* r contient 25 *)
print_int !r    (* affiche 25    *)
```

Q1. Écrire une fonction récursive `ajout: int list -> int ref -> unit` telle que `ajout l r` ajoute au contenu de r tous les éléments de l. Par exemple:

```
let l = [1; 10]
let r = ref 6;;
ajout l r;;
print_int !r (* affiche 17 = 6 + 1 + 10 *)
```

Les références permettent de garder une mémoire persistant à travers les appels de fonction (comme les éléments stockés dans le tas en C). Nous allons utiliser les références pour écrire une fonction `numéroter: 'a list -> (int * 'a) list` permettant de numéroter les éléments d'une liste. Par exemple:

```
assert (numeroter ['A'; 'B'; 'C'] = [(0, 'A'); (1, 'B'); (2, 'C')])
```

Q2. Écrire une première version, sans références, passant par la fonction auxiliaire suivante:

```
(* Numérote les éléments de l à partir de l'indice i *)
let rec numeroter_a_partir_de (l: 'a list) (i: int) : (int * 'a) list = ...

let numeroter l = numeroter_a_partir_de l 0
```

Q3. Compléter le code suivant pour obtenir une autre version de numéroter, passant par les références:

```
let numeroter l =
  let i = ref 0 in (* indice actuel *)
  (* numérote ll depuis la valeur courante de i *)
  let rec numeroter_ref ll =
    ...
  in numeroter_ref l
```

Q4. Selon le même principe, écrire une fonction de numérotation dans l'ordre préfixe pour les arbres binaires, qui rajoute sur chaque nœud son numéro dans l'ordre préfixe:

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

num_prefixe : 'a arbre -> (int * 'a arbre)
```

On pourra utiliser une fonction auxiliaire et une référence stockant le numéro actuel de l'ordre préfixe en cours de construction.

Tableaux

Les **tableaux** OCaml sont presque identiques aux tableaux C: on peut accéder à n'importe quelle case en $\mathcal{O}(1)$ et, comme les références, on peut modifier leur contenu[1]. Un tableau est délimité par `[| ... |]`, et on accède aux éléments avec la syntaxe `t.(i)` qui est l'équivalent de `t[i]` en C. Par exemple:

```
let t = [|"bla"; "truc"; "tata"; "titi"|]
let x = t.(0) (* "bla"  *)
let x = t.(3) (* "titi" *)
```

Enfin, pour modifier une case de tableau, on utilise la syntaxe `<-`, similaire à ce que l'on a pu voir dans les algorithmes en pseudo-code pendant l'année:

```
let t = [|"bla"; "truc"; "tata"; "titi"|];;
t.(1) <- "bli"; (* remplace "truc" par "bli" *)
t.(2) <- "toto";;
```

Comme une liste, un tableau ne peut contenir que des éléments d'un seul type `'a`, et est alors de type `'a array`. Enfin, le module `Array` contient plusieurs fonctions permettant de manipuler des tableaux. Notamment:

```
(* Array.length t renvoie la longueur de t *)
Array.length: 'a array -> int

(* Array.make n x renvoie un tableau de taille n dont toutes les
   cases valent x *)
Array.make: int -> 'a -> 'a array
```

Q5. Écrire une fonction permettant de calculer la somme des éléments d'un tableau. On pourra passer par la fonction récursive auxiliaire suivante:

```
(* Renvoie la somme des éléments de t à partir de l'indice i inclus *)
let rec somme_a_partir_de (t: 'a array) (i: int) = ...
```

Q6. Écrire une fonction permettant de créer un tableau contenant les entiers de 1 à n (*On pourra à nouveau passer par une fonction auxiliaire adéquate*):

```
(* range n = [|1; 2; ...; n|] *)
let range (n: int) : int array = ...
```

Boucles

On voit que pour manipuler des tableaux avec des fonctions récursives, il faut systématiquement écrire des fonctions auxiliaires comme dans les deux questions précédentes, ce qui n'est pas très intuitif.
OCaml dispose de **boucles pour et tant-que**. La syntaxe pour les boucles pour est:

```
for i = DEBUT to FIN do (* les bornes DEBUT et FIN sont incluses *)
  CODE
done
```

où `DEBUT` et `FIN` sont des expressions de type `int` et `CODE` une expression de type `unit`.

Une boucle for est une **expression** OCaml, et a donc une valeur, qui est systématiquement `()`. On peut donc l'enchaîner avec des print, des affectations de références, etc...

[1] En fait, une référence peut être vue comme un tableau de taille 1, un peu comme en C !

Par exemple, pour calculer la factorielle:

```
let factorielle n =
  let res = ref 1 in
  for i = 1 to n do
    res := !res * i
  done;
  res
```

Q7. Écrire une autre fonction calculant la somme d'un tableau, sans récursivité, en utilisant des références et une boucle for.

Q8. Implémenter le tri par sélection sur les tableaux, sans récursivité.

Les boucles while fonctionnent sur le même principe, la syntaxe étant:

```
while CONDITION do
   CODE
done
```

Par exemple, voici une implémentation de l'exponentiation rapide:

```
(* renvoie a puissance n *)
let expo_rapide (a: float) (n: int) =
  let res = ref 1. in
  let x = ref a in
  let nn = ref n in
  while !nn > 0 do
    if !nn mod 2 = 1 then begin
      res := !x *. !res
    end;
    x := !x *. !x;
    nn := !nn / 2
  done;
  !res
```

Q9. Implémenter au choix un des deux algorithmes suivants en utilisant des boucles:

- Recherche par dichotomie
- Tri par insertion

Pour faire une boucle for descendante, il faut utiliser le mot-clé `downto` au lieu de `to`:

```
for i = 10 downto 0 do
  print_int i;
  print_string " "
done;
print_string "Bonne année !"
```

Q10. Écrire une fonction permettant de transformer une liste en un tableau, et une fonction faisant l'inverse.

Tas binaire

Implémentons les tas binaires vus en cours. On s'intéresse à des tas-min, où la racine contient l'élément minimal. On rappelle que dans l'implémentation par tableau, un tas est représenté par un tableau donnant ses éléments dans l'ordre du parcours en largeur. Alors, le nœud numéro i a pour enfants $2i+1$ et $2i+2$, et comme parent $\lfloor \frac{i-1}{2} \rfloor$ (sauf la racine, qui n'a pas de parent).

Q11. Implémenter une fonction ayant la spécification suivante:

```
(* est tas t i renvoie true si les i premières cases de t forment un tas *)
est_tas: 'a array -> int -> bool
```

Indication: on pourra passer par une boucle vérifiant si chaque nœud est bien supérieur à son parent, ou bien par une fonction auxiliaire récursive.

Dans la suite, on écrira "le tas (t, i)" pour dire "le tas stocké dans les i premières cases du tableau t"

Q12. Écrire une fonction permettant d'échanger deux cases d'un tableau:

```
(* échanger t i j échange les cases t.(i) et t.(j) *)
echanger: ('a array) -> int -> int -> unit
```

Q13. Écrire trois fonctions permettant d'accéder aux enfants et au parent d'un nœud:

```
parent: int -> int (* Précondition: l'entrée n'est pas la racine *)
gauche: int -> int
droite: int -> int
```

On rappelle que la procédure d'insertion consiste à placer le nouvel élément sur une nouvelle feuille, puis à faire **remonter** cet élément vers la racine jusqu'à ce qu'il soit bien placé, i.e. qu'il soit plus grand que son parent.

Q14. Écrire une fonction `inserer: 'a array -> int -> 'a -> unit` telle que `inserer t i x` insère x dans le tas (t, i). On rappelle que pour un tas de taille i, la prochaine feuille se place dans la case i de t.

En particulier, si t est un tableau quelconque, on peut le transformer en tas en:

- insérant $t[0]$ dans le tas $(t, 0)$;
- insérant $t[1]$ dans le tas $(t, 1)$;
- ...
- insérant $t[k]$ dans le tas (t, k);
- ...
- insérant $t[n-1]$ dans le tas $(t, n-1)$;

Q15. Écrire une fonction `tasifier: 'a array -> unit` transformant un tableau en un tas.

On rappelle l'algorithme utilisé pour extraire le min d'un tas ($n.e$ désigne l'étiquette du nœud n):

Algorithme 1 : Extraction de la racine

Entrée(s) : T un tas
Sortie(s) : e étiquette de la racine de T. La racine a été supprimée, et T reste un tas

```
1  r ← racine de T ;
2  e ← r.e // valeur à renvoyer
3  f ← dernière feuille de T ;
4  Échanger r.e et f.e;
5  Réduire la taille de T de 1;
6  tant que r.e est plus grande que l'étiquette de l'un des enfants de r faire
7      n ← l'enfant de r avec la plus petite étiquette;
8      Échanger r.e et n.e;
9      n ← f;
10 retourner e
```

Plutôt que d'écraser simplement la valeur à la racine, on l'**échange** avec la valeur de la dernière feuille. Cette méthode permet de conserver globalement les valeurs contenues dans le tableau, et, si le tas est de taille i, la valeur extraite est placée à la case i du tableau t utilisé pour stocker le tas.

Q16. Implémenter une fonction déterminant si un nœud est bien placé par rapport à ses éventuels enfants. Attention, un noeud peut avoir 0, 1 ou 2 enfants:

```
(* est_bien_place t i j renvoie true si le noeud d'indice j
   dans le tas (t, i) est plus petit que son ou ses enfants. *)
est_bien_place: 'a array -> int -> int -> bool
```

Q17. Implémenter une fonction qui, étant donné un nœud mal placé (et possédant donc au moins un enfant), renvoie l'indice de son enfant ayant la plus petite étiquette:

```
(* bon_enfant t i j renvoie l'enfant de j dans le tas (t, i)
   ayant la plus petite valeur.
   Précondition: j est un noeud ayant au moins un enfant dans (t, i)
   dont l'étiquette est plus petite. *)
bon_enfant: 'a array -> int -> int -> int
```

Q18. Écrire une fonction d'extraction pour les tas binaires. **Attention**, après avoir échangé la racine et la feuille, la taille du tas n'est plus i mais $i-1$:

```
(* Extrait le minimum du tas de taille i stocké dans t *)
let tas_extraire (t: 'a array) (i: int) : 'a = ...
```

On rappelle le principe du tri par tas. Pour trier un tableau T, dans un premier temps, on transforme T en tas, puis on extrait son minimum n fois. La procédure d'extraction plaçant le minimum du tas (T, i) à la case i de T, on obtient essentiellement une version plus efficace du tri par sélection, en $\mathcal{O}(n \log n)$!

Q19. A l'aide des fonctions précédentes, implémenter le tri par tas.