TP5: OCaml impératif

MPSI 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 l'on présente dans ce TP.

Références

Avec ce que l'on a vu en OCaml pour l'instant, rien (entier, flottant, liste, arbre) n'est modifiable. Il n'est donc pas possible a priori d'écrire une fonction ayant la spécification suivante:

```
(* ajoute 0 en tête de l *)
let modifier (l: int list) : unit = ...

let l = [1; 2; 3];;
modifier l ;;
assert (l = [0;1;2;3]);;
```

Une **référence** OCaml est une case mémoire pouvant stocker une valeur modifiable. Une référence contenant un objet de type `'a` est de type `'a ref`, et la fonction `ref: 'a -> 'a ref` permet de créer une référence et de lui donner une valeur initiale:

```
let r = ref 32
```

Étant donné une référence `r`, on peut accéder à son contenu avec `!r`: on dit que l'on déréférence. On peut aussi modifier le contenu d'une référence grâce à l'opérateur binaire `:=`:

```
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` modifie r pour lui sommer tous les éléments de l. Par exemple:

```
let l = [3; 11]
let r = ref 6;;
ajout l r;;
print_int !r (* affiche 20 = 6 + 3 + 11 *)
```

Les références permettent donc de garder une mémoire modifiable et persistante à travers les appels de fonction. Elles vont aussi être utilisées lorsque l'on utilise des boucles.

Boucles for

Bien que jusqu'ici nous ayons exclusivement utilisé des fonctions récursives pour programmer, OCaml dispose bien de boucles. Cependant, lorsque l'on manipule des objets récursifs comme les listes et les arbres, les fonctions récursives sont bien plus adaptées. Les boucles, que l'on présente ici, vont permettre de manipuler des objets comme les tableaux et les chaînes de caractères, qui ne sont pas intrinsèquement récursifs. La syntaxe pour les boucles **for** 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 de type `unit`, et a donc une valeur, qui est systématiquement `()`. On peut donc l'enchaîner avec des print, des affectations de références, etc...

Par exemple, pour calculer la factorielle:

```
let factorielle (n: int) : int =
  let res = ref 1 in
  for i = 1 to n do
    res := !res * i
  done;
  print_string "Fini de calculer";
  !res (* récupérer la valeur de la référence *)
```

Attention, contrairement à Python, on ne dispose pas de "return" en OCaml. Impossible d'arrêter une boucle au milieu de son exécution.

Q2. Écrire une fonction `est_premier: int -> bool` déterminant si un entier n donné est premier ou non (sans se soucier de l'efficacité). On rappelle que l'opérateur `mod` calcule des restes euclidiens: par exemple, `16 mod 7 = 2`.

Tableaux

En OCaml, les **tableaux** sont des objets fondamentalement différents des listes, mais assez proches des listes Python: on peut accéder à n'importe quelle case d'un tableau en $\mathcal{O}(1)$, pour en lire ou modifier le contenu. Un tableau est délimité par `[| ... |]`, et on accède aux éléments avec la syntaxe `t.(i)` qui est l'équivalent de `t[i]` en Python. 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 `<-`:

```
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`.

Le module `Array` contient plusieurs fonctions permettant de manipuler des tableaux. Vous devez connaître les deux suivantes et pouvoir les réutiliser:

```
(* 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
```

Par exemple:

```
(* range n = [|0; 1; ...; n-1|] *)
let range (n: int) : int array =
  let t = Array.make n 0 in
  for i = 0 to n-1 do
    t.(i) <- i
  done;
  t
```

Q3. Écrire une fonction `mem: 'a -> 'a array -> bool` telle que `mem x t` détermine si le tableau `t` contient `x` ou non.

Q4. Écrire une fonction prenant en entrée un tableau T de taille n et renvoyant un tableau S de taille $n+1$ tel que $S[i]$ contient la somme des i premières cases de T:

```
let sommes_partielles (t: int array) : int array = ...
```

Q5. Rendre la fonction précédente $\mathcal{O}(n)$ si ce n'est pas déjà le cas. *Indication: calculer $S[i]$ en fonction de $S[i-1]$.*

Comme en Python, un tableau OCaml n'est qu'une référence. Considérons le code suivant:

```
let t = [|1; 2; 3; 4; 5|];;
let s = t;;
```

Les variables `s` et `t` font référence au même tableau: on n'a pas créé une copie du tableau avec la deuxième ligne. En particulier, modifier `s` va aussi modifier `t`, et inversement:

```
t.(i) = 5000;;
assert (s.(i) = 5000);;
```

Il faut donc faire attention à ne pas créer de "fausses copies" de tableaux ! Par exemple, le code suivant va créer une matrice (i.e. un tableau de tableaux) où les lignes font toutes référence au même tableau, ce qui empêche de modifier la matrice comme on le voudrait:

```
(* renvoie une matrice nulle n x n *)
let matrice_nulle (n: int) : int array array =
  let ligne = Array.make n 0 in
  Array.make n ligne
```

Q6. Recopier la fonction ci-dessus, l'utiliser pour créer une matrice nulle 3×3, et observer ce qui arrive lorsque vous modifiez la case $(0, 0)$.

Q7. A l'aide d'une boucle `for`, corriger le code précédent pour que les lignes soient toutes indépendantes.

Q8. Écrire une fonction calculant la transposée d'une matrice, représentée comme un tableau de tableaux:

```
let transpose (t: int array array) : int array array
```

Boucles while

Les boucles while fonctionnent sur le même principe que les boucles for. La syntaxe générale est:

```
while CONDITION do (* CONDITION est une expression de type bool *)
   CODE         (* CODE est une expression de type unit  *)
done
```

Par exemple:

```
(* renvoie le plus grand entier k tel que k^2 <= n *)
let racine_entiere (n: int) : n =
  let k = ref 0 in
  while !k * !k <= n do
    k := !k + 1
  done;
  (* en sortie, k^2 > n, et (k-1)^2 <= n *)
  !k - 1
```

Q9. Améliorer la fonction `est_premier` pour qu'elle ne teste que les diviseurs inférieurs à $\sqrt{n}$ et pour qu'elle s'arrête dès qu'un diviseur est trouvé.

Q10. Implémenter le tri par insertion sur les tableaux:

Algorithme 1 : Tri insertion

Entrée(s) : T tableau de taille n
Sortie(s) : T a été modifié et est trié
`// Invariant:` T `trié entre 0 et` $i-1$ `inclus`

```
1 pour i = 1 à n − 1 faire
      // Insérer T[i] dans T[0], ..., T[i − 1]
2     j ← i;
3     tant que j > 0 ET T[j] < T[j − 1] faire
4         Échanger T[j] et T[j − 1];
5         j ← j − 1;
```

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 !"
```

Q11. (**Optionnel**) Écrire une fonction permettant de transformer une liste en un tableau, et une fonction faisant l'inverse.

Parcours d'arbre impératif

Nous avons vu en cours l'algorithme de parcours en largeur, qui permet d'énumérer les noeuds d'un arbre par niveaux: d'abord la racine, puis les noeuds de profondeur 1, puis ceux de profondeur 2, etc... Cet algorithme utilise une file, on rappelle qu'en OCaml le module `Queue` permet d'utiliser cette structure:

On rappelle aussi l'existence du module OCaml `Queue` qui implémente une structure de file mutable:

```
type 'a Queue.t (* file contenant des 'a *)

(* create () renvoie une file vide *)
create : unit -> 'a Queue.t

(* push x f enfile x dans f *)
val push: 'a -> 'a t -> unit

(* take f renvoie la tête de f et la défile. *)
val take : 'a t -> 'a

(* is_empty f renvoie true si f est vide, false sinon *)
is_empty : 'a t -> bool
```

Commençons par quelques fonctions pour se familiariser avec cette structure:

Q12. Écrire une fonction `enfile_tab: 'a array -> 'a Queue.t` prenant en entrée un tableau et renvoyant une file contenant tous ses éléments.

Q13. Écrire une fonction `vider: int Queue.t -> unit` prenant en entrée une file d'entiers, et affichant chacun de ses éléments sur une ligne, puis affichant `File vidée` à la fin de l'opération.

Passons au parcours d'arbre. On pose le type suivant pour les arbres binaires:

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

Dans l'algorithme de parcours, la file va contenir des noeuds de l'arbre à parcourir. Ce sera donc une file de type `'a ab Queue.t`.

Q14. Écrire une fonction auxiliaire `enfile_enfants: 'a ab -> 'a ab Queue.t -> unit` prenant en entrée un arbre A non vide et une file F, et ajoutant les enfants de A à F. Attention, seul les **noeuds** sont des enfants, les arbres vides ne comptent pas !

A partir de la fonction précédente, on peut implémenter comme suit une fonction qui **affiche** les noeuds d'un arbre binaire par ordre de parcours en largeur:

```
let affiche_largeur (a: int ab) : unit =
  if a = V then () else
  let f = Queue.create () in
  Queue.push a f;
  (* invariant: f ne contient que des N, aucun V *)
  while not (Queue.is_empty f) do
    let u = Queue.take f in (* défiler *)
    match u with
    | N(x, _, _) ->
      print_int x; print_newline (); (* afficher l'étiquette *)
      enfile_enfants u f (* enfiler les enfants *)
    | V -> failwith "ne doit pas arriver"
  done
```

Q15. Recopier la fonction `affiche_largeur`, et vérifier sur un exemple qu'elle est correcte.

Q16. En adaptant la fonction précédente, écrire une fonction `somme: int ab -> int` renvoyant la **somme** des étiquettes d'un arbre d'entiers. On pourra utiliser une variable de type `int ref` pour stocker la somme au fil du parcours.

Q17. Écrire une fonction `liste_largeur: 'a ab -> 'a list` renvoyant la **liste** des étiquettes d'un arbre d'entiers dans l'ordre du parcours en largeur. On pourra utiliser une `'a list ref` pour stocker les noeuds visités au fil du parcours.

On fixe $n \in \mathbb{N}$.
On considère des arbres binaires étiquetés par des entiers de $[\![0, n-1]\!]$. Étant donné a un tel arbre, on voudrait savoir combien de fois chaque étiquette $e \in [\![0, n-1]\!]$ apparaît dans a. Plus précisément, on cherche à écrire une fonction `occurrences: int -> int ab -> int array` telle que `occurrences n a` renvoie un tableau T tel que $T[i]$ est le nombre d'occurrences de i dans a.

Q18. En adaptant le parcours en largeur, implémenter `occurrences`.

Réduction de polymères

Chaînes de caractères OCaml En OCaml, pour accéder au i-ème caractère d'une chaîne s, on écrit `s.[i]`. On peut accéder à la longueur d'une chaîne via la fonction `String.length: string -> int`. Contrairement aux tableaux, une chaîne de caractère n'est pas modifiable.

Q19. Écrire une fonction prenant en entrée une chaîne de caractères et vérifiant si c'est un **palindrome**, i.e. si elle se lit de la même manière dans les deux sens.

Polymères On considère des chaînes moléculaires [1], constituées de différents éléments. Chaque élément a deux polarités possibles: positive et négative. On représente les éléments par des lettres, en utilisant les majuscules pour les éléments positifs, et les minuscules pour les éléments négatifs. On appelle une chaîne d'éléments un ***polymère***. Par exemple, **dabAcCaCBAcCcaDA** est un polymère.

Deux éléments de mêmes types mais de polarités opposées peuvent réagir entre eux pour disparaître. Par exemple, si **a** et **A** sont adjacents dans un polymère, ils s'annulent (l'ordre n'importe pas). On dit qu'un polymère est instable s'il contient des éléments pouvant réagir. Un polymère instable va donc se réduire en un autre polymère. On appelle ***forme stable*** d'un polymère le polymère obtenu après avoir fait toutes les réductions possible. On admet que tout polymère admet une forme stable, et que cette forme stable est unique: l'ordre des réductions n'importe pas. Par exemple **dabAcCaCBAcCcaDA** peut se réduire 3 fois avant d'atteindre sa forme stable:

$$\begin{aligned} \textbf{dabAcCaCBAcCcaDA} &\longmapsto \textbf{dabAcCaCBAcaDA} \\ \textbf{dabAcCaCBAcaDA} &\longmapsto \textbf{dabAaCBAcaDA} \\ \textbf{dabAaCBAcaDA} &\longmapsto \textbf{dabCBAcaDA} \end{aligned}$$

On souhaite mettre au point une fonction qui calcule efficacement la forme stable d'un polymère.

Q20. A la main, calculer la forme stable du polymère **AABbDeEdaBbC**.

[1] Le problème dans cette partie est tiré de: adventofcode.com/2018/day/5.

Algorithme naïf Le premier algorithme auquel on s'intéresse consiste à trouver **une** paire d'éléments pouvant réagir, à les supprimer, et à recommencer tant que c'est possible:

Algorithme 2 : `stable`(s)

Entrée(s) : $s = s_0 s_1 \dots s_{n-1}$ un polymère
Sortie(s) : Forme stable de s

$i \leftarrow$ indice tel que s_i et s_{i+1} sont de même types et de polarités opposées, `None` s'il n'en existe pas 0**si** $i =$ *None* **alors**
  retourner s
sinon
  $s' \leftarrow s_0 \dots s_{i-1} s_{i+2} \dots s_{n-1}$;
  retourner *stable*(s')

Gestion des polarités En OCaml, et dans la plupart des langages de programmation, chaque caractère est représenté en mémoire par un entier appelé son code ASCII. Par exemple, 'A' est le caractère numéro 65, 'C' le caractère numéro 67, et le retour à la ligne est le caractère numéro 10. Les fonctions `int_of_char` et `char_of_int` permettent de passer d'un caractère à son code et inversement.

Q21. Déterminer le code des caractères suivants:

- '$'
- 'M'
- 'm'
- l'espace

Q22. Écrire une fonction `oppose: char -> char` prenant en entrée un caractère correspondant à un élément (i.e. une lettre majuscule ou minuscule), et renvoyant le même élément de polarité opposée. Par exemple, `oppose 'M'` renverra 'm' et inversement.

Q23. En déduire une fonction `est_oppose: char -> char -> bool` prenant en entrée deux caractères, et renvoyant `true` s'ils correspondent au même élément, en étant de polarités opposées.

Q24. Écrire une fonction `trouve_reaction: string -> int` renvoyant un indice i tel que s_i et s_{i+1} sont opposés (ou -1 si aucun tel indice n'existe).

Q25. Chercher en ligne la documentation de la fonction `String.sub`, qui permet de construire une sous-chaîne d'une chaîne donnée. En l'utilisant, implémenter une fonction récursive `stable_naif: string -> string` calculant la forme stable d'un polymère. On rappelle que `^` est l'opérateur de concaténation des chaînes de caractères.

Q26. Quelle est la complexité de la fonction `stable_naif` ? Trouver une famille de polymères atteignant cette complexité.

Avec une pile On rappelle l'algorithme vu en cours permettant de déterminer si un mot est bien parenthésé:

Entrée(s) : $s = s_0 s_1 \dots s_{n-1}$ un mot sur { '[', '(', ']', ')' }
Sortie(s) : "Oui" si le mot est bien parenthésé, "Non" sinon

```
1  P ← pile_vide();
2  i ← 0;
3  tant que i < n faire
4      si s_i = '(' ou s_i = '[' alors
5          empiler(P, s_i);
6      sinon
7          si est_vide(P) alors
8              retourner Non ;
9          sinon
10             b ← dépiler(P);
11             si b et s_i ne correspondent pas alors
12                 retourner Non ;
13     i ← i + 1 ;
14 retourner est_vide(P);
```

Q27. Expliquer comment modifier cet algorithme pour qu'il prenne en entrée une chaîne de caractères représentant un polymère et calcule sa forme stable. Écrire le pseudo-code et l'exécuter sur un exemple.

Gestion des piles OCaml dispose d'un module `Stack` permettant de manipuler des piles **mutables**. Le type d'une pile contenant des éléments `'a` est `'a Stack.t` Ses opérations sont:

```
(* Stack.create () renvoie une pile vide *)
Stack.create: unit -> 'a Stack.t

(* Stack.pop p dépile le sommet de p (non vide) et le renvoie *)
Stack.pop: 'a Stack.t -> 'a

(* Stack.push x p empile x dans p *)
Stack.push: 'a -> 'a Stack.t -> unit

(* Stack.is_empty p renvoie true ssi p est vide *)
Stack.is_empty: 'a Stack.t -> bool

(** EXEMPLE **)
let test () =
  let p = Stack.create () in          (* P <- pile vide          *)
  Stack.push 5 p;                     (* empiler(P, 5)           *)
  Stack.push 6 p;                     (* empiler(P, 6)           *)
  assert (not (Stack.is_empty p));    (* P n'est pas vide        *)
  let x = Stack.pop p in assert (x = 6); (* x <- depiler(P); x vaut 6 *)
  let x = Stack.pop p in assert (x = 5); (* x <- depiler(P); x vaut 5 *)
  assert (Stack.is_empty p)           (* P est vide              *)
```

Q28. Écrire une fonction `stable: string -> string` prenant en entrée un polymère et calculant sa forme stable. On pourra commencer par calculer le résultat sous la forme d'une liste, que l'on retransformera ensuite en string par le biais d'une fonction auxiliaire.