

TP n°6 - Correction

Références

Le but de ce TP est de vous familiariser avec les références en OCaml. Les variables en OCaml ne sont pas mutables – il s’agit simplement de noms donnés à des valeurs d’expressions. Pour manipuler des “variables” au sens usuel de la programmation impérative (*eg.* à la C), on se sert de types OCaml de la forme `'a ref`, où `'a` peut être n’importe quel type. Voici par exemple la manière dont on peut définir une nouvelle référence vers un `char` :

```
let x = ref 'b';;  
val x : char ref = {contents = 'b'}
```

On peut ensuite accéder en lecture à la valeur référencée par `x` :

```
!x;;  
- : char = 'b'
```

On peut également modifier cette valeur (noter que cette modification est une *action*, de type `unit`) :

```
x := 'c';;  
- : unit = ()
```

Deux références peuvent être *partagées*, c’est-à-dire référencer la même valeur :

```
let y = x;;  
val y : char ref = {contents = 'c'}  
y := 'd';;  
- : unit = ()  
!y;;  
- : char = 'd'  
!x;;  
- : char = 'd'
```

Une pile LIFO

Le but de cette section est de mettre en œuvre la structure de données des piles (*stack* en anglais) regroupant leurs éléments dans une liste suivant le principe du "dernier arrivé, premier sorti", (*Last In, First Out* en anglais). La différence par rapport aux listes définissables en fonctionnel pur est que les primitives d’une pile peuvent modifier son état pendant leur exécution.

Par exemple, si `s` est une pile non vide, un appel de la forme `pop s` renverra l’élément en tête de `s` (comme `List.hd` appliqué à une liste non vide), mais en plus, cet appel modifiera `s` en retirant cet élément de la pile.

Exercice 1. Définir le type `'a stack` comme le type des références vers `'a list`. Écrire une fonction `create : unit -> 'a stack` renvoyant une nouvelle pile vide. Écrire une fonction `empty : 'a stack -> bool` renvoyant `true` si et seulement si son argument est une pile vide.

Correction :

```
type 'a stack = 'a list ref
```

```
exception Empty_stack
```

```
let create () : 'a stack = ref []
```

```
let empty (s : 'a stack) = !s = []
```

Exercice 2. Écrire une fonction `pop : 'a stack -> 'a` renvoyant, s'il existe, l'élément en tête d'une pile, et retirant cet élément de la pile. Par exemple, si `s` vaut `{contents = [4; 3; 2]}`, `pop s` renverra 4 et modifiera `s` en `{contents = [3; 2]}`. Que doit faire `pop` si la pile est vide?

Correction :

```
let pop (s : 'a stack) = match !s with  
  | [] -> raise Empty_stack  
  | h :: t -> s := t ; h
```

Exercice 3. Écrire une fonction `peek : 'a stack -> 'a` renvoyant, comme `pop`, le premier élément d'une pile, mais sans retirer cet élément de la pile.

Correction :

```
let peek (s : 'a stack) = match !s with  
  | [] -> raise Empty_stack  
  | h :: t -> h
```

Exercice 4. Écrire une fonction `push : 'a -> 'a stack -> unit` permettant d'ajouter un élément en tête d'une pile. Par exemple, si `s` vaut `contents = [4; 3; 2]`, `push 1 s` renverra `()` et modifiera la valeur de `s` en `{contents = [1; 4; 3; 2]}`.

Correction :

```
let push e (s : 'a stack) = s := (e :: !s)
```

Exercice 5. Écrire une fonction `is_equal : 'a stack -> 'a stack -> bool` renvoyant `true` si et seulement les deux piles passées en arguments contiennent la même suite d'éléments. Cette fonction peut s'écrire en moins de 50 caractères, espaces compris.

Correction :

```
let is_equal (s : 'a stack) (t : 'a stack) = (!s = !t);;
```

Exercice 6. Écrire une fonction `clone : 'a stack -> 'a stack` renvoyant une copie de la pile passée en argument. Attention, `clone s` doit créer une *nouvelle* pile contenant les mêmes éléments que `s`, et non partager la référence `s`.

Correction :

```
let clone (s : 'a stack) : 'a stack = ref (!s)
```

Exercice 7. Testez vos fonctions sur le code suivant, et vérifiez que le comportement est bien celui attendu.

```
let t1 = let t = create () in push 3 t; push 4 t; t;;  
let t2 = let t = create () in push 3 t; push 4 t; push 1 t; t;;
```

```
is_equal t1 t2;; (* false *)
```

```
is_equal t1 t1;; (* true *)
```

```
let _ = pop t2 in is_equal t1 t2;; (* true *)
```

```
let t3 = clone t2;;
```

```

i s_equal t2 t3;; (* true *)
let _ = pop t2 t3;; (* false *)

```

Exercice 8. Écrire la fonction `pop_n : int -> 'a stack -> 'a` qui, étant donné un entier `n > 0` et une pile `s`, renvoie le `n`-ième élément de `s` (la tête étant le premier élément) en retirant de `s` cet élément et tous ses prédécesseurs. Que doit faire `pop_n` si `s` contient moins de `n` éléments ?

Correction :

```

let pop_n n s =
  for i = 1 to n do
    pop s
  done ; s

```

Listes simplement chaînées

Une référence est un particulier d'*enregistrement à champ mutable*. Un enregistrement peut contenir à la fois des valeurs mutables et des valeurs non mutables. Vous trouverez dans l'exemple de l'implémentation des listes simplement chaînées, dont nous rappelons le schéma ci-dessous.

Un élément d'une liste simplement chaînée est représenté comme un enregistrement contenant une valeur (non-mutable) et une référence (potentiellement mutable) vers l'élément suivant. Une liste est représentée comme un enregistrement contenant une valeur et une référence vers le dernier élément de cette liste. Pour tenir compte du fait qu'un enregistrement ne peut pas être le dernier élément d'une liste n'a pas de successeur, nous utilisons un type OCaml

Exercice 9. Écrire une fonction `is-equal : 'a sllist -> 'a sllist -> bool` permettant de vérifier que deux listes sont égales (c.à-d. contiennent la même suite de valeurs).

Correction :

```
let is-equal l1 l2 =
  let rec aux e1 e2 =
    match (e1, e2) with
    | None, None -> true
    | (Some f1, Some f2) -> f1.value = f2.value && aux f1.next f2.next
    | -, - -> false
  in aux l1.first l2.first;;
```

Exercice 10. Écrire une fonction `map : ('a -> 'b) -> 'a sllist -> 'b sllist` se comportant comme `List.map`, mais opérant sur les listes simplement chaînées (testez votre code).

Correction :

```
let map f l1 =
  let l2 = create-sll () in
  let rec aux f e =
    match e with
    | None -> l2
    | Some e -> app-sll (f e.value) (aux f e.next)
  in aux f l1.first;;
```

Exercice 11. Écrire une fonction `from-list : 'a list -> 'a sllist` qui étant donnée une liste OCaml, renvoie une liste `sllist` contenant les mêmes valeurs dans le même ordre. Par exemple, `from-list [1; 2; 3]` renverra

```
Some
  {value = 1; next = Some {value = 2; next = Some {value = 3; next = None}}};
last = Some {value = 3; next = None}}
```

Correction :

```
let from-list l = (* transforme une liste OCAML vers une sll *)
  let l1 = create-sll () in
  let rec aux l =
    match l with
    | [] -> l1
    | h::t -> let l = aux t in app-sll h l
  in aux l;;
```

Exercice 12. Écrire une fonction `to-list: 'a sllist -> 'a list` effectuant la traitement inverse de la fonction `from-list`.

Correction :

```
let to-list l = (* transforme une sllist vers une liste OCAML *)
  let rec aux e =
    match e with
    | None -> []
    | Some e1 -> e1.value::(aux e1.next) in
  aux l.first;;
```