TP n°1 : Récursivité et effets de bord en Caml Light
Corrigé

Stéphane Glondu

11 novembre 2007

Exercice 1 (Suite de Fibonacci)

1. (a) Version naïve du calcul de F_n :

```
let rec fib1 n = match n with
  | 0 -> 0
  | 1 -> 1
  | n -> fib1 (n-2) + fib1 (n-1);;
```

Les parenthèses dans `(n-2)` et `(n-1)` sont **obligatoires** : la priorité des opérateurs est telle que `fib1 n-2` est compris comme `(fib1 n)-2`. Les priorités des opérateurs sont données dans le manuel, section *The core Caml Light language, Expressions*.

(b) On a $F_{10} = 55$, $F_{34} = 5\,702\,887$ et $F_{35} = 9\,227\,465$.

(c) Soit A_n le nombre d'additions effectuées par `fib1 n`. A_n peut être déterminé exactement en résolvant une relation de récurrence linéaire d'ordre deux :

$$A_n = \left(\frac{5+\sqrt{5}}{10}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{5-\sqrt{5}}{10}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n - 1$$

Pour n grand, $n \approx 0,72 \times 1,62^n$.

2. (a) Version linéaire du calcul de F_n :

```
let fib2 n =
  let rec aux n =
    if n = 1 then
      (0, 1)
    else
      let (a, b) = aux (n-1) in
      (b, a+b)
  in
  if n = 0 then 0 else snd (aux n);;
```

Ici, on fait appel à une fonction auxiliaire `aux` qui, avec l'argument n, renvoie renvoie le couple (F_{n-1}, F_n). Ainsi, un seul appel récursif suffit pour calculer (F_n, F_{n+1}) à partir de (F_{n-1}, F_n). La fonction `aux` telle qu'écrite ci-dessus n'est pas accessible depuis l'extérieur de la fonction `fib2`. Il est à noter que `fib2` n'est pas elle-même récursive. Il faut bien faire attention au cas $n = 0$!

(b) En notant B_n le nombre d'additions effectuées par `fib2 n`, on a $B_n = n-1$ pour $n > 0$.

(c) La valeur renvoyée par `fib2 44` est `701408733` (ce qui est bien la valeur de F_{44}), et `fib2 45` renvoie `-1012580478`, qui n'est pas la même chose que F_{45} ! Ce résultat est dû à la représentation interne des `int` en Caml, ici sur 31 bits : tous les entiers de type `int` sont compris entre -2^{30} et $2^{30} - 1$ et toutes les opérations sont réalisées modulo 2^{31}, de sorte que $(2^{30} - 1) + 1 = -2^{30}$! La plage des `int` dépend de la machine ; le plus petit entier est noté `min_int` dans la bibliothèque standard, et le plus grand est noté `max_int`.

3. Ici, il était suggéré de représenter un entier de précision arbitraire (encore appelé *grand nombre*) par la liste renversée (unités en premier) de ses chiffres en base 10.

   (a) `let base = 10;;`

   ```
   let rec plus m n =
     match (m, n) with
       | ([], n) -> n
       | (m, []) -> m
       | (m0::m, n0::n) ->
   let a = m0+n0 in
   if a >= base then
     (a mod base)::(plus m (plus n [a/base]))
   else
     a::(plus m n);;
   ```

   Le choix de représenter les grands nombres « à l'envers » permet de simplifier considérablement les fonctions travaillant dessus. Bien sûr, il existe de nombreuses façons d'écrire `plus` et celle donnée ci-dessus est donnée à titre d'exemple. Elle utilise l'algorithme vu à l'école primaire. Il faut bien faire attention à la gestion des retenues et des zéros significatifs !

   (b) Le corps de `fib3` est exactement le même que celui de `fib2`, en remplaçant les appels à l'opérateur standard `+` par la fonction `plus` ci-dessus et en adaptant en conséquence :

   ```
   let fib3 n =
     let rec aux n =
       if n = 1 then
         ([], [1])
       else
         let (a, b) = aux (n-1) in
         (b, plus a b)
     in
     if n = 0 then [] else snd (aux n);;
   ```

   (c) On peut compter le nombre de zéros significatifs d'un grand nombre par exemple avec la fonction suivante :

   ```
   let compte n =
     let rec aux accu = function
       | [] -> accu
       | a::q -> aux ((if a = 0 then 1 else 0) + accu) q
     in aux 0 n;;
   ```

   Attention, on suppose que tous les zéros de n sont significatifs ! La réponse à la question est renvoyée par `compte (fib3 5000)` (c'est 107).

4. Ici, une implémentation similaire à la question 2 ne suffit pas, car chaque appel récursif prend de la place en mémoire (les valeurs de toutes les variables locales). Il faut faire un appel *récursif terminal*, c'est-à-dire que le résultat de l'appel récursif soit directement retourné par le code appelant :

   ```
   let g n =
     let base = n+1 in
     let rec aux n a b =
       if n = 1 then b else aux (n-1) b ((a+b) mod base)
     in
     if n = 0 then 0 else aux n 0 1;;
   ```

   Caml reconnaît les appels terminaux et les optimise, de telle sorte que le code ci-dessus est aussi efficace qu'une version impérative (utilisant des références et des boucles `while` ou `for`) de la même fonction. La fonction auxiliaire de `compte` ci-dessus est aussi récursive terminale.

Dans l'exemple ci-dessus, `n` est le « vrai » argument de la fonction, `a` est G_{n-1} et `b` est G_n. On a $G_{10\,000} = 2\,281$, $G_{100\,000} = 55$ et $G_{1\,000\,000} = 960\,308$.

Exercice 2 (Effets de bords)

1. Il est bien important de comprendre parfaitement ce qui se passe dans les exemples suivants, car l'incompréhension des phénomènes illustrés est source de nombreux bugs difficiles à localiser et à corriger ! Les lignes commençant par # correspondent (sans le #) à ce qui est envoyé à Caml.

```
# let a = ref 3 and b = ref (1+2) in
  (a = b, a == b);;
- : bool * bool = true, false

# let v = make_vect 3 "Bonjour ?";;
v : string vect = [|"Bonjour ?"; "Bonjour ?"; "Bonjour ?"|]

# v.(0).[8] <- `!`; v;;
- : string vect = [|"Bonjour !"; "Bonjour !"; "Bonjour !"|]

# let v2 = copy_vect v in
  v.(1).[0] <- `b`; v2, v;;
- : string vect * string vect =
 [|"bonjour !"; "bonjour !"; "bonjour !"|],
 [|"bonjour !"; "bonjour !"; "bonjour !"|]

# v.(2) <- "Hello?";;
- : unit = ()

# let v3 = concat_vect v v;;
v3 : string vect =
 [|"bonjour !"; "bonjour !"; "Hello?"; "bonjour !"; "bonjour !";
   "Hello?"|]

# v.(2).[5] <- `!`; v3;;
- : string vect =
 [|"bonjour !"; "bonjour !"; "Hello!"; "bonjour !"; "bonjour !";
   "Hello!"|]
```

2. Il est toujours possible de définir `length` récursivement sans utiliser le mot-clé `rec` :

```
let length l =
  let f = ref (function _ -> 0) in
  f := (function
    | [] -> 0
    | a::l -> 1 + (!f l));
  !f l;;
```

Cet exemple illustre un aspect important de Caml : les fonctions sont des valeurs comme les autres (on dit aussi valeurs de *première classe*), et peuvent en particulier être utilisées dans des références.

Exercice 3 (Fonctionnelles polymorphes)

1. La fonction `it_list` est standard dans Caml Light :

```
let rec it_list f a bl =
  match bl with
    | [] -> a
    | b::bl -> it_list f (f a b) bl;;
```

2. La fonction `it_list` permet d'écrire rapidement l'application d'une fonction binaire à une liste d'arguments (en associant à gauche — la fonction `list_it` est similaire, mais associe à droite). Le deuxième argument correspond à l'élément de base ou l'élément neutre de la fonction. Dans les exemples demandés, il y a l'addition, le ou exclusif et une fonction un peu plus compliquée :

```
let somme = it_list (fun x y -> x+y) 0;;
let parity = it_list (fun x y -> if x then not y else y) false;;
let pairs = it_list
  (fun res a -> if a mod 2 = 0 then a::res else res) [];;
```

Exercice 4

1. ```
   let compteur =
     let valeur = ref 0 in
     fun () -> incr valeur; !valeur;;
   ```
   `valeur` est une référence vers un entier qui sera unique pendant toute la durée de vie du programme (elle n'est initialisée qu'une seule fois), et qui sera incrémentée à chaque appel. Cette implémentation est préférable à l'utilisation d'une variable globale (`let ... ;;` plutôt que `let ... in ...`) car `valeur` n'est accessible qu'à partir du corps de `compteur` (une référence globale pourrait être modifiée par n'importe quelle autre fonction du programme).

2. Un programme qui affiche son propre code source est appelé *quine* dans la littérature. L'exemple ci-dessous utilise la fonction `string_for_read` de la bibliothèque standard, qui remplace les caractères spéciaux par des *séquences d'échappement*. Par caractère spécial, on entend caractère qui ne peut pas apparaître directement dans le code source d'une chaîne. Une séquence d'échappement est une suite de caractères qui représente autre chose que ces caractères. Par exemple, `"` — qui marque ordinairement la fin d'une chaîne — est remplacé par `\"`. `\n` est une séquence d'échappement qui représente un retour à la ligne.

```
(fun s -> print_string s; print_string (string_for_read s); print_endline "\";;")
  "(fun s -> print_string s; print_string (string_for_read s); print_endline \"\\\";;\")\n  \"";;
```
```