Université Pierre et Marie Curie
Formation permanente

– o –

Prise en Main d'Objective Caml

Travaux encadrés sur machines - J1

Emmanuel Chailloux Pascal Manoury Bruno Pagano Michel Mauny

Février 2006

Boucle d'interaction

Lancer la boucle d'interaction (le *toplevel*) en tapant la commande `ocaml` A l'écran doit s'afficher

```
        Objective Caml version 3.0x

#
```

On quitte la boucle d'interaction par la commande ocaml : `# quit ;;`
On peut charger, typer et compiler un fichier le source `myfile.ml` par la commande `#use "myfile.ml";;`
On peut également évaluer directement des expressions : celles-ci sont compilées, typées puis évaluées.

```
# let x = 25 in x*x;;
```

La boucle affiche alors le type et la valeur de l'expression

```
- : int = 625
#
```

Il en va de même pour les déclarations. Voici, par exemple une fonction qui calcule la sous liste des éléments inférieurs à un élément donné

```
# let rec get_inf x0 xs =
  match xs with
   [] -> []
  |x::xs'
   -> if x < x0 then x::(get_inf x0 xs')
      else get_inf x0 xs'
  ;;
val get_inf : 'a -> 'a list -> 'a list = <fun>
#
```

On peut alors effectuer quelques tests

```
# get_inf 3 [4; 6; 1; 0; 7; 8; 2; 4; 3; 5];;
- : int list = [1; 0; 2]
# get_inf 25 [4; 6; 1; 0; 7; 8; 2; 4; 3; 5];;
- : int list = [4; 6; 1; 0; 7; 8; 2; 4; 3; 5]
# get_inf (-1) [4; 6; 1; 0; 7; 8; 2; 4; 3; 5];;
- : int list = []
# get_inf 'd' ['a'; 'z'; 'e'; 'b'; 'q'; 'c'; 'o'];;
- : char list = ['a'; 'b'; 'c']
# get_inf 'a' ['a'; 'z'; 'e'; 'b'; 'q'; 'c'; 'o'];;
- : char list = []
```

1 Premier jour : noyau fonctionnel

1.1 Récurrence numérique

Exercice 1.1

L'élévation de b à la puissance n se définit récursivement par itération du produit $\begin{cases} b^0 & = & 1 \\ b^{n+1} & = & b.b^n \end{cases}$

Question 1.1 Définir la fonction `slow_exp : int -> int -> int` telle que `(slow_exp b n)` calcule $\texttt{b}^{\texttt{n}}$ en utilisant les égalités ci-dessus.

Question 1.2 Définir une version *récursive terminale* de cette fonction en utilisant une fonction auxiliaire `slow_exp_loop : int -> int -> int -> int` telle que `(slow_exp_loop b n a)` calcule $\texttt{a}.\texttt{b}^{\texttt{n}}$.

Exercice 1.2

On va calculer b^n en utilisant les égalités suivantes : $\begin{cases} b^n & = & (b^{n/2})^2 & \text{si } n \text{ est pair} \\ b^n & = & b.b^{n-1} & \text{sinon} \end{cases}$

Question 2.1 Définir la fonction `even : int -> bool` telle que `(even n)` vaut `true` si `n` est pair et `false` sinon.

Question 2.2 Définir la fonction `fast_exp : int -> int -> int` telle que `(fast_exp b n)` calcule $\texttt{b}^{\texttt{n}}$ en utilisant les égalités ci-dessus.

Exercice 1.3

Calcul du plus grand diviseur commun (*gcd*) de deux entiers selon l'agorithme d'Euclide. On utilise $gcd(n, m) = gcd(m, r)$ avec $r = n \bmod m$.

Question 3.1 Définir la fonction `gcd : int -> int -> int` telle que `(gcd n m)` calcule le plus grand diviseur commun de `n` et `m` selon l'algorithme d'Euclide.

1.2 Rationnels

L'ensembles des nombres rationnels est l'ensemble des nombres que l'on peut écrire sous la forme d'une fraction $\frac{n}{p}$ où n et p sont des entiers.

Exercice 1.4

On peut donc représenter un nombre rationnel comme un couple de deux entiers ou un enregistrement encapsulant deux entiers et définir les opérations arithmétiques sur une telle donnée.

Question 4.1 Déclarer le type `rat` comme un enregistrement contenant un champ `num` et un champ `den` représentant respectivement le numérateur et le dénominateur d'une fraction.

Question 4.2 Définir une valeur de type `rat` représentant le rationnel 1.75. Cette représentation est-elle unique ? Donnez un contre-exemple.

Question 4.3 Définir une fonction `rat_reduce : rat -> rat` telle que si `r` $= \frac{n}{p}$ et si $a = gcd(n, p)$ alors `(rat_reduce r1)` $= \frac{n/a}{p/a}$.

Question 4.4 Définir les fonctions `rat_plus : rat -> rat -> rat` et `rat_mul : rat -> rat -> rat` calculant respectivement la somme et le produit de deux rationnels.

Question 4.5 Comprenez pourquoi l'opérateur prédéfini `=` n'est pas ce qu'il faut pour tester l'égalité de deux rationnels. Définir la fonction `rat_equal : rat -> rat -> bool` qui teste cette égalité.

1.3 Récurrence, fonctionnelles d'itération

Exercice 1.5

La célèbre suite de Fibonacci est définie par les équations récursives

$$F_0 = 1 \qquad F_1 = 1 \qquad F_{n+2} = F_{n+1} + F_n$$

On considère la suite de couples

$$C_0 = (F_0, 0) \qquad C_1 = (F_1, F_0) \qquad C_2 = (F_2, F_1) \qquad \ldots \qquad C_n = (C_n, C_{n-1} \qquad \ldots$$

On remarque que, comme $C_n = (F_n, F_{n-1})$, on a : $C_{n+1} = (F_n + F_{n-1}, F_n) = (F_{n+1}, F_n)$. Soit alors la fonction ϕ telle que $\phi(x, y) = (x + y, x)$, on a $\phi^{n+1}(C_0) = (Fn + 1, F_n)$.

Question 5.1 Écrire l'itérateur `iter` de type `'a -> ('a -> 'a) -> int -> 'a` tel que `iter a f n` $\equiv$ `f`^{n}`(a)`.

Question 5.2 Utiliser cet itérateur pour écrire une fonction qui calcule le n-ième terme de la suite de Fibonacci.

Question 5.3 Comment utiliser l'itérateur `iter` pour (re)définir la focntion `slow_exp` de l'exercice 1.1 ?

Exercice 1.6

L'itérateur que nous avons vu fait toujours décroître l'entier d'itération de 1. On peut passer en argument de l'itérateur une fonction de décrément. Si f est la fonction à itérer, a la valeur en 0 et δ le décrément, on définit l'itérateur $\mathcal{I}$ par les équations $\begin{cases} \mathcal{I}(\delta, f, a, 0) & = & a \\ \mathcal{I}(\delta, f, a, n+1) & = & f(n, \mathcal{I}(\delta, f, a, \delta(n))) \end{cases}$

On peut utiliser cet itérateur pour calculer, par dichotomie, la n-ième puissance d'un entier e (voir exercice 1.1).

Soit ϕ telle que $\begin{cases} \phi(n, r) & = & r \times r & \text{si } n \text{ est pair} \\ \phi(n, r) & = & e \times r & \text{sinon} \end{cases}$

On a que $e^n = \phi(n, \phi(\delta(n), \phi(\delta^2(n), \ldots, \phi(1, 1) \ldots)))$

Question 6.1 Écrire une fonction `gen_rec` de type qui implante l'itérateur $\mathcal{I}$. Notez le type obtenu.

Question 6.2 Écrire une fonction `phi` qui implante la fonction ϕ avec $e = 2$.

Question 6.3 En déduire une fonction `puiss2`, de type `int -> int`, qui calcule la n-ième puissance de 2 (pour $n > 1$).

Question 6.4 Utiliser l'itérateur `gen_rec` pour (re) définir la fonction d'exponentiation `fast_exp`.

1.4 Les listes

Exercice 1.7

Mise en jambe.

Question 7.1 Écrire une fonction `supprime` qui, étant donnés une liste `l` et un entier `n`, renvoie la liste `l` privée de ses `n` premiers éléments et qui signale une erreur lorsque la liste `l` comporte moins de `n` éléments où `n` est strictement négatif.

Question 7.2 Définir une fonction `long_prefixe` qui rend le nombre d'éléments identiques consécutivement au début d'une liste `l`.

Exercice 1.8

Exercice "académique" : les membres d'une université peuvent être : soit des étudiants, soit des administratifs, soit des enseignants-chercheurs. Un étudiant est caractérisé par son nom et son numéro d'inscription. Un administratif est caractérisé par son nom et sa catégorie administrative, qui peut être A, B ou C. Un enseignant-chercheur est caractérisé par son nom et le numéro de l'UFR à laquelle il est rattaché. On souhaite établir un fichier des membres d'une université afin de pouvoir, par exemple, envoyer des courriers bien adaptés à chaque catégorie de personnel.

Question 8.1 En définissant éventuellement des types intermédiaires, définissez un type `univ` pour représenter une université.

Question 8.2 Écrivez une fonction `separe` qui prend une valeur de type `univ` en argument et rend un triplet dont la première composante est la liste des étudiants de l'université, la seconde la liste des administratifs, la troisième la liste des enseignants-chercheurs.

Exercice 1.9

Soit une pièce de monnaie dont on ne sait comment elle est truquée. En la lançant un certain nombre de fois, on obtient une liste de symboles `Pile` ou `Face`. On va calculer une liste de booléens aléatoires équiprobables à partir d'une telle donnée.

Question 9.1 Définir un type `piece` qui donne les symboles `Pile` et `Face`.

Question 9.2 Soit une "`piece list`". On se propose d'en extraire une liste de booléens selon le principe suivant : on considère les symboles deux par deux ; si la paire considérée est `Pile Face` alors on produit `true` ; si la paire considérée est `Face Pile` alors on produit `false` ; sinon on ne produit rien.
Écrire la définition de la fonction `alea1 : piece list -> bool list` qui implante ce principe. Pensez à utiliser les ressources du filtrage.

Question 9.3 On va maintenant calculer le *reste* d'une "`piece list`" en ne retenant que les paires répétées. Par exemple, de `[Pile ; Face ; Pile ; Pile ; Face ; Face ; Face ; Pile ; Face ; Pile]` on retient la liste `[Pile ; Face]`. Le `Pile` du résultat vient des 3-ième et 4-ième `Pile` de la liste originale et le `Face` du résultat vient des 3-ième et 4-ième `face` de la liste originale.
Donner la définition de la fonction `reste_alea1 : piece list -> piece list` qui met en œuvre ce calcul du *reste*.

Question 9.4 On va renforcer le calcul de la fonction `alea1` en utilisant le *reste* obtenu après un premier passage.
Définir le fonction `alea : piece list -> bool list` qui utilise `alea1` pour calculer un début de liste de booléens puis de la compléte par itération (d'elle même) sur le reste obtenu.

Exercice 1.10

Listes et fonctionnelles d'itération.

Question 10.1 Quel fonctionnelle utiliser pour redéfinir la fonction `get_inf` donnée en introduction (voir) ? Faites le.

Question 10.2 Même question avec la fonction `separe` de l'exercice précédent.

Exercice 1.11

On veut réaliser une fonction de tri par insertion pour les listes. Si l'on a une fonction d'insertion `ins`, de type `'a -> 'a list -> 'a list` telle que

$$\texttt{ins x [}x_1\texttt{ ; ... ; }x_i\texttt{ ; }x_{i+1}\texttt{ ; ... ; }x_n\texttt{]} \equiv \texttt{[}x_1\texttt{ ; ... ; }x_i\texttt{ ; x ; }x_{i+1}\texttt{ ; ... ; }x_n\texttt{]}$$

avec $x_1 \leq \texttt{dots} \leq x_i \leq \texttt{x}$ et $\texttt{x} < x_{i+1}$, alors une fonction de tri `sort` sera telle que

$$\texttt{sort [}x_1\texttt{ ; }x_2\texttt{ ; ... ; }x_n\texttt{]} \equiv \texttt{ins }x_1\texttt{ (ins }x_2\texttt{ (...(ins }x_n\texttt{ [])...))}$$

Question 11.1 Écrire la fonction `ins`.

Question 11.2 En déduire, en utilisant un itérateur sur les listes, la fonction de tri `sort`.

Question 11.3 Donnez une seconde version de `sort`, en utilisant *l'autre* itérateur.

1.5 Tris par arbres

Exercice 1.12

On peut donner une version fonctionnelle du tri rapide en passant par une structure intermédiaire d'*arbre binaire de recherche* (un `abr`). Un arbre binaire de recherche est un arbre dont les nœuds sont étiquetés et qui ont la propriété que l'étiquette de la racine est plus grande que les étiquettes contenues dans le sous-arbre gauche et plus petite que celles contenues dans le sous-arbre droit. Cette propriété étant récursivement satisfaite par les deux sous-arbres.
L'idée de la fonction de tri appliquée à une liste est de construire la structure d'arbre binaire de recherche pour les éléments de cette liste puis d'extraire de l'arbre obtenu la liste triée.

Question 12.1 Définir un type `'a bin_tree` pour représenter les arbres binaires dont les nœuds sont étiquetés par une valeur de type `'a`.

Question 12.2 Écrire la définition de la fonction `ins_abr : ('a -> 'a -> bool) -> 'a -> 'a bin_tree -> 'a bin_tree` telle que `(ins_abr r x t)` calcule l'arbre binaire de recherche obtenu en insérant `x` à sa place dans `t` selon la relation `r`.
Notez que le résultat donne un arbre binaire de recherche seulement si `t` en est déjà un.

Question 12.3 Écrire la définition de la fonction `abr_of_list : ('a -> 'a -> bool) -> 'a list -> 'a bin_tree` telle que `(abr_of_list r xs)` soit un arbre binaire de recherche selon `r`, qui contient les éléments de `xs`.

Question 12.4 Écrire le définition de la fonction `list_of_abr : 'a bin_tree -> 'a list` telle que `(list_of_abr t)` renvoie la liste triée, selon une relation `r` quelconque, des étiquettes de `t` si `t` est un arbre binaire de recherche selon `r`.

Question 12.5 En déduire une fonction de tri pour les listes.

1.6 Expressions et calculs symboliques

Les types somme et le filtrage facilitent grandement l'écriture de fonctions de manipulation symbolique. On se propose d'illustrer cela sur un petit programme de dérivation formelle d'expressions algébriques.

Exercice 1.13

On considère un petit langage d'expression pour les polynômes à puissance et coefficients entiers. Plus précisément l'ensemble des expressions est récursivement défini par :
– les constantes sont des expressions ;
– les variables sont des expressions ;
– sit e est une expression et n un entier alors e^n est une expression ;
– si e_1 et e_2 sont des expressions alors $e_1 \times e_2$ et $e_1 + e_2$ sont des expressions.

Les règles de dérivation formelles en x sont :

$$\frac{dc}{dx} = 0 \qquad \text{si } c \text{ est une constante}$$

$$\frac{dx}{dx} = 1$$

$$\frac{dy}{dx} = 0 \qquad \text{si } y \text{ est une variable différente de } x$$

$$\frac{de^n}{dx} = n \times e^{n-1} \times \frac{de}{dx}$$

$$\frac{d(e_1+e_2)}{dx} = \frac{de_1}{dx} + \frac{de_2}{dx}$$

$$\frac{d(e_1 \times e_2)}{dx} = (e_1 \times \frac{de_2}{dx}) + (e_2 \times \frac{de_1}{dx})$$

Question 13.1 Définir un type `exp` pour nos expressions.

Question 13.2 Définir la fonction `deriv` qui calcule la dérivée formelle d'une expression pour une variable,