TP2 : Récursivité, listes

MPSI Lycée Pierre de Fermat

Exercice 1: Définitions basiques

Créez un fichier `exercice1.ml`, et écrivez-y le code suivant :

```
(* Renvoie la somme de x et y *)
let add (x: int) (y: int) : int =
  x + y

let plus_7 = add 7

let a =
  let x = 3 in
  plus_7 a
```

Dans un terminal, lancez utop, et tapez-y la ligne suivante pour importer le code du fichier (si vous codez en ligne, vous pouvez cliquer sur “Évaluer le code”, ou utiliser le raccourci Ctrl-E) :

```
#use "exercice1.ml";;
```

Cette instruction devrait exécuter tout le code du fichier, et ajouter les objets définis avec des “let” au contexte global. Dans le reste de l'exercice, à chaque fois que vous définissez une nouvelle fonction, vous pouvez relancer la commande `#use "exercice1.ml";;` pour pouvoir l'importer dans utop.

Q1. Définir une fonction `appliquer: 'a -> ('a -> 'b)-> 'b` prenant en entrée deux paramètres x et f, et renvoyant $f(x)$.

Q2. Que représente `appliquer 0` ?

Q3. Définir une fonction `first: ('a * 'b)-> 'a` prenant en entrée un couple et renvoyant sa première composante.

On souhaite écrire une fonction `flip` permettant d'échanger l'ordre des arguments d'une fonction à deux paramètres. Par exemple, si l'on dispose d'une fonction `moins` telle que `moins a b = a-b`, on voudrait que la fonction `flip moins` soit telle que : `(flip moins)a b = b - a = moins b a`.

Q4. Quel sera le type de `flip` ?

Q5. Implémenter `flip`, et la tester.

Les fonctions `first` et `flip` existent par défaut en OCaml, elles s'appellent `fst` et `Fun.flip`.

Exercice 2: If-else, Match-with

Q1. En utilisant des if-else, écrire une fonction `n_roots: (float * float * float)-> int` qui prend en entrée un triplet (a, b, c) et calcule le nombre de racines réelles distinctes du polynôme $aX^2 + bX + c$.

Q2. En utilisant un match-with, écrire une fonction qui prend en entrée un entier n, et renvoie :

- "A" si n est divisible par 3 ;
- "B" si n est divisible par 5 et pas par 3 ;
- "C" si n n'est divisible ni par 5 ni par 3.

Indication : on pourra faire un match-with sur le couple $(n \bmod 3, n \bmod 5)$.

On veut maintenant écrire une fonction qui prend en entrée deux entiers x et y et qui :

- Si x vaut $\pm y$, renvoie 0
- Si $x \in \{y+1, y-1\}$, renvoie $(x+y)^2 + 1$
- Si $x + y \in \{1, -1\}$, renvoie $(x-y)^2 - 1$
- Sinon, renvoie $x * y$

Q3. Compléter le code suivant pour implémenter la fonction décrite :

```
let g x y = match (x-y, x+y) with
  | 0, _ -> 0
  | _, 0 -> (* A COMPLETER *)
  | 1, z -> (* A COMPLETER *)
  | (* A COMPLETER *)
  | (* A COMPLETER *)
  | (* A COMPLETER *)
  | _    -> (* A COMPLETER *)
```

Exercice 3: Affichage, type unit

OCaml dispose d'un dernier type de base, appelé `unit`, qui ne contient qu'un seul élément, noté `()` (et appelé "unit"). Ce type permet de représenter les fonctions qui ne **renvoient rien**. A titre de comparaison, regardons les deux fonctions Python suivantes :

```
def f(x, y):
  return x * y
```

```
def f(x, y):
  print("Le produit vaut: ", x*y)
```

Celle de gauche n'affiche rien mais renvoie le produit xy, tandis que celle de droite affiche une phrase et ne renvoie rien. Les fonctions OCaml devant être des fonctions au sens mathématique, elles doivent renvoyer quelque chose. Les fonctions d'affichage OCaml auront donc comme type de retour `unit`. C'est le cas des fonctions préexistantes suivantes :

```
print_int : int -> unit    (* print_int n affiche n *)
print_string : string -> unit (* print_string s affiche s *)
print_endline : string -> unit (* print_endline s affiche s puis un retour ligne *)
```

Ainsi, l'expression `print_int (1+2)` a comme valeur `()`, mais son calcul entraîne l'affichage de l'entier 3 dans le terminal.

Les expression OCaml de type `unit` peuvent être vues comme des **instructions** au sens des langages impératifs comme Python. On peut enchaîner deux instructions en les séparant d'un point-virgule :

```
(* Affiche s deux fois d'affilée *)
let affiche_deux_fois (s: string) : unit =
  print_string s;
  print_string s
```

Q1. Écrire une fonction qui prend en entrée un prénom, et affiche une salutation :

```
utop # saluer "Philippe";;
Bonjour, Philippe !
- : unit = ()
```

La fonction `assert: bool -> unit` prend en entrée un booléen, et :

- Si le booléen est vrai, ne fait rien ;
- Si le booléen est faux, arrête le programme avec une erreur.

Cette fonction permet de faire des tests automatiques :

```
let est_premier (n: int): bool = ...

let test () =
  assert (est_premier 17);
  assert (not (est_premier 14));
  assert (est_premier 000000007);
  print_endline "Les tests ont réussi !"
```

Q2. Écrire deux fonctions de test pour les fonctions des Q1 et Q2 de l'exercice 2.

Dans la suite du TP (et de l'année), on prendra l'habitude d'écrire des fonctions de test pour ne pas avoir à réécrire les tests à la main dans utop à chaque modification du code.

Exercice 4: Récursivité

Q1. Écrire une fonction `prod: float -> int -> float` qui calcule le produit d'un flottant par un entier, sans utiliser l'opérateur `*.`.

Q2. Écrire une fonction `reste: int -> int -> int` qui calcule le reste de la division euclidienne d'un entier a par un autre b, sans utiliser `mod` ou `/`. On supposera $a \geq 0$ et $b > 0$.

Q3. Modifier la fonction précédente, pour qu'elle renvoie plutôt un couple (q, r), où q est le quotient et r le reste.

Q4. Écrire une fonction `decomp: int -> unit` telle que `decomp x` affiche les chiffres de x un par un sur une ligne chacun. Par exemple :

```
>>> decomp 1879
1
8
7
9
```

(*Indication : pour* `n` *un entier, que représente* `n/10` *?*)

Q5. `fibo: int -> int -> int -> int` telle que `fibo a b n` renvoie le n-ème terme de la suite de Fibonacci de premiers termes $u_0 = a$ et $u_1 = b$. Essayez d'écrire cette fonction avec **un seul** appel récursif !

Exercice 5: Listes

On rappelle la syntaxe des listes :

- La liste vide s'écrit `[]`, elle est de type `'a list`
- Si `E1` est une expression de type `'a` et `E2` une expression de type `'a list`, alors `E1 :: E2` est aussi de type `'a list`, et contient E1 comme premier élément, suivi des éléments de E2.

On peut utiliser `[]` et `::` dans les motifs. Par exemple :

```
(* Nombre d'éléments de l *)
let rec taille (l: 'a list): int =
  match l with
  | [] -> 0
  | x :: q -> 1 + taille q
```

Q1. Écrivez une fonction `somme: int list -> int` renvoyant la somme d'une liste.

Q2. Écrivez une fonction `recherche: 'a list -> 'a -> bool` qui, étant donné L une liste et x un élément, détermine si $x \in L$.

En OCaml, la fonction `failwith` permet d'afficher un message d'erreur et d'arrêter le programme. On l'utilise par exemple lorsqu'une fonction est appelée sur des arguments invalides :

```
(* Renvoie le maximum de l, une liste non vide *)
let rec maximum (l: 'a list) : 'a =
  match l with
  | [] -> failwith "Liste vide"
  | x :: [] -> x (* liste àun seul élément *)
  | x :: q -> let mq = maximum q in
              if mq > x then mq else x
```

Q3. Écrivez une fonction `n_eme: 'a list -> int -> 'a` telle que `n_eme i l` renvoie le i-ème élément de l, et lève une erreur (avec failwith) si l est trop courte.

Q4. Écrivez une fonction `filter: ('a -> bool)->'a list -> 'a list` telle que `filter f l` renvoie la liste de tous les éléments `x` de `l` vérifiant `f x = true`. Par exemple :

```
let est_pair (x: int) : bool =
  x mod 2 = 0

let l1 = filter est_pair [1;2;3;4;2;3;4] (* vaudra [2;4;2;4] *)
```

Q5. Pour deux listes $L_1 = [x_1; ...; x_n]$ et $L_2 = [y_1; ...; y_m]$, la concaténation de L_1 et L_2 est $L = [x_1; \ldots; x_n; y_1; \ldots; y_m]$. Écrivez une fonction `concatener` qui prend en entrée deux listes L_1, et L_2 et renvoie leur concaténation. (*Indication : on pourra raisonner par récurrence sur L_1*)

Q6. Écrivez une fonction `multi_concat: 'a list list -> 'a list` prenant en entrée une liste de listes et renvoyant leur concaténation.
Par exemple, `multiconcat [[1; 2]; [3]; [4; 5; 6]] = [1; 2; 3; 4; 5; 6]`.

Q7. Écrivez une fonction `map: ('a -> 'b)-> 'a list -> 'b list` prenant en entrée une liste L, une fonction f, et renvoyant la liste obtenue en appliquant f à chaque élément de L. Par exemple, `map (fun x -> x*x)[2; 3; 4] = [4; 9; 16]`.

Exercice 6: Fonctions classiques sur les listes

Q1. Copiez-collez le code des fonctions `taille`, `somme`, `recherche`, `filter`, `map`.

Q2. Écrivez une fonction `string_cat: string list -> string` qui renvoie la concaténation de toutes les chaînes de caractères de la liste donnée. On rappelle que l'opérateur `^` permet de concaténer deux strings.

On constate que toutes les fonctions des deux questions précédentes ont des définitions très proches, et on propose d'étudier une fonction qui permet de les généraliser. Cette fonction, classique en programmation fonctionnelle, s'appelle ***fold*** (ou parfois ***reduce***) :

```
let rec fold (f: 'a -> 'b -> 'b) (l: 'a list) (b: 'b) : 'b = match l with
| [] -> b
| x::q -> f x (fold f q b)
```

Q3. Recopiez la fonction fold, et évaluez les expressions suivantes :

```
fold (fun x y -> x^y) ["vive"; " "; "OCaml"; "!!!"] "";;
(* les opérateurs sont des fonctions, on peut donc également écrire: *)
fold (^) ["vive"; " "; "OCaml"; "!!!"] "";;

fold (+) [1;2;3;4] 0;;
fold (fun x l ->x::l) [1;2;3;4] [];;
```

`fold` sert donc à utiliser les éléments d'une liste pour accumuler un résultat à partir d'un élément de départ b et d'une fonction d'agrégation f. Plus précisément, `fold f [x1;x2;...;xn] b` renvoie $f(x_1, f(x_2, \dots f(x_n, b)\dots))$.

Par exemple, pour `fold (+) [1;2;3;4] 0`, le résultat est $1+(2+(3+(4+0)))=10$. On peut utiliser `fold` pour donner une nouvelle définition de la fonction `somme` :

```
let somme l =
  fold (+) l 0
```

Q4. En utilisant fold, donnez une nouvelle définition des fonctions `taille`, `recherche`, `filter`, `map`. A chaque fois, réfléchissez à :

- Quel est le résultat sur une liste vide : cela vous donne b
- Pour une liste `x::q`, si j'ai déjà le résultat r sur q, comment est-ce que je mets à jour ce résultat avec x : cela vous donne $f(x, r)$.

Les fonctions `filter, map, fold` forment un triplet d'outils très puissants en programmation fonctionnelle.

Q5. Sans utiliser `filter, map, fold`, écrire une fonction `range: int -> int list` telle que `range n = [0;1;2;...;n-1]`

Q6. En utilisant uniquement `filter, map, fold` et la fonction `range` de l'exercice précédent, écrire une fonction `somme_carres_div: int -> int` calculant la somme des carrés des diviseurs d'un entier non nul.

Exercice 7: Tris

Savoir comment trier une liste d'éléments est un problème classique d'informatique, pour lequel il existe plusieurs algorithmes à connaître.
Vous avez pu déjà étudier ces algorithmes en informatique tronc commun, mais lorsque l'on veut les implémenter en OCaml, on ne manipule plus des tableaux python, dans lesquels on peut lire et modifier n'importe quelle case. Il faut donc trouver des définitions récursives de nos tris.

On s'intéresse pour commencer au tri par sélection. Le principe, pour trier une liste L, est :

1. Si L est vide, on renvoie L
2. Sinon, on trouve l'élément m minimal dans L. On note L' la liste obtenue en ayant retiré de L une occurrence de m.
3. On trie récursivement L'
4. On renvoie $m :: L'$.

Par exemple, pour trier la liste $[4; 1; 3; 2]$:

- On trouve le minimum de la liste, 1, et on trie récursivement la liste des éléments restants $[4; 3; 2]$:
  - On trouve le minimum de la liste, 2, et on trie récursivement la liste des éléments restants $[4; 3]$:
    - ...
  - on obtient $[3; 4]$, on renvoie donc $2 :: [3; 4] = [2; 3; 4]$
- on obtient $[2; 3; 4]$, on renvoie donc $1 :: [2; 3; 4] = [1; 2; 3; 4]$

Q1. Écrire une fonction `find_min: 'a list -> 'a` qui renvoie le minimum d'une liste non vide.

Q2. Modifier la fonction pour qu'elle renvoie plutôt un couple (m, L') avec m le minimum de la liste d'entrée, et L' la liste privée d'une occurrence de m.

Q3. Écrire une fonction `select_sort: 'a list -> 'a list` implémentant le tri par sélection.

On s'intéresse maintenant au tri par insertion. Ce tri repose sur la fonction suivante :

```
1 (* Insère x dans l
2    Hypothèse: l est triée dans l'ordre croissant.
3    La liste obtenue reste triée dans l'ordre croissant.
4    Ex: insert 7 [1;4;8] = [1;4;7;8] *)
5 let rec insert (x: 'a) (l: 'a list) : 'a list =
```

Pour trier une liste non vide $x :: Q$, on commence par trier récursivement Q, puis on insère x dans la liste obtenue.

Q4. Implémenter la fonction `insert`, et en déduire une fonction de tri `insert_sort`.

Nous allons voir en cours que les deux fonctions de tri précédentes sont en $\mathcal{O}(n^2)$, ce qui signifie que pour trier une liste de taille n, elles prennent un temps proportionnel à n^2. Par exemple, pour trier une liste de 10^6 éléments, il faudra de l'ordre de 10^{12} opérations, ce qui prendra plusieurs heures sur un ordinateur standard. Il existe des

algorithmes de tri plus rapide, en $\mathcal{O}(n \log_2 n)$: pour trier une liste de 10^6 éléments, il faudra alors de l'ordre de $\sim 10^7$ opérations, ce qui prend quelques secondes sur un ordinateur standard !

Le **tri fusion** est un exemple d'un tel tri. Il fonctionne selon un principe appelé **diviser pour régner** (ou Divide and Conquer en anglais). Pour trier une liste L :

1. Si L est de taille 0 ou 1, on renvoie L ;
2. Sinon, on commence par séparer L en deux listes L_1, L_2 de tailles égales ;
3. On trie récursivement L_1 et L_2, notons L_1', L_2' les listes obtenues ;
4. On fusionne L_1 et L_2 en une liste triée.

La force du tri fusion vient du fait que l'on peut fusionner deux listes triées très efficacement.

Q5. Compléter le code suivant :

```
(* Renvoie une liste triée contenant les éléments de l1 et l2
   Hypothèse: l1 et l2 sont triées *)
let rec fusion (l1: 'a list) (l2: 'a list) : 'a list =
  match l1, l2 with
  | [], _ -> l2
  | _, [] -> ...
  | x1 :: q1, x2 :: q2 -> ...
```

Q6. Écrire une fonction `separer: 'a list -> ('a list * 'a list)` permettant de diviser une liste en deux listes de tailles égales. Si vous n'avez pas d'idée, imaginez que la liste en entrée est un deck de cartes, que vous distribuez à deux joueurs/-joueuses !

Q7. Écrire une fonction de tri fusion.