Lycée Chateaubriand
Informatique MP 2002/03
Mathieu Giraud

TP Caml n° 1 – Corrigé
Listes et polynômes

Ce corrigé ne prétend évidemment pas fournir la seule solution aux diverses questions. N'hésitez pas à me demander des précisions ou à me signaler des erreur ! Ce corrigé ainsi que les sources Caml sont disponibles à l'adresse `http://magiraud.free.fr/tpcaml`.

1 Opérations de base sur les listes

a.* On peut définir les listes à partir de "rien du tout" en utilisant des types (à comparer avec la définition des arbres). Les deux opérations de base sont en $\mathcal{O}(1)$.

```
type 'a liste = Vide | Suite of 'a * 'a liste ;;

let tete l = match l with
  | Vide -> failwith "tete: liste vide"
  | Suite (t, q) -> t ;;

let queue l = match l with
  | Vide -> failwith "queue: liste vide"
  | Suite (t, q) -> q ;;

let cons x l = Suite(x,l) ;;
```

b.

```
let rec longueur l = match l with
  | [] -> 0
  | _::q -> 1 + longueur q ;;

let rec est_element x l = match l with
  | [] -> false
  | t::q -> if (t == x) then true
                        else est_element x q ;;

let rec ajoute_a_la_fin l x = match l with
  | [] -> [x]
  | t::q -> t::(ajoute_a_la_fin q x) ;;

let rec miroir = function
  | [] -> []
  | t::q -> ajoute_a_la_fin (miroir q) t;;
```

La fonction `miroir` peut se réaliser à partir de `ajoute_a_la_fin`, ce qui mène à une complexité $\mathcal{O}\left(n^2\right)$. Voici une solution plus efficace ($\mathcal{O}(n)$) :

```
let rec aux l1 l2 = match l1 with
  | [] -> l2
  | t1::q1 -> aux q1 (t1::q2) ;;
```

```
let miroir l = aux l [] ;;
```

c. Les fonctions `reunion`, `intersection`, `difference` peuvent utiliser `est_element`.

```
let rec reunion l1 l2 = match l1 with
  | [] -> l2
  | t::q -> if (est_element t l2) then (reunion q l2)
                                  else t::(reunion q l2) ;;

let rec intersection l1 l2 = match l1 with
  | [] -> []
  | t::q -> if (est_element t l2) then t::(intersection q l2)
                                  else (intersection q l2) ;;

let rec difference l1 l2 = match l1 with
  | [] -> []
  | t::q -> if (est_element t l2) then (difference q l2)
                                  else q::(difference q l2) ;;
```

Si les listes `l1` et `l2` ont pour tailles respectives n_1 et n_2, chaque appel à `est_element` est en $\mathcal{O}(n_2)$. Il y a n_1 appels de ce type, d'où une complexité totale quadratique de $\mathcal{O}(n_1 n_2)$ (qu'on peut par exemple réécrire en $\mathcal{O}(n^2)$ si $n = \max\{n_1, n_2\}$).

d. Les fonctions `reunion_t`, `intersection_t`, `difference_t` peuven ne parcourir qu'une fois chaque liste. Il faut juste veiller à bien choisir "la liste dans laquelle on avance".

```
let rec reunion_t l1 l2 = match (l1,l2) with
  | [], _ -> l2
  | _, [] -> l1
  | t1::q1, t2::q2 -> if (t1 < t2) then t1::(reunion_t q1 l2)
                                  else t2::(reunion_t l1 q2) ;;

let rec intersection_t l1 l2 = match (l1,l2) with
  | [], _ -> []
  | _, [] -> []
  | t1::q1, t2::q2 -> if (t1 = t2) then t1::(intersection_t q1 q2)
                else if (t1 < t2) then     (intersection_t q1 l2)
                                  else     (intersection_t l1 q2) ;;

let rec difference_t l1 l2 = match (l1,l2) with
  | [], _ -> l2
  | _, [] -> l1
  | t1::q1, t2::q2 -> if (t1 = t2) then     (difference_t q1 q2)
                else if (t1 < t2) then t1::(difference_t q1 l2)
                                  else     (difference_t l1 q2) ;;
```

Ces fonctions ont une complexité $\mathcal{O}(n_1 + n_2) = \mathcal{O}(n)$. Remarquons enfin que l'intersection et la différence de la question **c** laissent les listes triées ! On aurait donc pu écrire (mais avec une complexité quadratique) :

```
let intersection_t = intersection ;;
let difference_t = difference ;;
```

e. Le tri fusion s'appuie sur la séparation d'une liste en deux moitiés (`separe`) et sur la fusion de deux listes triées (`reunion_t`).

```
let rec separe = function             (* Une possibilité *)
  | [] -> [], []
```

```
  | [a] -> [a], []
  | a::b::q -> let (s1, s2) = (separe q)
               in (a::s1, b::s2) ;;

let rec separe = function              (* Une autre possibilité *)
  | [] -> [], []
  | t::q -> let (s1, s2) = (separe q)
            in (t::s2, s1) ;;

let rec tri_fusion = function
  | [] -> []
  | [a] -> [a]
  | l -> let (s1, s2) = separe l
         in reunion_t (tri_fusion s1) (tri_fusion s2) ;;
```

Ce tri a une complexité de $\mathcal{O}(n \log n)$.

Aperçu de preuve dans le cas où n est une puissance de 2 : soit $C(n)$ le coût du tri fusion pour trier cette liste. Ce coût comprend

- la séparation de la liste en deux : n
- le tri récursif de chacune des deux demi-listes : $2C(n/2)$
- la fusion des deux listes triées : $n/2 + n/2 = n$

D'où $$C(n) = 2C(n/2) + 2n$$

En itérant, $$C(n) = 2(2C(n/4) + 2n/2)) + 2n = 4C(n/4) + 4n$$

En itérant k fois, $$C(n) = 2^k C(n/2^k) + 2kn$$

Lorsque $n = 2^k$ (c'est à dire $k = log_2 n$), on a $$C(n) = 2^k C(1) + 2kn$$

Or $C(1) = 1$ (cas d'arrêt dans `fusion` : une liste à un élément est déjà triée).

D'où $$C(n) = 2^k \cdot 1 + 2(\log_2 n)n = \mathcal{O}(n \log n)$$

f.*

```
let rec itere_operateur op x0 l = match l with
  | [] -> x0
  | t::q -> itere_operateur op (op x0 t) q ;;

let somme = itere_operateur (fun x y -> x + y) 0 ;;
let produit = itere_operateur (fun x y -> x * y) 1 ;;
let concatenation = itere_operateur (fun x y -> x ^ y) "" ;;
let longueur = itere_opetateur (fun _ y -> y + 1) 0 ;;
```

La fonction d'itération existe déjà en Caml (`iter`). On appelle parfois ce genre de fonctions prenant des fonctions en arguments fonctions "d'ordre supérieur". Il est bon de bien comprendre le type de ces fonctions :

```
itere_operateur : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

- `('a -> 'b -> 'a)` est le type de l'opérateur `op`
- `'a` celui de `x0`
- `'b list` celui de la liste

– et `'a` le résultat.

Pourquoi y a-t-il deux types génériques différents ? Si on prend la fonction `longueur`, elle s'applique à une liste quelconque (`'b list`), mais son `x0` comme son résultat sont des `int`.

g.* On peut trouver toutes les permutations d'une liste de manière récursive en trouvant les permutations de `t::q` à partir de celles de `q`. Je vous laisse y réfléchir. Comme il y a $n!$ permutations, le programme utilisé aura au moins une complexité exponentielle.

2 Polynômes d'entiers

h. `let degre p = (longueur p) - 1;;`

i. Cette question est difficile à traiter juste du premier coup. Le schéma de Hörner s'écrit :

$$\begin{aligned} P &= 5X^4 - 4X^3 + X^2 - 1 \\ &= 5X^4 - 4X^3 + 1X^2 + 0X - 1 \\ &= (((((0)X + 5)X - 4)X + 1)X + 0)X - 1 \end{aligned}$$

Autrement dit, de manière récursive, $P_0 = 0$, $P_{i+1} = P_i X + \alpha_i$, et $P_n = P$, où les α_i sont les coefficients en partant du poids le plus fort.

```
let rec horner p a = match p with
  | [] -> 0
  | t::q -> t + a * (horner q a) ;;
```

j. On pourra s'aider de deux fonctions auxiliaires :
– une fonction `lim : int list -> limite` calculant la limite du polynôme en $+\infty$
– une fonction `oppose : limite -> limite` calculant l'opposé d'une limite
Écrire enfin la fonction demandée en se servant éventuellement de `degre`. La complexité totale de `limite_poly` sera de $\mathcal{O}(n)$.

La suite...

– Les questions **k** à **o** n'ayant pas été abordées au cours du TP, je vous les laisse en exercice; n'hésitez pas à m'en parler si vous n'y arrivez pas.
– Si vous voulez faire du Caml à la maison, CamlLight se trouve à l'adresse

`http://caml.inria.fr/distrib-caml-light-fra.html`

– Les deux prochains TPs (après la Toussaint) $\longrightarrow$ arbres.