Lycée Chateaubriand
Informatique MP 2002/03
Mathieu Giraud

TP Caml n° 2 – Corrigé
Arbres

Ce corrigé ainsi que les sources Caml sont disponibles à l'adresse `http://magiraud.free.fr/tpcaml`.

1 Opérations de base sur les arbres binaires

a.

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree ;;

let rec hauteur = function
  | Leaf _ -> 0
  | Node(g,d) -> 1 + (max (hauteur g) (hauteur d)) ;;

let rec nb_noeuds = function
  | Leaf _ -> 0
  | Node(g,d) -> 1 + (nb_noeuds g) + (nb_noeuds d) ;;

let rec nb_feuilles = function
  | Leaf _ -> 1
  | Node(g,d) -> (nb_feuilles g) + (nb_feuilles d) ;;
```

On peut aussi se souvenir que, dans un arbre binaire, il y a exactement une feuille de plus qu'il y a de noeuds internes... et alors on écrit `let nb_feuilles a = (nb_noeuds a) + 1`.

b.

```
let rec miroir = function
  | Leaf n -> Leaf n
  | Node(g,d) -> Node(d,g) ;;
```

c.

```
let rec somme = function
  | Leaf n -> n
  | Node(g,d) -> (somme g) + (somme d) ;;
```

Toutes les fonctions que nous venons d'écrire parcourent entièrement l'arbre : elles ont une complexité de $\mathcal{O}(n)$.

Les fonctions `nb_noeuds`, `nb_feuilles` et `hauteur` ont la même forme et parcourent l'arbre en entier. Comme au dernier TP, il est possible de créer une fonction générique qui parcourt un arbre :

```
let rec parcours f1 f2 = function
  | Leaf n -> f1 n
  | Node(g,d) -> f2 (parcours f1 f2 g) (parcours f1 f2 d) ;;

let hauteur =     parcours (fun _ -> 0) (fun g d -> 1 + (max g d)) ;;
let nb_noeuds =   parcours (fun _ -> 0) (fun g d -> 1 + g + d) ;;
let nb_feuilles = parcours (fun _ -> 1) (prefix +) ;;
let somme =       parcours (fun x -> x) (prefix +) ;;
```

2 Arbres binaires de recherche

d. La fonction `recherche` ne parcourt qu'une branche de l'arbre. Sa complexité est donc de $\mathcal{O}(h)$, où h est la hauteur de la branche en question. Si n est le nombre de noeuds de l'arbre, on a toujours $0 \leq h \leq n$ (dans le cas d'un arbre peigne, $h = n$).

```
let rec recherche x = function
  | Vide -> false
  | Noeud (g,n,d) -> if (x == n) then true
                                 else if (x <= n) then recherche x g
                                                  else recherche x d ;;
```

e.

```
let rec insere x = function
  | Vide -> Noeud (Vide, x, Vide)
  | Noeud (g,n,d) -> if (x <= n) then Noeud ((insere x g), n, d)
                                 else Noeud (g, n, (insere x d)) ;;
```

La fonction `insere` est là encore en $\mathcal{O}(h)$.

f.* La réalisation de `retire` demande plusieurs étapes. Tout d'abord, on peut se rendre compte qu'enlever un noeud d'un ABR ne "chamboule" que le sous-arbre attaché à ce noeud, et pas ses ascendants ou ses cousins. On peut donc écrire `retire` à l'aide d'une fonction `retire_racine` qui enlève la racine d'un ABR.

```
let rec retire x a = match a with
  | Vide -> failwith "retire : élément non présent"
  | Noeud (g,n,d) ->  if (x == n) then retire_racine a
                                  else if (x <= n) then Noeud ((retire x g), n, d)
                                                   else Noeud (g, n, (retire x d)) ;;
```

Une solution pour enlever la racine d'un ABR est d'enlever l'élément maximal du sous-arbre gauche pour en faire la nouvelle racine. Ceci est effectué par la fonction auxiliaire `retire_plus_grand`.

```
let rec retire_plus_grand = function
  | Vide -> failwith "retire_plus_grand : arbre vide"
  | Noeud (g,n,d) -> if (d == Vide) then n, g
                                    else let dmax, d2 = retire_plus_grand d
                                         in dmax, Noeud (g,n,d2) ;;
```

```
let retire_racine = function
  | Vide -> failwith "retire_racine : arbre vide"
  | Noeud (g,_,d) -> if (g == Vide) then d
                                    else let gmax, g2 = retire_plus_grand g
                                         in Noeud(g2, gmax, d) ;;
```

La fonction `retire` va explorer une branche de l'arbre (jusqu'à trouver l'élément recherché) puis appeler `retire_racine` et donc `retire_plus_grand` qui va parcourir une branche de l'arbre : la complexité de l'ensemble est là encore de $\mathcal{O}(h)$.

g. Toutes les opérations que nous avons faites sur les ABR sont en $\mathcal{O}(h) = \mathcal{O}(n)$, ce qui peut poser problème pour un grand volume d'informations conservé dans l'arbre. Leur complexité en moyenne est beaucoup plus faible : si on crée un ABR en insérant successivement des nœuds "au hasard", sa hauteur va être en $\mathcal{O}(\log n)$ (Devroye, 1986).

Cependant, h peut valoir n (cas d'un arbre peigne). Pour éviter ces pires cas, il faut chercher à **équilibrer les arbres**. Par exemple, les AVL sont des ABR tels qu'en chaque noeud la différence de hauteur entre le sous-arbre gauche et le sous-arbre droit soit au plus de 1 en valeur absolue. Ces arbres nécessitent des opérations de base un peu plus complexes mais permettent des complexités en $\mathcal{O}(\log n)$ pour certaines opérations.

3 ABR et listes

h. Vous avez été nombreux à écrire `liste_of_arbre` sous la forme :

```
let rec liste_of_arbre = function
  | Vide -> []
  | Noeud (g,n,d) -> (liste_of_arbre g) @ [n] @ (liste_of_arbre d) ;;
```

C'est une solution. Nous avons cependant vu au TP 1 que l'opérateur de concaténation `@` est linéaire (on parcourt la première liste pour y "attacher au bout" la deuxième liste). Au total, cette première version de `liste_of_arbre` sera quadratique. On peut l'éviter en construisant la liste résultante au fur et à mesure :

```
let rec aux l = function
  | Vide -> l
  | Noeud (g,n,d) -> aux (n::(aux l d)) g ;;

let liste_of_arbre = aux [] ;;
```

i.

```
let rec arbre_of_liste = function
  | [] -> Vide
  | t::q -> insere t (arbre_of_liste q) ;;
```

On peut ainsi faire un tri en utilisant les deux fonctions. Ce tri reste quadratique.

```
let tri_abr l = liste_of_arbre (arbre_of_liste l) ;;
```

4 Arbres n-aires

Pour réaliser un arbre n-aire, il faut permettre à chaque noeud d'avoir une liste de sous-arbres.

```
type 'a n_arbre = nVide | nNoeud of 'a * ('a n_arbre list) ;;
```

Il est même possible de supprimer le type de feuille `nVide` en considérant que les feuilles sont les noeuds sans fils. Pour réaliser les "opérations de base", on peut par exemple utiliser des fonctions doublement récursives :

```
let rec somme_arbre = function
  | nVide -> 0
  | nNoeud (n,l) -> n + (somme_liste l)
and somme_liste = function
  | [] -> 0
  | t::q -> (somme_arbre t) + (somme_liste q) ;;
```

5 Retour sur le TP 1 : polynômes d'entiers

Il y avait une erreur dans l'exemple donné dans le TP1 : le polynôme $5X^4 - 4X^3 + X^2 - 1$ est représenté par la liste `[ -1; 0; 1; -4; 5]`. Ce codage 'coefficients de poids faible d'abord' permet d'effectuer facilement les opérations comme le schéma de Hörner (voir précédent corrigé) ou l'addition.

L'**addition** de deux polynômes pose problème pour l'élimination des zéros supplémentaires.[1] Nous utilisons pour cela une fonction `normalise` qui enlève les zéros en queue de liste.

```
let rec normalise = function
  | [] -> []
  | t :: q -> let q_norm = normalise q
              in if (q_norm = []) && (t = 0)
                 then []
                 else t :: q_norm ;;
```

Nous pouvons maintenant procéder à l'addition de deux polynômes par un parcours simultané des deux listes effectué dans une fonction `add_aux`. La fonction `add` finale appelle la normalisation sur le résultat de `add_aux`.

[1] Le corrigé qui suit provient des *Annales des concours E3A MP* aux éditions H&K.

```
let rec add_aux p1 p2 = match (p1, p2) with
  | [], _ -> p2
  | _, [] -> p1
  | t1 :: q1, t2 :: q2 -> t1 + t2 :: (add_aux q1 q2) ;;

let add p1 p2 = normalise (add_aux p1 p2) ;;
```

Occupons-nous maintenant de la **multiplication**. Le premier réflexe peut être d'expliciter tous les $\mathcal{O}(n^2)$ produits partiels. En effet, si $P = \sum a_i x^i$ et $Q = \sum b_i x^i$, alors les coefficients de $R = PQ$ s'écrivent

$$r_i = \sum_{k=0}^{i} a_k b_{i-k} \qquad (0 \leq i \leq m+n)$$

Le calcul séparé de chaque r_i par accumulation des produits partiels produirait idéalement un algorithme en $\mathcal{O}(n^2)$, mais il est difficile à mettre en place lorsqu'on travaille sur des listes. Nous allons ici utiliser l'addition des polynômes pour obtenir une méthode plus directe.

Soient donc deux polynômes à multiplier $P = \sum_{i=0}^{n} a_i X^i$ et Q.

On cherche $$R = PQ = \sum_{i=0}^{n} a_i Q X^i$$

Nous allons calculer R de manière récursive en mettant à profit l'addition des polynômes que nous avons traitée ci-dessus. Ainsi, en posant

$$R_k = \sum_{i=0}^{k} a_i Q X^i$$

nous avons $$\begin{cases} R_0 = 0 \\ R_k = R_{k-1} + a_k Q X^k \quad \forall k \in [\![1 ; n]\!] \\ R = R_n \end{cases}$$

Nous écrivons donc deux fonctions : `mult_cons` qui réalise la multiplication d'un polynôme P par une constante a, et `mult_xn` qui multiplie P par le monôme X^k. Ces deux fonctions sont linéaires.

```
let rec mult_cons a = function
  | [] -> []
  | t :: q -> (a * t) :: (mult_cons a q) ;;

let rec mult_xn i p =
  if (i=0)
  then p
  else mult_xn (i-1) (0 :: p) ;;
```

Nous pouvons écrire maintenant la multiplication en nous servant des fonctions précédemment définies. L'algorithme final est quadratique.

```
let rec mult_aux p1 p2 p3 i =
```

```
  match p1 with
  | [] -> []
  | t1 :: q1 -> let t1p2i = mult_xn i (mult_cons t1 p2)
                in add t1p2i (mult_aux q1 p2 p3 (i + 1)) ;;

let mult p1 p2 = mult_aux p1 p2 [] 0 ;;
```

Il existe d'autres méthodes plus efficaces comme celle de Karatsuba qui permet une complexité en temps de $\mathcal{O}\left(n^{\log_2 3}\right) \simeq \mathcal{O}\left(n^{1,58}\right)$. Pour cela, on remarque que le calcul $aX^2 + bX + c = (p_1X + p_0)(q_1X + q_0)$ peut être effectué en seulement trois multiplications au lieu de quatre en utilisant le fait que $b = (p_1 + q_1) * (p_0 + q_0) - a - c$. L'algorithme de Karatsuba applique récursivement ce schéma (stratégie *diviser pour régner*) en remplaçant le calcul de PQ par celui de

$$(P_1X^{\lfloor \frac{n}{2} \rfloor} + P_2)(Q_1X^{\lfloor \frac{n}{2} \rfloor} + Q_2)$$

La suite...

- prochain TP $\longrightarrow$ toujours sur les arbres.