Science Sorbonne Université

UE Ouverture

Rapport de projet

Huynh Tan Khiem
Kébé Yatté

Enseignants : Antoine GENITRINI
Emmanuel CHAILLOUX

Présentation

Dans ce projet, nous avons pour objectif d'élaborer des stratégies de manipulation de deux structures de données : l'une sous forme linéaire et l'autre arborescente.

1.1 - Polynôme sous forme linéaire

Dans cette partie nous considérons la représentation suivante des polynômes en la variable formelle x. Le monôme c·x^n est donné par le couple (c, d) $\in Z \times N$. Un polynôme est une liste de monômes.

Question 1.1

Définition d'une structure de données permettant de manipuler les polynômes.

```
type monome = {coeff : int; degre : int};;
type polynome = monome list;;
```

Dans la suite nous dirons qu'un polynôme est représenté sous forme canonique si la liste le représentant est triée par ordre de croissant des degrés des monômes, s'il n'existe pas plusieurs monônes de même degré et si (c; d) est un élément de la liste alors c $\neq 0$. Enfin le polynôme valant 0 est représenté par une liste vide.

Question 1.2

Définition d'une fonction **canonique** qui prend un entrée un polynôme sous forme linéaire et qui renvoie sa représentation canonique.

```
let addition_monomes (ma : monome) (mb : monome) : monome =
        if ma.degre <> mb.degre then failwith "Addition Impossible"
        else {coeff=ma.coeff+mb.coeff; degre=ma.degre};;
```

```
let rec inserer_monome (m:monome) (p:polynome) = match p with
 | [] -> m::[]
 | t::q ->
         if m.degre > t.degre then m::p
         else if m.degre = t.degre then (addition_monomes m t)::q
         else t::(inserer_monome m q);;
```

```
let rec canonique (p:polynome) = match p with
 | [] -> []
 | t::q -> inserer_monome t (canonique q);;
```

- La complexité est mesuré en fonction de nombre de comparaison de degré dans la fonction **inserer_monome**.
La complexité en pire cas de la fonction **inserer_monome** est n avec n la taille de polynome en entrée.
Donc la complexité de **canonique** est:
T(n) = n - 1 + T(n-1)
$\Rightarrow T(n) = O(n^2)$

Question 1.3

Définition d'une fonction **poly_add** prenant en entrée deux polynômes canoniques et qui renvoie leur somme sous forme canonique.

```
let rec poly_add (p1 : polynome) (p2 : polynome) : polynome = match p1, p2 with
        |p1,[]-> p1
        |[],p2 -> p2
        |({coeff = _; degre=d1} as h1)::t1, ({coeff = _; degre=d2} as h2)::t2 ->
                if d1 < d2
                        then h1::poly_add t1 p2
                else if d2 < d1
                        then h2::poly_add p1 t2
                else
                        let c = h1.coeff+h2.coeff in
                                if c = 0
                                        then poly_add t1 t2
                                else
                                        {coeff=c; degre=d1}::poly_add t1 t2;;
```

- La complexité en pire cas en fonction de nombre d'addition de monome: $O(max(n_1, n_2))$ avec n_1, n_2 la taille de deux polynomes en entrée.

Question 1.4

Définition d'une fonction **poly_prod** prenant en entrée deux polynômes canoniques et qui renvoie leur produit sous forme canonique.

```
let produit_monomes (ma : monome) (mb : monome) : monome =
        {coeff=ma.coeff*mb.coeff ; degre=ma.degre+mb.degre};;
```

```
let produit_monomes (ma : monome) (mb : monome) : monome =
        {coeff=ma.coeff*mb.coeff ; degre=ma.degre+mb.degre};;
```

```
let produit_monome_polynome (m : monome) (p : polynome) =
    List.map (fun mp -> produit_monomes m mp) p;;

let poly_prod (p1 : polynome) (p2 : polynome) =
  List.fold_left
    (fun acc m1 -> poly_add acc (produit_monome_polynome m1 p2))
    [] p1;;
```

- La complexité en pire cas c'est $O(n^3)$ en fonction d'opération entre des monomes (addition, multiplication) avec n la taille supérieure des polynomes en entrée.

1.2 - Expression arborescente

Dans cette partie nous allons définir des expressions arborescentes qui représentent des opérations simples sur des polynômes en la variable x. Voilà la grammaire qu'elles vérifient :

L'arbre représenté à gauche dans la Figure 1 vérifie la grammaire précédente. Par contre, celui représenté à droite ne la vérifie pas (deux opérateurs + se succèdent, et une variable x n'est pas associée à une puissance).

$$
\begin{aligned}
E &= int \mid E_{\wedge} \mid E_{+} \mid E_{*} \\
E_{\wedge} &= x \ \hat{} \ int^{+} \\
E_{+} &= (E \setminus E_{+}) + (E \setminus E_{+}) + \ldots \\
E_{*} &= (E \setminus E_{*}) * (E \setminus E_{*}) * \ldots
\end{aligned}
$$

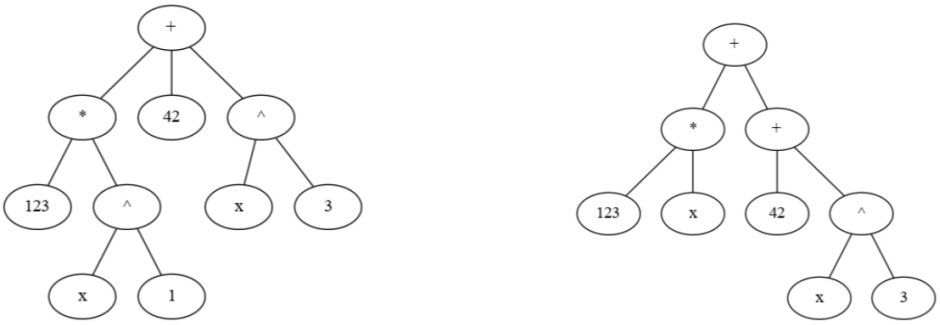


Figure 1: Arbres représentant $123 \cdot x + 42 + x^3$

Question 1.5

Définition d'une structure de données permettant de manipuler les arbres présentés sur la figure 1.

```
exception GrammaireNotVerified;;

type operator = Mul | Plus | Pow;;

(* Arbre qui vérifie la grammaire *)
type grammaire_arbre =
  | Empty
  | Pow_Node of {op : operator; var : string; degre : int}
  | Mul_Or_Plus_Node of {op : operator; sous_arbres : grammaire_arbre list}
  | Int_Node of {label : int};;
```

Question 1.6

Définition de l'expression de l'arbre à gauche de la figure 1 :

```
    let a = Grammaire_arbre.Mul_Or_Plus_Node{
    op = Mul;
    sous_arbres = [
        Mul_Or_Plus_Node{
            op = Mul;
            sous_arbres = [
                Int_Node{label = 123};
                Pow_Node{
                    op = Pow;
                    var = "x";
                    degre = 1
                }
            ]
        };
```

```
        Int_Node{label = 42};
        Pow_Node{
            op = Pow;
            var = "x";
            degre = 3
        }
    ]
}
```

Question 1.7

Définir une fonction **arb2poly** prenant en entrée une expression arborescente et la transformant en un polynôme canonique

```
let rec arb2poly (g_a : grammaire_arbre) : Polynomes.polynome = match g_a with
    | Empty -> []
    | Int_Node{label = x} -> [{coeff = x; degre = 0}]
    | Pow_Node{op = Pow; var = _; degre = d} -> [{coeff = 1; degre = d}]
    | Mul_Or_Plus_Node{op = Plus; sous_arbres = lst_arbre}
        -> List.fold_left (fun acc sous_g_a
                -> Polynomes.poly_add acc (arb2poly sous_g_a)) [] lst_arbre
    | Mul_Or_Plus_Node{op = Mul; sous_arbres = lst_arbre}
        -> List.fold_left (fun acc sous_g_a
                -> Polynomes.poly_prod acc (arb2poly sous_g_a)) [{coeff = 1; degre = 0}] lst_arbre
    | _ -> raise GrammaireNotVerified
```

1.3 - Synthèse d'expressions arborescentes

Question 1.8

Implémentation de la fonction **extraction_alea** prenant en entrée deux listes d'entiers, notée L et P. Cette fonction aura pour variables locales *len* (variable ayant pour valeur la taille de la liste d'entier L) et r qui contiendra une valeur comprise entre 1 et *len*, l'affectation de r se fera de manière aléatoire. La fonction retournera un couple de listes dont la première est la liste L dans laquelle on a retiré le r-ième élément et la deuxième est la liste P dans laquelle on a ajouté, en tête, le r-ième élément extrait de L, (celui qui vient d'être retiré de L).

```
let extraction_alea l p =
  let len = (List.length l) in
  let r = (Random.int len) in
  let rec aux c l1 l2 =
    match c, l1 with
    | _, [] -> (l, p)
    | 0, h::t -> (List.rev_append l2 t, h::p)
    | _, h::t -> aux (c - 1) t (h::l2)
  in aux r l [];;
```

Question 1.9

Implémentation de la fonction **gen_permutation** prenant en entrée une valeur entière n. Cette fonction génère une liste L des entiers de 1 à n (L est triée de façon croissante) et une liste vide P, puis vide entièrement la liste L et remplit la liste P en appelant **extraction_alea**.

```
let rec shuffle (l, p) =
  match l with
```

```
    | [] -> p
    | _ -> shuffle (extraction_alea l p);;
```

```
  let gen_permutation n =
    let rec aux count list =
      match count with
      | 0 -> shuffle  (list, [])
      | _ -> aux (count - 1) (count::list)
    in aux n [];;
end;;
```

Question 1.10

Implémentation de la fonction **ABR** qui étant donnée une liste d'entiers tous distincts en entrée, construit l'ABR associé à cette liste.

```
(*Creation du type ABR*)
  type 'a binary_tree =
    | Empty
    | Node of {label :'a; left : 'a binary_tree; right : 'a binary_tree};;

(*Fonction d'insertion*)
  let rec insert (tree : int binary_tree) (n : int) : int binary_tree =
    match tree with
    | Empty -> Node{label = n; left = Empty; right = Empty}
    | Node{label = x; left = l; right = r} ->
      if n = x then
        tree
      else if n < x then
        Node{label = x; left = insert l n; right = r}
      else
        Node{label = x; left = l; right = insert r n}

(*Fonction ABR*)
  let construct (list : int list) : int binary_tree =
    let rec aux l bt =
      match l with
      | [] -> bt
      | h::t -> aux t (insert bt h)
    in aux list Empty
```

```
  let rec etiquetage ?operator_list:(op_lst=[("+", 0.75); ("*", 0.25)])
    (tree : int binary_tree) : string binary_tree =
    match tree with
    | Empty -> Empty
    | Node{label = l; left = Empty; right = Empty} ->
      if l mod 2 = 1 then
        let e = Random.int 400 - 200 in
          Node{
                label = "*";
                left = Node{label = string_of_int e; left = Empty; right = Empty};
```

Question 1.11 Définir une fonction `etiquetage` prenant en entrée un ABR, le parcourant et reconstruisant un arbre ayant la même structure et tel que les sous-arbres de l'entrée soit reconstruit de la façon suivante :

- si un nœud interne de valeur ℓ a deux enfants qui sont des feuilles $\bullet$, alors
  - si ℓ est impair, construire un enfant gauche étiqueté par un *int*, généré uniformément dans l'intervalle $\{-200, -199, \cdots -1, 0, 1, \ldots, 200\}$, un enfant droit étiqueté par x et le parent des deux enfants est étiqueté par $*$;
  - si ℓ est pair, construire un enfant gauche étiqueté par x, un enfant droit étiqueté par un *int* généré uniformément dans l'intervalle $\{0, 1, \ldots, 100\}$ et l'étiquette du nœud parent a valeur $\wedge$.
- si un nœud interne de valeur ℓ a deux enfants dont au moins l'un des deux n'est pas une feuille, alors construire un nœd interne étiqueté $+$ avec probabilité $3/4$ ou étiqueté $*$ avec probabilité $1/4$.
- pour chaque feuilles restante, construire un nœud étiqueté avec un entier (avec probabilité $1/2$) ou avec la variable x (probabilité $1/2$).

L'arbre à droite de la Figure 1 est une expression que les dernières fonctions pourraient produire, par exemple à partir de la liste d'entiers $[2, 1, 3, 4]$.

```
                right = Node{label = "x"; left = Empty; right = Empty}
            }
      else
        let e = Random.int 100 in
          Node{
                label = "^";
                left = Node{label = "x"; left = Empty; right = Empty};
                right = Node{label = string_of_int e; left = Empty; right = Empty}
           }
    | Node{label = _; left = left_tree; right = Empty} ->
      let op = Random_choice.random_choice op_lst (Random.float 1.) in
        let e = Random.int 400 - 200 in
          let lst = [(string_of_int e, 0.5); ("x", 0.5)] in
            Node{
                label = op;
                left = etiquetage left_tree; r
                right = Node{
                            label = Random_choice.random_choice lst (Random.float 1.);
                            left = Empty;
                            right = Empty}
            }
    | Node{label = _; left = Empty; right = right_tree} ->
      let op = Random_choice.random_choice op_lst (Random.float 1.) in
        let e = Random.int 400 - 200 in
          let lst = [(string_of_int e, 0.5); ("x", 0.5)] in
            Node{
                label = op;
                left = Node{
                            label = Random_choice.random_choice lst (Random.float 1.);
                            left = Empty;
                            right = Empty};
                right = etiquetage right_tree
            }
    | Node{label = _; left = left_tree; right = right_tree} ->
      let op = Random_choice.random_choice op_lst (Random.float 1.) in
        Node{
            label = op;
            left = etiquetage left_tree;
            right = etiquetage right_tree
        }
  ;;
```

Pour assurer le bon déroulement de l'algorithme, la fonction **etiquetage** fait appel à la fonction

random_choice qui prend en entrée une liste de tuples ('a * float) représentant un couple liant un élément et et une probabilité et renvoie un élément de la liste selon les weights (probability)

```
  exception EmptyList;;
  let rec random_choice (l : ('a * float) list) (k : float) : 'a = match l with
    | [] -> raise EmptyList
    | (v, p) :: t -> if k < p then v else random_choice t (k -.p)
;;
```

2. Expérimentations

- Dans la suite nous allons étudier trois stratégies pour effectuer l'addition ou la multiplication sur une liste de polynomes.
- La fonction **generate_polynome** pour générer un polynome en étant donné la taille

```
let generate_polynome (taille : int) : Polynomes.polynome =
  let lst = Utils.gen_permutation taille in
    let a = Abr.construct lst in
      let a_labeled = Abr.etiquetage a in
        let a_grammaire = Abr.gen_arb a_labeled in
          let p = Grammaire_arbre.arb2poly a_grammaire in p;;
```

- La fonction **construct_list_polynomes** pour construire une liste de polynomes en étant donné la taille

```
let construct_list_polynomes (taille_lst : int) (taille_polynome : int) : Polynomes.polynome list =
  let rec aux count lst = match count with
  | 0 -> lst
  | _ -> aux (count - 1) (generate_polynome taille_polynome :: lst)
  in aux taille_lst [];;
```

- Stratégie 1: Utiliser la fonction **List.fold_left** du module liste.
- Stratégie 2: Utiliser récursive pattern matching.

```
let rec add_list_polynome (lst_pols : Polynomes.polynome list) : Polynomes.polynome =
  match lst_pols with
  | [] -> []
  | p1 :: ps -> Polynomes.poly_add p1 (add_list_polynome ps);;
```

- Stratégie 3: Même idée avec la stratégie 2 mais cette fois on utiliser un accumulator pour avoir une "tail recursion"

```
let add_list_polynome_tail_recursive (lst_pols : Polynomes.polynome list) : Polynomes.polynome =
  let rec add_acc acc lst = match lst with
  | [] -> acc
  | p1 :: ps -> add_acc (Polynomes.poly_add acc p1) ps
  in add_acc [] lst_pols;;
```

- Stratégie 2 et 3 pour la multiplication est un peu différent au niveau de la syntaxe mais l'idée reste la même:

```
let rec mul_list_polynome (lst_pols : Polynomes.polynome list) : Polynomes.polynome =
  match lst_pols with
  | [] -> [{coeff = 1; degre = 0}]
```

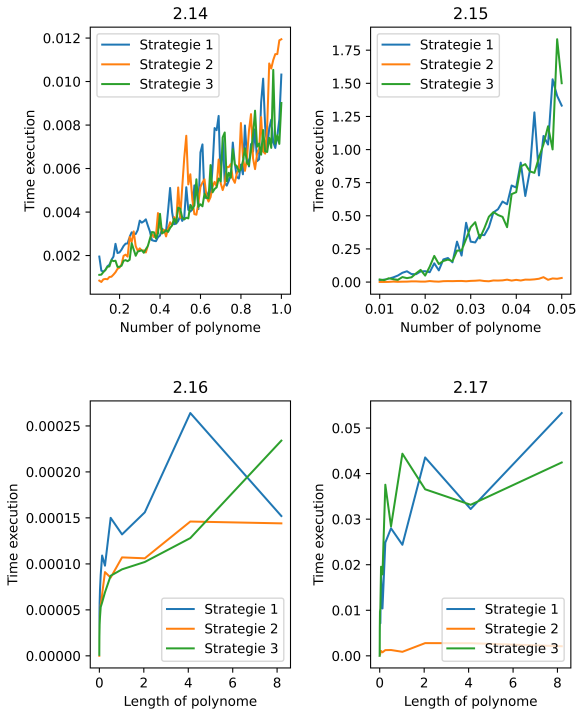
```
  | p1 :: ps -> Polynomes.poly_prod p1 (mul_list_polynome ps);;

let mul_list_polynome_tail_recursive (lst_pols : Polynomes.polynome list) : Polynomes.polynome =
  let rec mul_acc acc lst = match lst with
  | [] -> acc
  | p1 :: ps -> mul_acc (Polynomes.poly_prod acc p1) ps
  in mul_acc [{coeff = 1; degre = 0}] lst_pols
```

- Pour les deux question 2.14 et 2.15, pour chaque n on a t le temps d'exécution de l'addition ou la multiplication sur n polynomes de taille 20. On écrire ces tuple (n, t) dans des fichier texte et dessiner les courbes avec python en récupérant ces données et les traiter avec quelques étapes de "pre-processing"
- Exemple l'implémentaion de la stratégie 1:

```
let n = ref 100 and lst = ref [] and lst_time = ref [] in
  while !n <= 1000 do
    lst := !n :: !lst;
    let start = Unix.gettimeofday ()
    in let l = construct_list_polynomes !n 20
    in let _ = List.fold_left (fun acc pol -> Polynomes.poly_add acc pol) [] l
    in let stop = Unix.gettimeofday ()
    in let t = stop -. start
    in lst_time := t :: !lst_time;
    n := !n + 10
  done;
  let data = List.combine !lst !lst_time in
  let oc = open_out "data/data1.txt" in
  print_numbers oc data;;
```

- Les résultats on a obtenu:



- Avec l'addition, les 3 stratégies donnent les résultat assez similaires. Mais avec la multiplication, la stratégie 2 donne un résultat beaucoup meilleurs que les autres deux stratégies ($\approx$ 10 fois plus vite), ce qui est suprenant.