DEVOIR DE PROGRAMMATION
UE OUVERTURE

M1 - STL

2021/2022

Joumana ELDAKAR

Sommaire

Introduction

Un projet de master Informatique – STL pour le UE d'Ouverture. Il est consisté de programmer avec le langage de programmation « OCaml » deux modèles de structure de données : forme linéaire et arborescente. Aussi d'analyser la complexité du code.

Partie 1

Polynôme sous forme linéaire

La représentation des polynômes en la variable formelle x

Le monôme c . x^d est un couple (c, d) ∈ Z x N.
c = coefficient de x **d** = puissance de x

Un polynôme est une liste de monôme

1. Structure de données pour manipuler des polynômes `(coef, puissance de x)` :

```
type monome = int * int;;

type polynome = monome list;;
```

2. Une fonction canonique : qui représente une forme canonique de polynôme.
   La forme canonique a trois règles :
   - Pas de coefficient de 0 pour x
   - Pas de doublon de degré de x
   - Trier par ordre décroissant les degrés

```
(** tri par insertion O(N2) complexité *)

let rec insere (elem : monome) (liste : polynome) : polynome  =
    match liste with
    |  [] -> elem::[]
    |  tete::queue ->
        if snd elem > snd tete then elem :: liste
        else if snd elem = snd tete then (fst tete + fst elem, snd elem):: queue
        else tete :: insere elem queue
;;

let rec canonique (l : polynome) : polynome =
    match l with
    | [] -> []
    | h::t -> if (fst h == 0) then canonique t else
        insere h (canonique t)
```

On utilise une fonction auxiliaire **insere** pour pouvoir insérer un élément dans la liste de polynôme de l'ordre décroissant de puissance de x

Complexité

La fonction **insere** est un tri par insertion avec la complexité de $O(n^2)$.
La fonction **canonique** fait un appel de la fonction **insere** n fois alors $O(n^2)$.

3. La fonction poly_add : l'addition de deux polynômes canoniques.

```
(** val poly_add : polynome -> polynome -> polynome *)

let rec poly_add (l1 : polynome) (l2 : polynome) : polynome  =
    match l1, l2 with
    | [], _ -> l2
    | _,[] -> l1
    | h1::q1, h2::q2 ->
        if snd h1 > snd h2 then (h1 :: poly_add q1 l2) else
        if snd h1 = snd h2
            then (fst h1 + fst h2, snd h1):: poly_add q1 q2
        else (h2 :: poly_add l1 q2)
;;
```

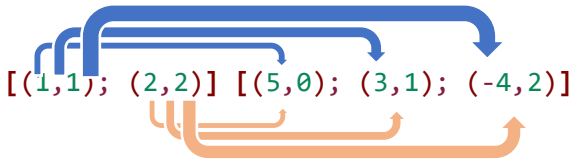
Complexité

Poly_add on parcours les deux listes en même temps alors $O(n^2)$.

4. La fonction poly_prod : la multiplication de deux polynômes canoniques.

```
(** val poly_prod : polynome -> polynome -> polynome *)
let rec poly_prod (l1 : polynome) (l2 : polynome) : polynome  =
    let rec produit (e : monome) (lp : polynome) : polynome =
        match lp with
        | [] -> []
        | h1::q1 -> (fst h1 * fst e, snd h1 + snd e)::produit e q1
    in match l1, l2 with
        | l1, [] -> l1
        | h1::[], l2 -> produit h1 l2
        | [], l2 -> l2
        | h1::q1, h2::q2 -> poly_add (produit h1 l2) (poly_prod q1 l2)
```

La fonction **poly_prod** a une fonction **produit** qui fait le produit de chaque élément de la liste 1 avec chaque élément de liste 2.



Complexité

La complexité au pire cas est $O(n^2)$:

- **Produit** a comme complexité O(n)
- Et **poly_add** : $O(n^2)$

Partie 2

Expression Arborescent

Grammaire :

$\mathbf{E} = \text{int} \mid E_{\wedge} \mid E_{+} \mid E_{*}$

$\mathbf{E_{\wedge}} = x \wedge \text{int}^{+}$

$\mathbf{E_{+}} = (E \setminus E_{+}) + (E \setminus E_{+}) + \ldots$

$\mathbf{E_{*}} = (E \setminus E_{*}) * (E \setminus E_{*}) * \ldots$

E contient est un entier, $\mathbf{E_{\wedge}}$, $\mathbf{E_{+}}$ ou $\mathbf{E_{*}}$

$\mathbf{E_{\wedge}}$: **x** avec un nombre entier positif pour sa puissance

$\mathbf{E_{+}}$ **:** la somme des **E** sauf $\mathbf{E_{*}}$ (ne contient pas un $\mathbf{E_{*}}$)

$\mathbf{E_{*}}$ **:** la multiplication des **E** sauf $\mathbf{E_{+}}$ (ne contient pas $\mathbf{E_{*}}$)

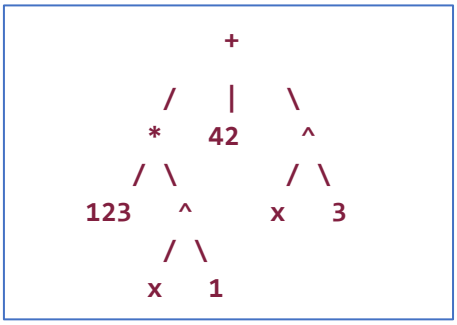
5. Structure de données d'un arbre de cette grammaire

```
type arbre =
    | NodeInt of int
    | NodePower of int
    | NodePlus of arbreP list
    | NodeMulti of arbreM list

and arbreP =
    | NodeIntP of int (* int est positif ou negatif *)
    | NodePowerP of int (**x^int+, int est positif *)
    | NodeMultiP of arbreM list
        (* jamais deux * successif parce que le noeud d'addition
      contient l'arbreM où ne contient pas le noeud addition *)

and arbreM =
    | NodeIntM of int (* int est positif ou negatif *)
    | NodePowerM of int (**x^int+, int est positif *)
    | NodePlusM of arbreP list
        (* jamais deux + successif parce que le noeud de multiplication
      contient l'arbreP où ne contient pas le noeud multiplication *)
;;
```

Création d'une structure comme suivant : **arbre** où équivaut à **E** qui contient toutes les autres règles de grammaire, deux autres types diffèrent :

- **Type arbreP** : est E_+ qui contient toutes les règles sauf la règle de E_*
- **Type arbreM** : est E_* qui contient toutes les règles sauf la règle de E_+

Le type arbre contient **NodeMulti** et **NodePlus** qui se sont des listes de **arbreM** et **arbreP** respectivement parce que ces nœuds peuvent contenir plusieurs fils.



6. Expression qui représente le polynôme (123 * x + 42 + x^3) en type arbre

```
let contruireArbre = NodePlus([
                NodeMultiP([ NodeIntM 123; NodePowerM 1 ]);
                NodeIntP 42;
                NodePowerP 3
                ])

    ;;
```

7. La fonction arb2poly : transformer une expression arborescente en polynôme canonique

- Deux fonctions auxiliaires : pour transformer les deux autres types de l'arbre du nœud plus et produit en polynôme canonique
  - **Fonction récursif arbM2poly :** en utilisant la fonction **poly_prod** pour multiplier chaque polynôme produit de chaque nœud.
    (Transformer **type arbreM** en **type polynome)**
  - **Fonction récursif arbP2poly :** en utilisant la fonction **poly_add** pour addition chaque polynôme produit de chaque nœud.
    (Transformer **type arbreP** en **type polynome)**
- La fonction **arb2poly** :
  - Le cas où on a un nœud **NodeMulti** -> un appel vers la fonction **arbM2poly**
  - Le cas où on a un nœud **NodePlus** -> un appel vers la fonction **arbP2poly**
  - Et pour les autres cas on produit un **monome**
    (pour le cas d'un entier ou une puissance)

```
let rec arbM2poly (a: arbreM list) : polynome =

    match a with
    | [] -> []
    | NodeIntM x :: t-> poly_prod [(x,0)] (arbM2poly t)
    | NodePowerM x :: t-> poly_prod [(1, x)] (arbM2poly t)
    | NodePlusM l :: t -> (arbP2poly l) @ (arbM2poly t)
and arbP2poly  (a: arbreP list) : polynome =
     match a with
    | [] -> []
    | NodeIntP x :: t -> poly_add [(x,0)] (arbP2poly t)
    | NodePowerP x :: t -> poly_add [(1, x)] (arbP2poly t)
    | NodeMultiP l :: t -> (arbM2poly l) @ (arbP2poly t)
;;

let arb2poly (a: arbre) : polynome =
    match a with
    | NodeInt x -> [(x,0)]
    | NodePower x -> [(1, x)]
    | NodeMulti [] -> []
    | NodePlus [] -> []
    | NodeMulti l -> canonique (arbM2poly l)
    | NodePlus l -> canonique (arbP2poly l)
;;
```

Partie 3

Synthèse d'expressions arborescentes

8. La fonction extraction_alea :

Prend deux listes d'entiers et choisir arbitrairement un élément et l'ajoute entête de la deuxième liste.

- Une fonction auxiliaire : **remove** pour supprimer un élément d'une liste avec une complexité de **O(n)**
- **Utilisation de Random :**
  - **Random.self_init** pour générer chaque fois un nombre différent de celui générer avant,
  - On utilise **Random.int** pour générer un nombre entier aléatoire
  - Appel de la fonction **remove** pour supprimer l'entier dans la première liste et qui renvoie un couple une liste entier sans l'entier à supprimer et l'entier qui a été supprimer de la liste pour pouvoir l'ajouter dans la deuxième liste.

```
let rec remove (x: int) (i: int) (l: int list) : int list  * int  =

  match l with
  | [] -> failwith "liste vide"
  | h :: t -> if x=i then (t,h)
                  else h::(fst (remove x (i+1) t)), snd (remove x (i+1) t)

let extraction_alea (m: int list) (p: int list) : int list * int list =
    let l = List.length m in
      let _ = Random.self_init () and r = Random.int l in
          match m with
          | [] -> ([],p)
          | h::t -> let c = (remove r 0 m) in
                      fst c, snd c :: p
;;
```

9. La fonction gen_permutation :

L'algorithme de shuffle de Fisher-Yates.

a. Générer une liste **L** de 1 à n (avec **create_liste**) en utilisant une fonction auxiliaire **insere_liste** similaire à celui de **insere** de la **partie 1**
b. Générer une liste vide **P**
c. Vider **L** et remplir **P** avec la méthode de ***extraction_alea***

```
let rec insere_liste (elem : int) (liste : int list) : int list  =

    match liste with
    |  [] -> elem::[]
    |  tete::queue ->
        if  elem <= tete then elem :: liste
        else tete :: insere_liste elem queue ;;

let rec create_liste n =
    if n!=0 then (insere_liste n (create_liste (n-1))) else [] ;;

let gen_permutation n =
    let rec permutation (m: int list) (p: int list) : int list =
        match m with
        | [] -> p
        | h::t -> let (a, b) = (extraction_alea m p) in permutation a b
    in
    permutation (create_liste n) [] ;;
```

Arbre Binaire de Recherche

10. La structure de ABR :

```
type abr =
  | Feuille
  | Noeud of int *  abr *  abr
;;
```

```
let rec insert l a =
  match a with
    | Feuille -> Noeud(l,Feuille,Feuille)
    | Noeud(k, fg, fd) -> if l < k  then
                          Noeud(k, (insert l fg), fd )
                        else Noeud(k,fg, (insert l fd))
;;

let rec construireARB l a =
  match l with
    | [] -> a
    | h::t -> construireARB t (insert h a)
```

11. La fonction etiquetage :

Après la structure de la fonction **etiquetage** défini dans le sujet, on peut la défini comme suit

```
let rec etiquetage a =
  match a with
  | Feuille -> if Random.bool() then Noeud(int_of_char 'x', Feuille, Feuille) else Noeud(Random.int (201+201)-201, Feuille, Feuille)
  | Noeud(l, fg, fd) -> match fd, fg with
                        | Feuille, Feuille -> if (l mod 2 == 1) then let _ = Random.self_init () and r = Random.int (201+201)-201 in
                                                                     Noeud(int_of_char '*', Noeud(r, Feuille, Feuille), Noeud(int_of_char 'x', Feuille, Feuille))
                                              else let _ = Random.self_init () and r = Random.int 100 in
                                                                     Noeud(int_of_char '^', Noeud(int_of_char 'x', Feuille, Feuille), Noeud(r, Feuille, Feuille))
                        | _, _ -> if (Random.bool() && Random.bool()) then Noeud(int_of_char '*', etiquetage fg, etiquetage fd)
                                  else Noeud(int_of_char '+', etiquetage fg, etiquetage fd)
                                  (*Random.bool donne un true avec probabilité 0.5 alors on veut que la probabilité soit 0.25 qlors 0.5*0.5=0.25
                                    en bool 0.5&&0.5=0.25 alors Random.bool() && Random.bool() = 0.25 de probabilité*)
;;
```

Test

Partie 1

```
# type monome = int * int
# type polynome = monome list
```

Tester la fonction canonique :

```
let listetest : polynome = [(120,3); (-2,2); (10,14); (-5,2); (0,2)];;
canonique(listetest);;
```

```
# val insere : monome -> polynome -> polynome = <fun>
# val canonique : polynome -> polynome = <fun>
# val listetest : polynome = [(120, 3); (-2, 2); (10, 14); (-5, 2); (0, 2)]
# - : polynome = [(10, 14); (120, 3); (0, 2); (0, 2)]
```

Tester la fonction poly_add

```
let ltest1 : polynome = [(3,3); (5,2); (3,1)];;
let ltest2 : polynome = [(10,4); (2,3); (2,2); (1,1)];;
poly_add ltest1 ltest2;;
poly_add [] ltest2;;
poly_add ltest1 [];;
```

```
# val poly_add : polynome -> polynome -> polynome = <fun>
# val ltest1 : polynome = [(3, 3); (5, 2); (3, 1)]
# val ltest2 : polynome = [(10, 4); (2, 3); (2, 2); (1, 1)]
# - : polynome = [(10, 4); (5, 3); (7, 2); (4, 1)]
# - : polynome = [(10, 4); (2, 3); (2, 2); (1, 1)]
# - : polynome = [(3, 3); (5, 2); (3, 1)]
```

Tester la fonction poly_add

```
poly_prod ltest1 ltest2;;
let ltest3 : polynome = [(1,1); (2,2)];;
let ltest4 : polynome = [(5,0);(3,1); (-4,2)];;
poly_prod ltest3  ltest4;;
poly_prod [(123,0)] [(1, 1)];;
poly_prod [] [(2, 10)];;
poly_prod ltest1 [];;
poly_prod [] [];;
```

```
# val poly_prod : polynome -> polynome -> polynome = <fun>
# - : polynome = [(30, 7); (56, 6); (46, 5); (19, 4); (11, 3); (3, 2)]
# val ltest3 : polynome = [(1, 1); (2, 2)]
# val ltest4 : polynome = [(5, 0); (3, 1); (-4, 2)]
# - : polynome = [(10, 2); (6, 3); (-8, 4); (5, 1); (3, 2); (-4, 3)]
# - : polynome = [(123, 1)]
# - : polynome = [(2, 10)]
# - : polynome = [(3, 3); (5, 2); (3, 1)]
# - : polynome = []
```

Partie 2

```
# type arbre =
  NodeInt of int
| NodePower of int
| NodePlus of arbreP list
| NodeMulti of arbreM list
and arbreP = NodeIntP of int | NodePowerP of int | NodeMultiP of arbreM list
and arbreM = NodeIntM of int | NodePowerM of int | NodePlusM of arbreP list
```

```
let contruireArbre = NodePlus([
          NodeMultiP([ NodeIntM 123; NodePowerM 1 ]);
          NodeIntP 42;
          NodePowerP 3
          ]);;
```

```
       # val contruireArbre : arbre =
             NodePlus
              [NodeMultiP [NodeIntM 123; NodePowerM 1]; NodeIntP 42;
       NodePowerP 3]
```

```
let contruireArbre2 = NodePlus([
                NodeMultiP([ NodePlusM([NodeIntP 3 ; NodeIntP 3; NodeIntP (-
1)]);  NodePowerM 15 ]);
                NodeIntP 20;
                NodeMultiP([ NodeIntM 20 ; NodePowerM 4])
                ])

    ;;
```

```
       # val contruireArbre2 : arbre =
         NodePlus
          [NodeMultiP
            [NodePlusM [NodeIntP 3; NodeIntP 3; NodeIntP (-1)]; NodePowerM
       15];
           NodeIntP 20; NodeMultiP [NodeIntM 20; NodePowerM 4]]
```

Tester la fonction arb2poly :

```
arb2poly contruireArbre ;;
arb2poly contruireArbre2;;
```

```
       # val arbM2poly : arbreM list -> polynome = <fun>
       val arbP2poly : arbreP list -> polynome = <fun>
       # val arb2poly : arbre -> polynome = <fun>

       # - : polynome = [(123, 1); (42, 0); (42, 0)]
       # - : polynome = [(20, 4); (5, 0)]
```

Partie 3

Tester la fonction *extraction_alea :*

```
# val remove : int -> int -> int list -> int list * int = <fun>
# val extraction_alea : int list -> int list -> int list * int list = <fun>
```

```
let x = (extraction_alea [0;1;2;3;4] [5;6;7;8;9]);;
      # val x : int list * int list = ([0; 1; 2; 3], [4; 5; 6; 7; 8; 9])
```

Tester la fonction *gen_permutation :*

```
# val insere_liste : int -> int list -> int list = <fun>
# val create_liste : int -> int list = <fun>
# val gen_permutation : int -> int list = <fun>
```

```
gen_permutation 5;;
      # - : int list = [1; 3; 5; 4; 2]
```

ABR :

```
# type abr = Feuille | Noeud of int * abr * abr
# val insert : int -> abr -> abr = <fun>
# val construireARB : int list -> abr -> abr = <fun>
```

```
construireARB [4;2;3;8;1;9;6;7;5] Feuille;;
      #   - : abr =
      Noeud (4,
       Noeud (2, Noeud (1, Feuille, Feuille), Noeud (3, Feuille, Feuille)),
       Noeud (8,
        Noeud (6, Noeud (5, Feuille, Feuille), Noeud (7, Feuille, Feuille)),
        Noeud (9, Feuille, Feuille)))
```

Tester la fonction *etiquetage :*

```
# etiquetage : abr -> abr = <fun>
```

```
etiquetage (construireARB [4;2;3;8;1;9;6;7;5] Feuille);;
      # - : abr =
      Noeud (43,
       Noeud (43,
        Noeud (42, Noeud (131, Feuille, Feuille), Noeud (120, Feuille,
      Feuille)),
        Noeud (42, Noeud (46, Feuille, Feuille), Noeud (120, Feuille,
      Feuille))),
       Noeud (43,
        Noeud (43,
         Noeud (42, Noeud (-13, Feuille, Feuille), Noeud (120, Feuille,
      Feuille)),
         Noeud (42, Noeud (160, Feuille, Feuille), Noeud (120, Feuille,
      Feuille))),
        Noeud (42, Noeud (-118, Feuille, Feuille), Noeud (120, Feuille,
      Feuille))))
```