DEVOIR DE PROGRAMMATION

UE OUVERTURE

M1 - STL

2021/2022

Joumana ELDAKAR

Sommaire

[Introduction 2](#_Toc89014208)

[Partie 1 3](#_Toc89014209)

[Polynôme sous forme linéaire 3](#_Toc89014210)

[Partie 2 5](#_Toc89014211)

[Expression Arborescent 5](#_Toc89014212)

[Partie 3 7](#_Toc89014213)

[Synthèse d’expressions arborescentes 7](#_Toc89014214)

[Arbre Binaire de Recherche 9](#_Toc89014215)

[Test 10](#_Toc89014216)

[Partie 1 10](#_Toc89014217)

[Partie 2 11](#_Toc89014218)

[Partie 3 12](#_Toc89014219)

# 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 . xd est un couple (c, d) ∈ Z x N.  
**c** = coefficient de x **d** = puissance de x

Un polynôme est une liste de monôme

##### **Structure de données** pour manipuler des polynômes (coef, puissance de x) :

type **monome** **=** int **\*** int;;

type **polynome** **=** monome **list**;;

##### 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(n2).  
La fonction **canonique** fait un appel de la fonction **insere** n fois alors O(n2).

##### 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(n2).

##### 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.

**[**(1,1); (2,2)**]** **[**(5,0); (3,1); (-4,2)**]**

###### Complexité

La complexité au pire cas est O(n2) :

* **Produit** a comme complexité O(n)
* Et **poly\_add** : O(n2)

# Partie 2

## ****Expression Arborescent****

**Grammaire** **:**

**E** = int | E∧ | E+ | E∗

**E∧**= x ∧ int+

**E+**= (E \ E+) + (E \ E+) + . . .

**E∗**= (E \ E∗) ∗ (E \ E∗) ∗ . . .

**E** contient est un entier, **E∧** , **E+** ou **E∗**

**E∧**: **x** avec un nombre entier positif pour sa puissance

**E+****:** la somme des **E** sauf **E∗ (**ne contient pas un **E∗ )**

**E∗:**  la multiplication des **E** sauf **E+** **(**ne contient pas **E∗)**

1. **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.

**+**

**/   |   \**

**\*   42    ^**

**/ \       / \**

**123   ^     x   3**

**/ \**

**x   1**

##### **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

**]**)

    ;;

##### 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****

##### 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

;;

##### La fonction **gen\_permutation :**

L’algorithme de shuffle de Fisher-Yates.

1. 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**
2. Générer une liste vide **P**
3. 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****

type **abr** **=**

**|** **Feuille**

**|** **Noeud** of int **\***  abr **\***  abr

;;

##### La structure de **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))

;;

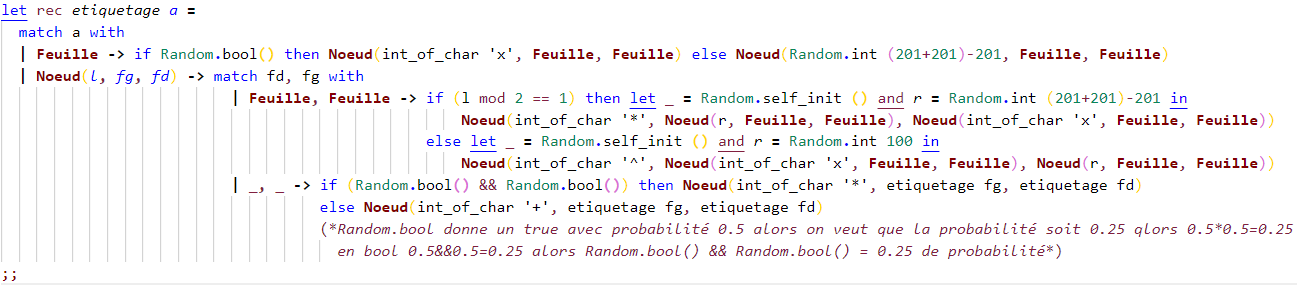
let rec *construireARB* *l* *a* **=**

  match l with

**|** **[]** **->** a

**|** *h*::*t* **->** construireARB t (insert h a)

##### La fonction **etiquetage** :

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

## 

# 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))))