UE Ouverture **Devoir de Programmation**

---

*Devoir à faire en **trinôme**. Langage de programmation **OCaml**.*
Les enseignants décideront d'un ordre de passage pour une soutenance le mardi matin 10/12/2024.
Rapport et Code Source à rendre dans une archive nommée Nom1_Nom2_Nom3_OUV.zip *(sous moodle) au plus tard le 08/12 à 23h59.*

1 Présentation

Le but du problème consiste à manipuler deux modèles de structure de données : l'une sous forme linéaire et l'autre arborescente. Des expérimentations sur l'efficacité d'évolution des structures arborescentes ainsi que de leur transformation sous forme linéaire concluront le travail.
Il est attendu un soin particulier concernant la programmation, dans le paradigme fonctionnel, et par rapport à la réflexion et la mise en place des expérimentations.

1.1 Polynôme sous forme linéaire

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

Question 1.1 Définir une structure de données permettant de manipuler des polynômes.

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 Définir une fonction `canonique` qui prend un entrée un polynôme sous forme linéaire et qui renvoie sa représentation canonique. Analyser la complexité de la fonction (indiquer une mesure de complexité pertinente pour cette fonction, et la complexité au pire cas pour cette mesure).

Question 1.3 Définir une fonction `poly_add` qui prend en entrée deux polynômes canoniques et qui renvoie leur somme sous forme canonique. Analyser la complexité.

Question 1.4 Définir une fonction `poly_prod` qui prend en entrée deux polynômes canoniques et qui renvoie leur produit sous forme canonique. Analyser la complexité.

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 :

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

Nous remarquons en particulier :
- int représente un entier, x la variable formelle et int^+ un entier strictement positif ;
- les expressions de la sous-famille $E_\wedge$ représentent les puissances de la variable x ;
- les expressions de la sous-famille E_+ sont des sommes *d'au moins deux* expressions dont les opérateurs principaux ne sont pas des sommes ;
- les expressions de la sous-famille E_* sont des produits *d'au moins deux* expressions dont les opérateurs principaux ne sont pas des produits.

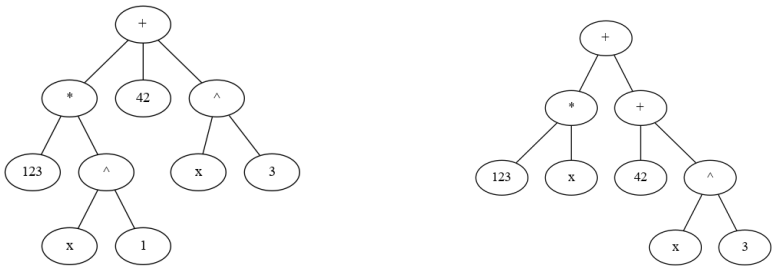


FIGURE 1 – Arbres représentant $123 \cdot x + 42 + x^3$

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).

Question 1.5 Définir une structure de données permettant de manipuler ces arbres.

Question 1.6 Définir une expression représentant l'arbre à gauche de la Figure 1.

Question 1.7 Définir une fonction `arb2poly` prenant en entrée une expression arborescente et la transformant en un polynôme canonique.

1.3 Synthèse d'expressions arborescentes

Cette partie permet de construire un générateur d'expressions arborescentes.

La première étape consiste à construire la structure arborescente, la deuxième étape s'attache à étiqueter la structure suivant des règles peu contraintes, et enfin la troisième partie consiste à modifier localement l'arbre étiqueté afin qu'il satisfasse la grammaire décrite dans la section précédente.

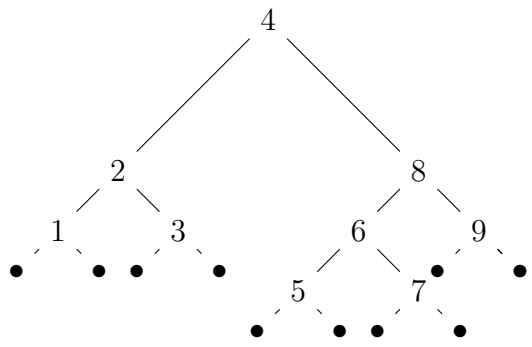
Question 1.8 Implémenter une fonction `extraction_alea` prenant en entrée deux listes d'entiers, notée L et P. Appelons ℓ le nombre de valeurs que contient L. La fonction choisit aléatoirement un entier r entre 1 et ℓ, puis retourne un couple de listes dont la première est la liste L dans laquelle on 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).

Question 1.9 Implémenter une 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`.

L'algorithme que nous venons d'implémenter est *l'algorithme de shuffle de Fisher-Yates*. Il génère une permutation (aléatoirement de manière uniforme : toutes les permutations ont la même probabilité d'être construites).

Pour rappel, un arbre binaire est : (1) soit réduit à une feuille, (2) soit décomposable en une racine qui est un nœud interne et qui pointe vers deux enfants ordonnés, l'enfant gauche et l'enfant droit.

On peut dès lors définir un ABR. C'est un arbre binaire dont les nœuds internes sont étiquetés (par des entiers distincts) de telle sorte que la racine possède une étiquette plus grande que toutes les étiquettes de l'enfant gauche, et plus petite que toutes les étiquettes de l'enfant droit. Récursivement, les deux enfants de la racine sont des ABR.



Question 1.10 Implémenter une fonction `ABR`, qui étant donnée une liste d'entiers tous distincts en entrée, construit l'ABR associé à cette liste. Pour rappel, on insère une nouvelle valeur toujours dans une feuille. Ainsi, si la liste est vide, on renvoie

l'arbre réduit à une feuille. Sinon, pour insérer l'entier ℓ en tête de liste, on parcourt la branche partant de la racine de l'arbre actuel allant à une feuille, en utilisant les étiquettes des nœuds internes pour progresser. Par exemple, en un nœud étiqueté par k, si $\ell < k$ on insérera ℓ dans l'enfant gauche de k, et sinon ce sera dans l'enfant droit.

Ainsi, l'arbre représenté ci-dessus est construit à partir de la liste $[4, 2, 3, 8, 1, 9, 6, 7, 5]$.

Remarque : plusieurs listes différentes peuvent donner le même ABR.

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 $[2, 1, 3, 4]$.

Question 1.12 Définir une fonction `gen_arb` prenant en entrée une structure issue de la question précédente et reconstruisant une expression équivalente qui vérifie la grammaire stipulée dans la Section 1.2.

Remarquons que si l'on applique la fonction `gen_arb` à l'arbre de droite de la Figure 1, on souhaite retrouver l'arbre de gauche de la Figure 1.

2 Expérimentations

Pour les trois questions suivantes, prendre successivement $n = 100, \ldots, 1000$ avec un pas de 100, et représenter les temps de calculs sur les mêmes courbes (soit *addition* soit *produit*).

Question 2.13 Générer n ABR de taille 20, puis les transformer avec les fonctions `etiquetage` et `gen_arb`.

Question 2.14 On veut calculer le polynôme canonique issu de la somme des n arbres précédents. Proposer et analyser plusieurs (au moins 3) stratégies et représenter le temps de calcul nécessaire sur un graphique *addition*.

Question 2.15 On veut calculer le polynôme canonique issu du produit des n arbres précédents. Proposer et analyser plusieurs (au moins 3) stratégies et représenter le temps de calcul nécessaire sur un graphique *produit*.

Question 2.16 Générer 15 ABR de tailles respectives $1, 1, 2, 4, 8, \ldots 2^{13}$ puis les transformer avec les fonctions `etiquetage` et `gen_arb`.

Question 2.17 On veut calculer le polynôme canonique issu de la somme des 15 arbres précédents. Proposer et analyser plusieurs (au moins 3) stratégies et calculer le temps de calcul nécessaire.

Question 2.18 On veut calculer le polynôme canonique issu du produit des 15 arbres précédents. Proposer et analyser plusieurs (au moins 3) stratégies et représenter le temps de calcul nécessaire.

Question 2.19 (Facultatif) Répéter les questions 2.16, 2.17 et 2.18 plusieurs fois afin d'obtenir des temps moyens.