# Expression arithmétique, pile, listes chaînées et arbres

Considérons l'expression suivante :

$$a+b*c$$

La multiplication doit être exécutée avant l'addition, c'est-à-dire que l'on doit sommer a et b $*$ c. La multiplication a une priorité supérieure à l'addition. De même, la division est prioritaire sur l'addition (et la soustraction).

Considérons l'expression suivante :

$$a-b+c$$

Elle se calcule comme $(a-b)+c$ et non pas comme $a-(b+c)$. On dit que les opérateurs addition et soustraction (qui ont la même priorité) sont associatifs à gauche. De même l'expression :

$$a/b * c$$

se calcule comme $(a/b) * c$ et non pas comme $a/(b * c)$. Les opérateurs multiplication et division sont associatifs à gauche. L'opérateur de puissance est prioritaire sur la multiplication, et il est associatif à droite avec lui-même.

Comme nous venons de le voir, la notation usuelle des expressions mathématiques fait appel à des règles implicites, auxquelles nous sommes habitués. Pour rendre ces règles explicites, il faudrait ajouter des parenthèses dans les expressions. D'autre part, il est souvent nécessaire d'utiliser des parenthèses pour écrire ce type d'expressions.

La notation $a + b$ pour l'addition est appelée notation infixe, car le symbole + est placé en infixe par rapport aux opérandes. Une autre notation consiste à placer l'opérateur en préfixe :

$$+ab$$

Cette notation est aussi appelée notation polonaise, en raison de la nationalité du mathématicien qui l'a introduite. Une notation plus intéressante consiste à placer l'opérateur en suffixe (postfixe en anglais) :

$$ab+$$

C'est la notation suffixée, appelée aussi notation polonaise inversée. Elle est utilisée dans certaines calculatrices HP, ou dans certains langages informatiques (Forth, Postscript). Cette notation n'est pas très commode pour l'écriture manuscrite, car il faut bien séparer les deux opérandes. En revanche, elle a l'avantage de dispenser complètement des parenthèses. Considérons par exemple l'expression en notation infixe suivante :

$$a*(b+c)$$

En notation suffixée, elle peut s'écrire :

$$abc+*$$

Une expression suffixée est très facile à évaluer pour un programme informatique. On utilise pour cela une pile contenant les opérandes. L'expression est parcourue de la gauche vers la droite. Lorsqu'un opérande est rencontré, elle est placée au sommet de la pile. Lorsqu'un opérateur est rencontré, les deux derniers opérandes sont enlevés de la pile, l'opération leur est appliquée, et le résultat est placé au sommet de la pile. Lorsqu'une fonction est rencontrée, le dernier opérande est retiré de la pile, la fonction lui est appliquée, et le résultat est placé au sommet de la pile.

**Question 1:**
Montrer comment, d'après le paragraphe précédent, on évalue grâce à une pile l'expression *abc+**

- Écrire la classe pile avec les fonctions empiler/dépiler/vide, **cette classe doit être implémentée avec une liste chaînée.**
- Écrire la fonction évaluer qui prend un argument de type string correspondant à une expression suffixée et qui retourne le résultat de l'évaluation de celle-ci avec cette méthode :
  on utilise pour cela une pile ; l'expression est parcourue de la gauche vers la droite.

    - Lorsqu'un opérande est rencontré, il est placé au sommet de la pile.
    - Lorsqu'un opérateur est rencontré, les deux derniers opérandes sont enlevés de la pile, l'opération leur est appliquée, et le résultat est placé au sommet de la pile.

- N'oubliez pas de bien tester la fonction

**Question 2:**
Transformer une expression infixée en une expression suffixée

Pour évaluer une expression infixée, une méthode consiste à la convertir tout d'abord en expression suffixée. L'algorithme de la gare de triage (shunting yard) permet de faire cette conversion. Les éléments de l'expression infixée (l'entrée) sont traités de la gauche vers la droite. Les opérandes sont ajoutés à une liste de sortie. Les opérateurs sont ajoutés dans une pile d'attente. Lorsqu'un opérateur de priorité inférieure à celui du sommet de la pile est traité, celui-ci est enlevé de la pile et placé dans la liste de sortie. Lorsque tous les lexèmes sont traités, la pile est vidée dans la liste de sortie.

Voici l'algorithme complet:

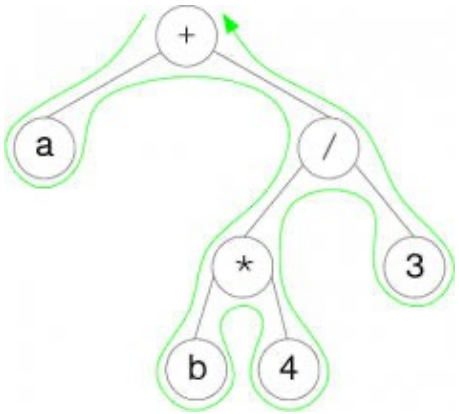
- Si l'élément est un opérande, l'ajouter à la liste de sortie.
- Si l'élément est un opérateur, vider la pile vers la sortie tant que la priorité de l'opérateur est inférieure ou égale à celle de l'élément du haut de la pile. Si l'associativité de l'opérateur est à droite, la condition inférieure strictement doit être utilisée.
- Si l'élément est (, le placer dans la pile.
- Si l'élément est ), vider la pile vers la sortie jusqu'à rencontrer ). Enlever ) de la pile.
- Lorsque l'entrée est vide, vider la pile dans la sortie.

**Question 3:**
Construire un arbre binaire contenant une expression
Pour distinguer les noeuds contenant un opérateur des noeuds contenant une valeur utilisez la classe noeud suivante :

```
class noeud
{
    char type ; //'o' pour opérateur et 'f' pour valeur.
    char ope ;
    float val ;
    noeud * fg, * fd ;

} ;
```

Voici un exemple:




Cet arbre représente l'expression a+((b*4)/3).
On commence par transformer l'expression infixe en une représentation suffixe, en utilisant la méthode décrite précédemment mais en utilisant une pile d'adresses de Noeuds.
Nous allons procéder de la même manière que lors de l'évaluation, mais au lieu de mettre dans la pile les opérandes eux-mêmes, on mettra des pointeurs sur des Noeuds contenant les opérandes. L'expression suffixée est lue de gauche à droite, si l'élément lu n'est pas une opération, alors on le met dans un Noeud (alloué dynamiquement) dont on empile l'adresse. Lorsqu'on rencontre une opération, on crée un noeud N représentant cette opération puis on dépile un élément qui constituera le fils gauche de N et un autre élément qui sera le fils droit de N. L'adresse de N est ensuite empilée. Le processus s'arrête lorsqu'on traite tous les éléments de l'expression. La pile contiendra à ce moment-là l'adresse de la racine de l'arbre.

Écrire les classes noeud et arbre avec entre autres les différents éléments:
- Constructeur sans argument
- Constructeur (de Arbre) avec une expression arithmétique normale (infixe)
- Destructeur
- Une fonction évaluation qui évalue un arbre
- Écrire une fonction qui affiche une expression contenue dans un arbre de manière infixe avec les parenthèses nécessaires.

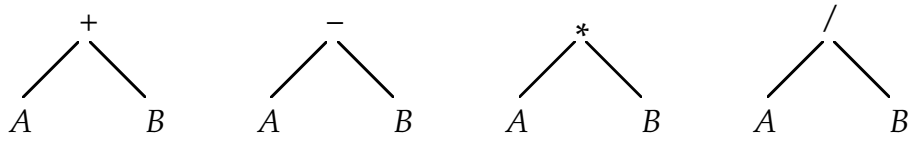
## Question 4

On va maintenant considérer des expressions arithmétiques construites à partir de nombres, variables et opérateurs. Plus précisément, les expressions étudiées pourront être :

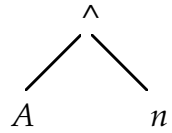
- un **nombre flottant** *r*. Dans ce cas, l'expression sera représentée par un arbre réduit à un seul nœud contenant *r*,
- l'une des **variables** *X*, *Y* ou Z. Dans ce cas, l'expression sera représentée par un arbre réduit à un seul nœud contenant cette variable,

ou bien, si *u* et *v* sont des expressions représentées par les arbres *A* et *B*, et si *n* est un nombre entier :

- la *somme*, la *différence*, le *produit* ou le *rapport* de *u* et *v*. Dans ces cas, les expressions obtenues seront respectivement représentées par les arbres :




- la *mise à la puissance n*, avec *n* entier, de *u*. Pour éviter toute ambiguïté, le moins unaire de l'opposé sera noté par le caractère « souligné » ('_'). Dans ces cas, les expressions obtenues seront respectivement représentées par les arbres :




Adapter les classes arbre et noeuds précédentes pour créer ces arbres.

Écrire les fonctions membres permettant de fabriquer l'arbre codant la dérivée d'une expression par rapport à une variable prise en paramètre.

Rappels (y en a-t-il besoin?) : Si l'on note e' la dérivée en X de l'expression e,

- r' = 0, si r est un nombre,
- X' = 1,
- Y' = Z' = 0,
- (u + v)' = u' + v',
- (u – v)' = u' – v',
- (u.v)' = u'.v + u.v',
- (u/v)' = (u'.v – u.v') / $v^2$,
- ($u^n$)' = n.u'.$u^{n-1}$

```
// X + 3
arbre A1(new Noeud('+',
           new Noeud('X'),
           new Noeud(3)));
```

```
// 4 * X ^2 - 2 * y + (-Z)
arbre A2(new Noeud('-',
           new Noeud('*',
             new Noeud(4),
             new Noeud('^',
               new Noeud('X'),
               new Noeud(2))),
           new Noeud('+',
             new Noeud('*',
               new Noeud(2),
               new Noeud(Y)),
             new Noeud('_',
              new Noeud('Z'))))));
```