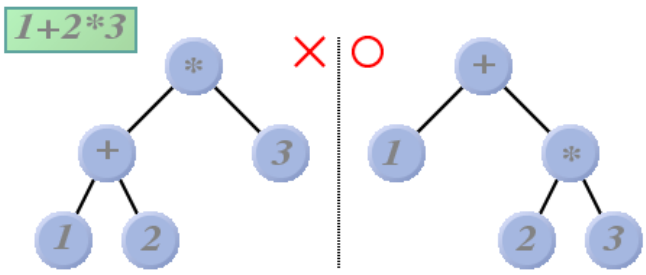
Arbres et calcul formel

Nous allons dans ce TP utiliser une structure d'arbre binaire afin de représenter des expressions mathématiques contenant des constantes et des variables formelles reliées par des opérateurs. On cherchera ensuite à réaliser des calculs simples et des transformations sur ces expressions.

Représentation arborescente d'une expression mathématique

Dans cette première partie, une expression ne contient que des opérateurs et des constantes. Les constantes sont des entiers et nous nous limitons aux opérateurs + et * (addition et multiplication).

La figure ci-dessous représente (à droite) un arbre correspondant à l'expression mathématique 1+2*3, en tenant compte de la priorité de la multiplication sur l'addition.



Si on parcourt l'arbre en :

- pré-ordre on obtient : + 1 * 2 3
- in-ordre on obtient : 1 + 2 * 3
- post-ordre on obtient : 1 2 3 * +

La notation in-ordre nécessite une connaissance de la priorité des opérateurs et l'utilisation de parenthèses pour définir une expression sans ambigüité. Nous allons dans ce TP utiliser la notation post-ordre, également appelée notation polonaise inversée, afin de simplifier la saisie d'expressions.

Création d'un arbre à partir d'une notation post-ordre

On se propose de pouvoir créer un arbre à partir de la notation post-ordre. On accède à un arbre à partir de son nœud racine. La structure d'un nœud est la suivante :

```
typedef struct Node {
      struct Node * left ;  /* fils gauche */
      struct Node * right ; /* fils droit */
      char name ;           /* nom de l'opérateur */
      int value ;           /* valeur de la constante */
} Node ;
typedef Node* NodePtr ;     /* pointeur sur un noeud */
```

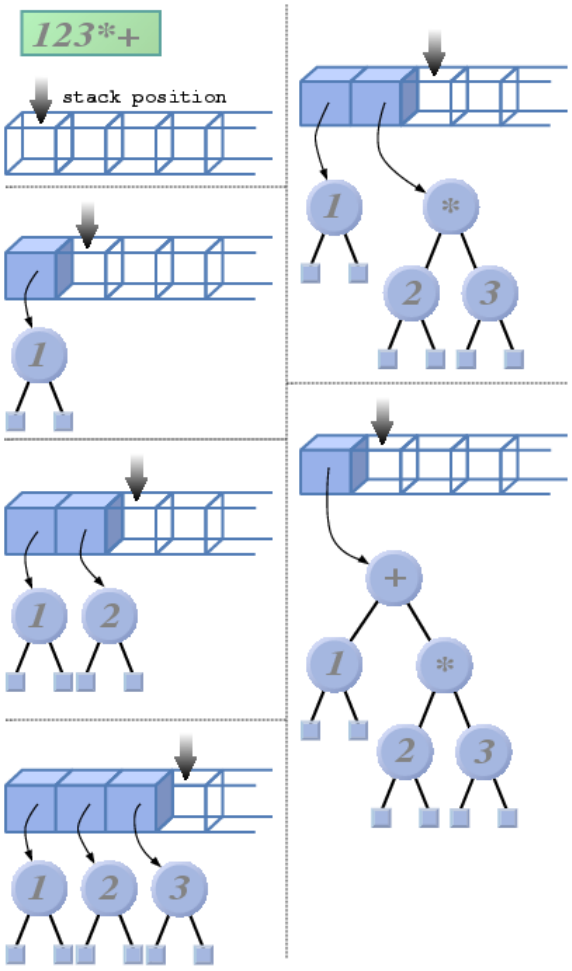
Le champ *name* contient '+' ou '*' si le nœud correspond à un opérateur. Si le nœud est une constante, le champ *name* contient le caractère nul '\0', et le champ value la valeur de la constante.

Écrire la fonction `NodePtr saisie_expression()` qui permet la création d'un arbre à partir d'une expression en notation polonaise inversée lue au clavier. La création de l'arbre se base sur une pile, stockant les nœuds à insérer.

I. Pour chaque entrée, un nœud doit être créé.
   I. Si le caractère est un nombre, il doit être inséré dans la pile
   II. Si le caractère est un opérateur, deux caractères sont sortis de la pile et le caractère d'entrée doit être inséré dans la pile. Le premier caractère sortant bifurque à gauche et le deuxième caractère bifurque à droite de l'opérateur.
   III. On termine la saisie avec la touche « entrée », si la pile ne contient qu'un seul élément.

On considérera que la saisie d'un opérateur ou d'un caractère d'espacement permet de terminer la saisie d'un nombre.

La figure ci-dessous résume la création d'un arbre à partir de l'expression saisie.



Parcours de l'arbre

Ecrivez et testez les fonctions de récursivité suivantes pour le parcours de l'arbre :

- void pre_ordre(NodePtr node) ;
- void in_ordre(NodePtr node) ;
- void post_ordre(NodePtr node) ;

Affichage de l'expression

Écrivez une fonction `void affiche_expression(NodePtr node)` qui affiche à l'écran l'expression représentée par l'arbre, en mettant des parenthèses afin de représenter l'ordre des calculs. L'arbre de l'exemple produira l'affichage suivant :

```
1+(2*3)
```

Duplication de l'arbre

Écrivez une fonction `NodePtr clone(NodePtr node)` qui permet de créer une copie de l'arbre désigné par la racine node, et retourne un pointeur vers le nouvel arbre créé.

Calcul numérique

Écrire une fonction `void calcul_intermediaire(NodePtr node)` qui réduit la profondeur de l'arbre en effectuant les calculs dont les deux opérandes sont des feuilles de l'arbre.

L'arbre de l'exemple, après un appel de calcul_intermediaire, devra devenir

```
  +
 / \
1   6
```

Ecrire la fonction `void calcul(NodePtr node)` qui effectue tous les calculs possibles. L'arbre obtenu sera donc réduit à un nœud unique contenant le résultat numérique.

Calcul formel

Nous introduisons maintenant la notion de variable. Les variables sont représentées par les lettres de l'alphabet. Elles sont stockées dans l'attribut *name* du nœud.

Ajout de variables

Modifier la fonction saisie_expression afin de pourvoir entrer des variables dans les expressions.

Modifier les fonctions calcul_intermediaire et calcul afin de prendre en compte la présence de variables dans l'arbre. Seuls les calculs mettant en jeux deux opérandes numériques seront effectués. La multiplication d'une variable par 1 ou zéro, ainsi que l'addition de 0 à une variable seront également gérées.

Développement

Créer une fonction `void developpement(NodePtr node)` qui transforme l'arbre de calcul sous une forme développée.

Par exemple, l'expression `3*(b+c)` devient `(3*b)+(3*c)`.
De même, l'expression `(3+a)*(b+c)` devient `((3*b)+(3*c))+((a*b)+(a*c))`

Dérivation

Créer une fonction `void derivation(NodePtr node, char v)` qui dérive selon la variable *v* l'expression représentée par l'arbre.

Pour rappel, les règles de dérivation classique selon l'addition et la multiplication sont :

- (f+g)' = f' + g'
- (f*g)' = f'*g + f*g'

Exemple : l'expression `(3+a)*(b+c)` devient `b+c` après dérivation selon la variable *a*.

Remarques Générales

Un menu utilisateur permettant de lancer toutes les fonctions développées pendant ce TP, devra être programmé

D'autres fonctions seront éventuellement nécessaires au bon fonctionnement de l'ensemble des modes de traitement demandés. A vous de les identifier et de les développer

Un rapport d'au plus 4 pages sera à rendre, contenant une introduction, les choix de programmation, le calcul de complexité des principaux algorithmes et une conclusion.

Pas de code source dans le rapport.

Le programme doit être composé d'au moins 5 fichiers :

- arbre.h et arbre.c contiennent les définitions de structures et les fonctions de manipulation des arbres de calcul
- pile.h et pile.c contiennent les définitions d'une structure de pile
- tp5.c : contient la fonction *main()*.