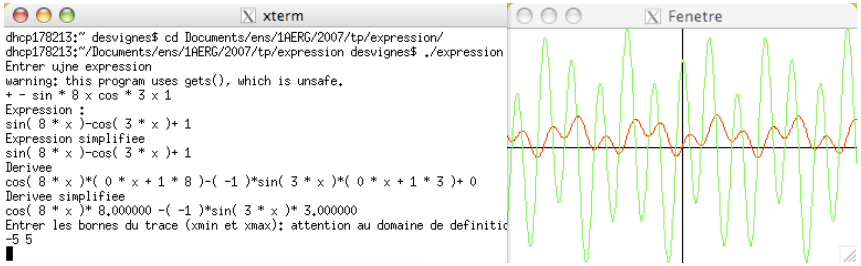
TD n° 13 : Expressions arithmétiques

Buts : Pointeurs, Récursivité, Arbres binaires.
Durée : 2 semaines

Les logiciels qui manipulent les expressions arithmétiques comme l'éditeur d'équation de Word ou les logiciels de calcul symbolique (mapple) doivent avoir une représentation interne de ces expressions. Pour faciliter ces manipulations, on utilise une représentation en arbre. Vous allez réaliser une application qui trace une expression entrée au clavier, calcule sa dérivée formelle et trace cette dérivé, comme dans l'exemple ci dessous.

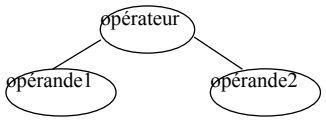


Toute expression arithmétique correctement formée est une succession d'opérations binaires (+ ,- ,*,/,^^) ou unaires (sin, cos, tan, etc..). L'écriture usuelle est une écriture infixée, qui impose des parenthèses en fonction des priorités des opérateurs utilisés. Ainsi, les expressions `9*x-6` ou `(9*x)-6` sont identiques, alors que `9*(x-6)` est différente. La forme générale d'une opération est donc (**`opérande1 opérateur opérande2`**).

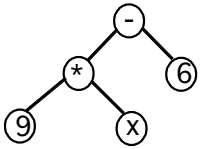
L'écriture préfixée permet de supprimer les parenthèses, moyennant un peu de gymnastique intellectuelle. Il suffit de représenter une opération en commençant la notation par cette opération : **`opérateur opérande1 opérande2`**. Les parenthèses deviennent inutiles et l'opération `(9*x)-6` s'écrit alors `- * 9 x 6`

Graphiquement, une opération binaire est représentée par un arbre binaire dont la racine détient l'opérateur et dont les fils représentent respectivement le premier et le second opérande. Chaque fils peut être lui même une expression binaire. Une expression est donc formée récursivement d'éléments de base du type :

(**opérateur opérande1 opérande2**) se représente graphiquement par



l'expression **((9*x)-6)** se représente par

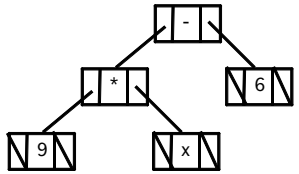


Une expression arithmétique est donc un arbre binaire où chaque nœud est soit :

- un **nœud** qui contient une chaîne de caractères représentant les opérateurs ['+', '-', '*', '/'] et possède exactement deux fils. Chacun de ses fils est elle-même une expression.
- un **nœud** qui contient une chaîne de caractères représentant les opérateurs [sin, cos, tan, sqrt] et possède un fils unique. Ce fils est une expression. Par convention, le fils droit sera NULL.
- une **feuille** qui contient une chaîne de caractères représentant soit une variable soit un nombre et dont les 2 fils sont nuls.

Un nœud de l'arbre est donc représenté par une structure :

L'expression précédente se traduit par l'arbre ci-dessous :



Structure de données

Les structures de données que vous allez utiliser sont les suivantes :

```
typedef
  enum { OPERATEUR_BINAIRE, OPERATEUR_UNAIRE, VALEUR, VARIABLE }
       TYPE;
```

TYPE est un type qui représente les 4 constantes possibles: OPERATEUR_BINAIRE, OPERATEUR_UNAIRE, VALEUR, VARIABLE. Il sera utilisé pour connaître la nature du noeud de l'arbre (voir la fonction creerneoud par exemple).

```
typedef
  struct noeud {
    char* val;
    TYPE type;
    struct noeud* fg, *fd; }* ARBRE;
```

ARBRE : c'est le type permettant de définir les noeuds d'un arbre. fd et fg sont les fils gauche et droit (un peu comme suiv est l'élément suivant d'une liste). Les informations utiles pour notre application sont les champs val, qui contient une chaîne de caractères (un nom de fonction comme "sin", un opérateur comme "+", une variable comme "x" ou un nombre) et le champ type, qui indique quelle est la nature du noeud et qui peut prendre les valeurs OPERATEUR_BINAIRE, OPERATEUR_UNAIRE, VALEUR, VARIABLE

Les fonctions déjà réalisées

Les fonctions qui permettent de lire une expression préfixée au clavier et de créer l'arbre correspondant sont déjà réalisées. Elles se trouvent dans le fichier arbre.c

- `ARBRE lire(char* s)` qui transforme la chaîne de caractère s en écriture préfixée en un arbre que vous pourrez ensuite utiliser. Les fonctions gérées par cette fonction sont les opérateurs + ,- ,*,/,^, sin,cos,tan,sqrt. Elle retourne NULL si l'expression est mal formée.
- `ARBRE creernoeud(char* s)` : construit un unique noeud correspondant à la chaîne s. Cette fonction alloue la mémoire nécessaire à un noeud, copie la chaîne s dans ce nœud et positionne le type correct (opérateur binaire, unaire, variable, nombre).
- `ARBRE simplifie(ARBRE r)` : simplifie si possible l'arbre r et retourne l'arbre simplifié. L'arbre initial peut être modifié. Les simplifications concernent les multiplications par 1 ou 0, les additions avec 0, les additions et les multiplications de constantes.
- `ARBRE libere(ARBRE r)` : libère la mémoire allouée à l'arbre r et retourne NULL

Travail à réaliser

1. Ecrire la fonction d'affichage `void affiche(ARBRE r)` : affiche l'expression "r" en notation normale, avec les parenthèses nécessaires. Cette fonction affiche ((9*x)-6) à l'écran avec l'exemple précédent. Pour afficher l'expression sous la forme **`(op1 operateur op2)`**, il suffit d'utiliser une fonction récursive dont la forme générale est :
   a. on affiche d'abord l'opérande 1, c'est à dire qu'on affiche le fils gauche
   b. on affiche l'opérateur, c'est le contenu du noeud
   c. on affiche l'opérande 2, c'est à dire qu'on affiche le fils droit

   Cette fonction s'écrit en 5 lignes.

   Faire un programme qui lit une expression au clavier en notation préfixée, puis affiche cette expression en notation infixée. Tester ce programme sur plusieurs exemples.

2. Ecrire la fonction `double eval(ARBRE r, double x)` qui calcule la valeur de l'expression "r" pour la valeur donnée par x. On utilise la récursivité :
   - Si le noeud "r" est un nombre, la valeur retournée par eval est le nombre lui même
   - Si le noeud "r" est une variable, la valeur retournée par eval est la valeur de x
   - Si "r" est un +, la valeur retournée par eval est la somme de la valeur du sous arbre gauche (l'opérande1) pour x (il suffit d'utiliser la fonction eval sur le sous arbre gauche, ie le fils gauche) ET de la valeur du sous arbre droit (l'opérande 2) pour x (il suffit d'utiliser la fonction eval sur le sous arbre droit, ie le fils droit)
   - etc...

   Cette fonction s'écrit en 20 lignes au maximum, dont la moitié sont similaires.

   Faire un programme qui lit une expression préfixée au clavier, et affiche la valeur de cette expression pour différentes valeurs de la variable.

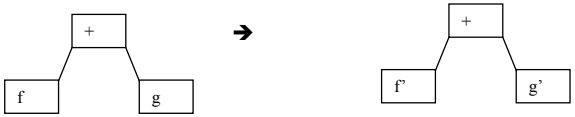
   Fonctions utiles :

   `double atof(char* s)` convertit une chaîne de caractère contenant un nombre en réel double précision.

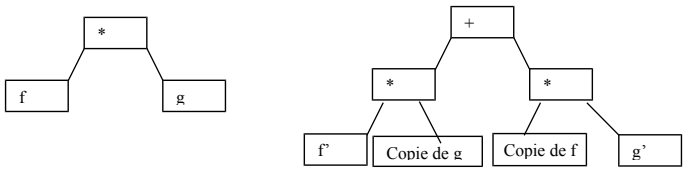
   `int strcasecmp(char* s1, char* s2)` compare les deux chaînes s1 et s2 sans tenir compte des majuscules et minuscules. Cette fonction sera utile pour déterminer quelles sont les fonctions mathématiques utilisées (sin, cos, sqrt, etc…)

3. Tests : le fichier expression1.c lit une chaîne au clavier et affiche l'expression en notation classique, puis affiche le tracé de la fonction dans une fenêtre graphique avec les 2 axes. La fonction plot (fichier traceexp.c) trace la courbe, mais utilise la fonction eval ci dessous. Il faut donc vérifier que cette fonction eval est correcte.

4. Ecrire la fonction `ARBRE copie(ARBRE r)` qui copie une expression en récréant une nouvelle expression identique: tous les nœuds et feuille de l'arbre d'origine sont recopiés dans l'arbre copié. Pour recopier l'expression "r", il faut donc créer un **nouveau** nœud "a" dont les champs val et type seront identiques à ceux de r, puis mettre dans le fils gauche de "a" la copie du fils gauche de "r" et mettre dans le fils droit de "a" la copie du fils droit de "r". Cette fonction s'écrit en 6 lignes.
5. Tester la fonction copie grâce au fichier expression2.c. On lit une expression au clavier, on la copie dans une autre expression, on libère la première, on affiche la première (elle est vide) puis on affiche la copie et on trace la copie.
6. Écrire une fonction `ARBRE deriv(ARBRE r)` qui dérive une expression arithmétique par rapport à la variable (x par exemple). La dérivée de l'expression f+g est f'+g', celle de f*g est f'*g+f*g', etc… L'arbre dérivé est un arbre indépendant de l'expression d'origine :
   - Si le noeud "r" est un nombre constant, on retourne un nœud contenant la valeur '0'.
   - Si le noeud "r" est la variable x, on retourne un nœud contenant la valeur '1'.
   - Si le noeud "r" est un +, on a alors une expression du type **`f+g ie (le fils gauche) + (le fils droit)`**. Sa dérivé est donc f'+g'. Il faut donc créer un nœud "a" de type opérateur contenant '+'. Il faut mettre f' (la dérivé du fils gauche du noeud "r") dans le fils gauche de "a". De même, le fils droit contient la dérivé du fils droit du noeud "r"

   - Si c'est un *, la dérivé est un peu plus complexe et l'arbre dérivé est :

   Le code de cette fonction deriv pour l'opérateur plus ressemblera donc à cela :

```
ARBRE deriv(ARBRE r) { ARBRE c;
  if (r!=NULL)  /* L'expression n'est pas vide */
    switch(r->type)  {
      case OPERATEUR_BINAIRE : /*  Une expression binaire*/
        if (r->val[0]=='+' ) {  /* C'est un + */
           c=creernoeud("+");   /* Cration d'un noeud + */
           c->fg=deriv(r->fg);  /* Copie de la dérivée du fg*/
           c->fd=deriv(r->fd);  /* Copie de la dérivée du fd*/
           return c ;     /* On retourne l'arbre dérivé créé*/
        } ......
```

   Commencer par écrire la fonction dérivée pour les opérateurs + et –, les variables et les constantes. Tester votre fonction en vérifiant que l'arbre dérivé est correct sur des exemples simples (3+x, x + x , x + x + x –x, x + 2 –x + 8 +x -7)
7. Tests : le fichier expression3.c lit une expression, calcule sa dérivée, la simplifie, l 'affiche et trace la courbe initiale et la dérivé dans une fenêtre graphique