TP 14 : Expressions arithmétiques symboliques

Notions : Arbres

1 Préambule

Ce TP se réalise sur 2 séances encadrées sans compter le travail personnel à la maison. Le sujet comporte deux parties obligatoires, portant sur la notion d'arbres et travaillant les parcours, modifications et créations d'arbres.

Une troisième partie, pour ceux qui sont en avance, concerne les automates et grammaires et permet d'entrer des expressions infixées classiques sous forme d'arbre. Les notions abordées concernent les grammaires, la récursivité et les arbres.

Pour ceux qui n'ont pas pu installer les bibliothèques graphiques, nous avons réalisé des versions des fichiers de tests qui enregistrent les résultats dans un fichier `donnees.csv` qui peut alors être utilisé par un tableur comme excel ou un logiciel de tracé de de courbes comme gnuplot.

2 Introduction

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 dérivée, comme dans l'exemple ci dessous.

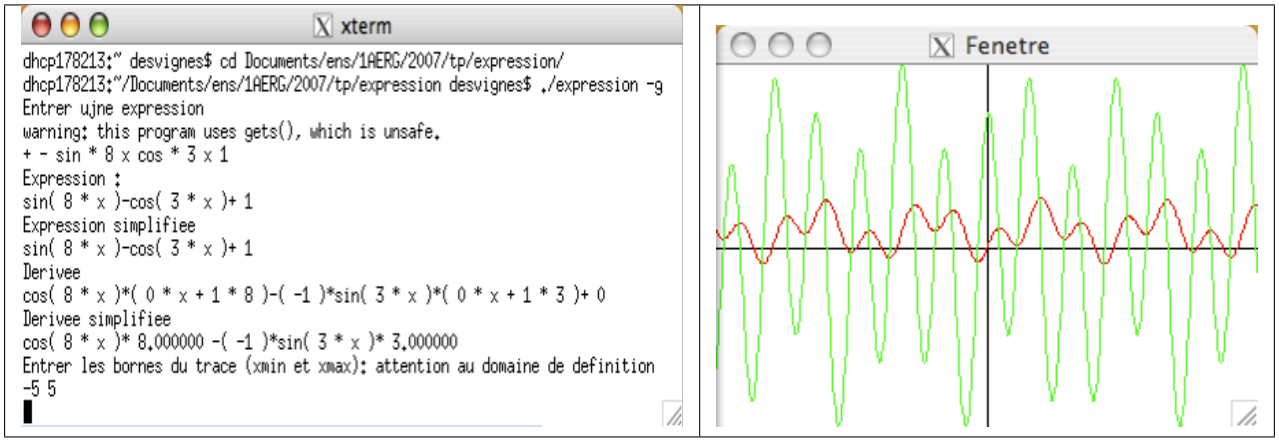
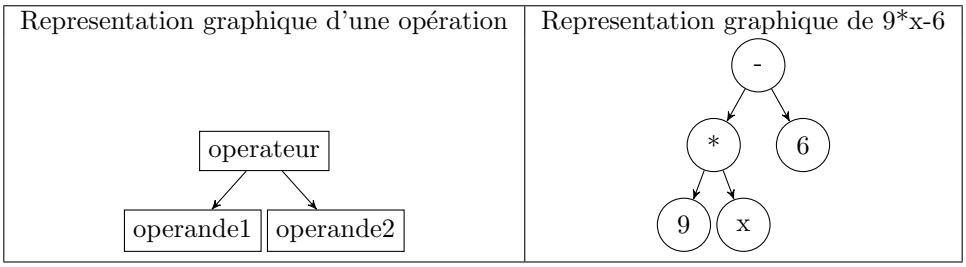


FIGURE 1 – Exemple

3 Représentation d'une expression

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 `(opérande1 opérateur opérande2)`.



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

Graphiquement, une expression 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 (des noeuds) du type :

— un nœud qui contient une chaîne de caractères représentant les opérateurs `"+","-","*","/"]` et possède exactement deux fils. Chacun de ses fils est lui-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 gauche 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 comportant au moins 3 champs : la valeur du noeud, un pointeur vers le sous arbre gauche (éventuellement vide), un pointeur vers le sous arbre droit (éventuellement vide).

3.1 Structure de données : `element_t`, `binarytree_t` et `expression_t`

3.1.1 La structure `element_t`

La structure `element_t` est fournie dans le fichier `element.h` :

```
#ifndef _ELEMENT
#define _ELEMENT
#include <string.h>

// Type enumere pour une lecture plus simple du code
typedef enum { PASDEFINI, OPERATEUR_BINAIRE, OPERATEUR_UNAIRE, VALEUR, VARIABLE } TYPE;

/* Un neoud contient 2 informations :
     - une chaine de caracteres
     - la nature (type) de la chaine
*/
typedef
  struct {
    char* string;
    TYPE type;
  } element_t;

// Principales fonctions utiles sur les elements
// Afficher la valeur d'un element
```

```
void element_print (element_t e);
// Tester l'egalite de 2 elements
int element_equal(element_t*, element_t*);
// Cloner et copier un element
element_t element_clone(element_t e);
// Creer un nouvel element a partir d'une chaine et d'un type
element_t element_new(char*, TYPE );
// Detruire et liberer la memoire allouee a un element
element_t element_delete(element_t e);

// Definit le type de x
TYPE element_type(char* x);
#endif
```

`TYPE` est un type qui représente les 5 constantes possibles pour la valeur d'un noeud : `PASDEFINI`, `OPERATEUR_BINAIRE`, `OPERATEUR_UNAIRE`, `VALEUR`, `VARIABLE`. Il sera utilisé pour connaître la nature du noeud de l'arbre (voir la fonction `expression_node_new` ou la fonction `expression_derivation` dans la partie 2 du sujet par exemple).

Toutes les fonctions déclarées dans le fichier `element.h` sont déjà implantées dans le fichier `element.c`.

3.1.2 La structure `binarytree_t`

La structure `binarytree_t` est fournie dans le fichier `binarytree.h` :

```
#ifndef _BINARYTREE
#define _BINARYTREE
#include "element.h"

typedef struct noeud {
  element_t value;
  struct noeud* left, *right; } node_t,* binarytree_t;

    // Principales fonctions
  // Detruit et libere le neoud r uniquement
binarytree_t node_del(binarytree_t r) ;
  // Retourne 1 si l'arbre est vide, 0 sinon
int binarytree_is_empty(binarytree_t);
  // Creer un arbre vide
binarytree_t binarytree_new();
  // Detruit et libere tous les noeuds d'un arbre
binarytree_t binarytree_del(binarytree_t r) ;

  // Fonctions a ecrire
  // Clone et copie tous les noeuds d'un arbre
binarytree_t binarytree_clone(binarytree_t b);
  // Retourne 1 si tous les noeuds de l'arbre sont egaux
int binarytree_equal(binarytree_t r1, binarytree_t r2) ;

#endif
```

`binarytree_t` : c'est le type permettant de définir les noeuds d'un arbre. `left` et `right` sont les fils gauche et droit, donnant accès aux sous arbres gauche et droit.

Les informations utiles pour notre application sont regroupées dans le champ `value` qui est définit comme un élément. Cet élément contient une chaîne de caractères (un nom de fonction comme "sin", un opérateur comme "+", une variable comme "x" ou un nombre comme "3.14159") et le champ `type`, qui indique quelle est la nature du noeud et qui peut prendre les valeurs `PASDEFINI`, `OPERATEUR_BINAIRE`, `OPERATEUR_UNAIRE`, `VALEUR`, `VARIABLE`.

Le fichier `binarytree.c` contient le code des 4 premières fonctions ; Ce fichier sera à compléter avec le code des 2 autres fonctions dans la deuxième partie.

3.1.3 La structure `expression_t`

La structure `expression_t` est fournie dans le fichier `expression.h` :

```
typedef binarytree_t expression_t;

  // Creation d'un noeud dans un arbre d'expression a partir d'une chaine
expression_t expression_node_new(char *c);
  // Creation d'un arbre d'expression a partir d'une chaine
expression_t expression_sscanf(char *s) ;

    // Fonctions a ecrire
  // Affichage infixe d'une expression
void expression_print_infixe(expression_t);
  // Retourne la valeur de l'arbre d'expression pour la valeur de x
double expression_eval(expression_t r, double x);
  // Retourne un arbre correspondant a la derivation de l'arbre d'expression r
expression_t expression_derivation(expression_t r);
  // Retourne 1 si les 2 expressions sont egales
int expression_equal(expression_t r1, expression_t r2) ;
  // Retourne un arbre d'expression correspondant à la derivation symbolique
  // Simplifie a la construction les cas particuliers. Par exemple, 1+x ==> 1 et non 0 + x
expression_t expression_deriv_simplify(expression_t r);
#endif
```

Une expression est donc simplement un arbre binaire.

Pour simplifier les illustrations, on dessinera directement un noeud sous la forme de 4 champs :

— `left` pour le fils gauche `left`

— `string` pour la chaine de caractères désignant un operateur `value.string`

— `string` pour l'entier désignant un operateur `value.type`

— `right` pour le fils droit `right`

L'arbre représentant `9*x+6` est celui présenté figure 2

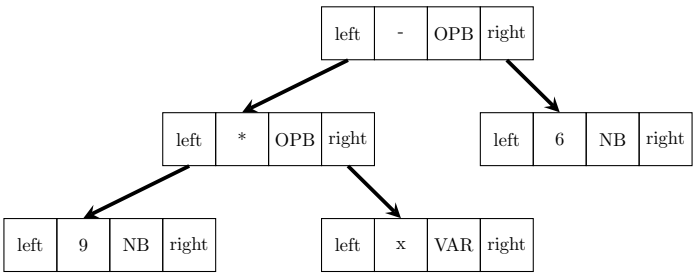


FIGURE 2 – Arbre 9*x+6

Les fonctions suivantes pour créer un arbre à partir d'une chaîne de caractères sont implantées dans le fichier `expression.c`,

— `binarytree_t expression_sscanf(char *s);` retourne un arbre d'expression construit à partir de la chaîne de caractère `s` en écriture **préfixée**. Les opérateurs et fonctions prévues sont les opérateurs $+, -, *, /,$ ' $sin, cos, tan, sqrt, log$. Elle retourne `NULL` si l'expression est mal formée. L'expression "+ * 9 x 6" conduit à la figure 2.

— `binarytree_t expression_node_new(char *c);` : construit et retourne un unique noeud correspondant à la chaîne `s`. Cette chaîne est un unique opérateur ou valeur ou variable, comme par exemple "sin" ou "*" ou "3.145". Cette fonction alloue la mémoire nécessaire à un noeud, alloue et recopie la chaîne `s` dans le champ `value` de l'élément du noeud et **positionne le champ `type` selon la nature de la chaine `s`** (`OPERATEUR_BINAIRE, OPERATEUR_UNAIRE, VALEUR, VARIABLE`).

4 Première partie : affichage et évaluation d'une expression

4.1 Affichage d'un arbre

4.1.1 Version basique

Pour afficher l'expression sous la forme (op1 operateur op2), il suffit d'utiliser une fonction récursive qui s'écrit en quelques lignes dont la forme générale est :

1. on affiche d'abord l'opérande1 (le fils gauche) entre parenthèses
2. on affiche l'opérateur, c'est le contenu du noeud
3. on affiche l'opérande 2 (le fils droit) entre parenthèses

4.1.2 Version standard

Pour améliorer cet affichage qui comporte beaucoup trop de parenthèses, on remarque que les parenthèses sont nécessaires uniquement

— lorsqu'on affiche un noeud multiplicatif `*` ou `/`, si l'opérande gauche (resp. droit) est un opérande additif `+` ou `-`, alors cet opérande gauche (resp. droit) doit être mis entre parenthèses.

— lorsqu'on affiche un noeud `-` et `^`, l'opérande 2 (fils droit) doit être mis entre parenthèses.

4.1.3 Travail à réaliser

1. Complétez le fichier `expression.c` en écrivant la fonction d'affichage `void expression_print_infixe (expression_t r);` qui affiche l'expression `r` en notation infixée (habituelle), avec les parenthèses nécessaires. Cette fonction affiche ((9*x)-6) à l'écran avec l'exemple précédent.
2. Faire un programme qui lit une expression au clavier en notation préfixée, puis affiche cette expression en notation infixée. Utiliser la fonction `fgets` pour la lecture (cf fichier `test_exp1.c`). Tester ce programme sur plusieurs exemples.

4.2 Evaluation d'un arbre

4.2.1 Algorithme

Lorsqu'une expression dépend d'une variable, on peut calculer la valeur du résultat pour différentes valeurs de cette variable. On utilise la récursivité pour trouver la valeur de l'expression dont la racine est le noeud r :

— Si le nœud r est un nombre, la valeur de l'expression est le nombre lui même. La conversion d'une chaine de caractères s en nombre se fait grâce à la fonction `double atof(char* s)` ou à la fonction `double strtod(char* s, char** endprt).`

— Si le noeud r est une variable, la valeur de l'expression est la valeur de x

— Si r est un '+', la valeur de l'expression est la somme de la valeur du sous arbre gauche (l'opérande1) pour x (il suffit d'utiliser la fonction récursivement sur le sous arbre gauche) ET de la valeur du sous arbre droit (l'opérande 2) pour x (il suffit d'utiliser la fonction récursivement sur le sous arbre droit)

— Si c'est un '-', ...

4.2.2 Travail à réaliser

1. Complétez le fichier `expression.c` en écrivant la fonction `double expression_eval(expression_t root , double x)` qui calcule la valeur de l'expression `root` pour la valeur donnée par `x`. En suivant l'algorithme précédent, cette fonction s'écrit en 20 lignes au maximum, dont la moitié sont similaires.
2. Faire un programme qui lit une expression préfixée au clavier, et affiche la valeur de cette expression pour différentes valeurs de la variable.

4.3 Premier test graphique

Le fichier `test_exp1.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.

Vous pouvez le tester en compilant avec la commande unix : `make test_exp1` et en l'executant avec la commande `test_exp1`.

Si vous n'avez pas la bibliothèque graphique, utiliser le fichier `test_exp1_file.c`, en compilant avec la commande unix : `make test_exp1_file`. Ce programme produit un fichier `donnees.csv` qui contient les valeurs x et $f(x)$ de l'expression entrée au clavier. Vous pouvez ensuite utiliser un tableau comme excel ou un logiciel de tracé de courbe pour visualiser le résultat sous forme de courbes.

Pour gnuplot, après avoir lancé gnuplot, il faut utiliser la commande `plot 'donnees.csv' u 1:2 with line lt 4 title 'courbe'` pour afficher la courbe contenue dans le fichier.

5 Deuxième partie : dérivation d'une expression

5.1 Copie d'arbre

5.1.1 Algorithme

La copie d'arbre consiste à dupliquer ou cloner tous les noeuds de l'arbre pour obtenir une copie indépendante de l'arbre d'origine (pas de partage de pointeurs). C'est une fonction récursive.

Pour recopier l'expression `root` **non vide**, il faut créer un nouveau nœud `a` dont les champs `string` et `type` de la partie `value` seront identiques à ceux de `root`, puis mettre dans le fils gauche de `a` la copie du fils gauche de `root` et mettre dans le fils droit de `a` la copie du fils droit de `root`.

5.1.2 Travail à réaliser

1. Complétez le fichier `binarytree.c` en écrivant la fonction `binarytree_t binarytree_clone(binarytree_t b)` qui copie une expression en récréant une nouvelle expression identique : tous les nœuds et feuilles de l'arbre d'origine sont recopiés dans l'arbre copié. Cette fonction s'écrit en 6 lignes.
2. Tester la fonction `binarytree_clone` grâce au fichier `test_exp2.c` [1]. Ce programme 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.

   Compiler avec la commande unix : `make test_exp2`.

   Si la fonction de copie est mal réalisée, l'exécution produira une erreur de segmentation ou un message d'erreur de l'allocateur mémoire du type `malloc: *** error for object 0x7fecc244b5d0: pointer being freed was not allocated .....`

   La commande `valgring ./test_exp2` doit se terminer avec une ligne du type `HEAP SUMMARY`, `in use at exit: 0 bytes in 0 blocks`, indiquant que toute la mémoire a été libérée.

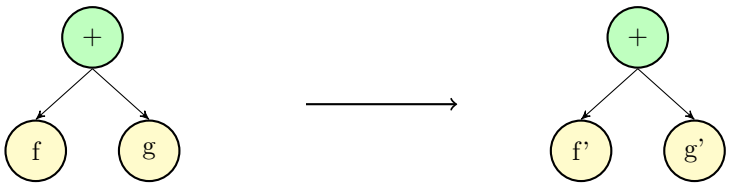
5.2 Dérivation

5.2.1 Algorithme

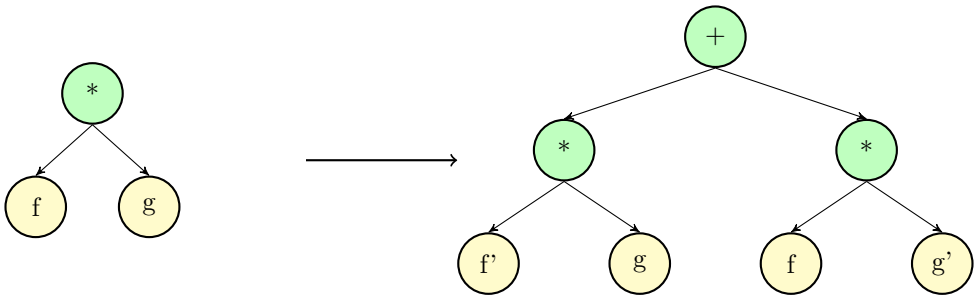
La dérivée formelle de l'expression $f+g$ est $f'+g'$, celle de $f*g$ est $f'*g+f*g'$, etc... A partir de l'arbre représentant une expression, on peut construire un nouvel arbre représentant la dérivée formelle.

Par exemple, pour un opérateur "+",

1. ou bien `test_exp2_file.c`, en compilant avec la commande unix : `make test_exp2_file` si vous n'avez pas la bibliothèque graphique. Le fichier `donnees.csv` contient les valeurs x et $f(x)$.



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



L'arbre dérivé est un arbre indépendant de l'expression d'origine. La fonction est récursive et l'algorithme pour construire la dérivée de l'arbre de racine `root`ressemble alors à :

- Si le noeud `root` est un nombre, sa dérivée est nulle et on retourne un nœud contenant la valeur '0'.
- Si le noeud `root` est la variable x, sa dérivée est 1 et on retourne un nœud contenant la valeur '1'.
- Si le noeud `root` est un +, on a alors une expression du type f+g ie (le fils gauche) + (le fils droit). Sa dérivée est donc f'+g'. Il faut donc créer un nœud `deriv` de type opérateur contenant '+'. Il faut mettre f' (la dérivée du fils gauche du noeud `root`) dans le fils gauche de `deriv`. De même, le fils droit du noeud `deriv` contient la dérivée du fils droit du noeud `root`.
- Si le noeud `root` est un *, c'est .....

Le code de cette fonction `expression_derivation` pour l'opérateur '+' ou pour une variable ressemblera donc à cela :

```
expression_t expression_derivation(expression_t r) { expression_t deriv;
  if (r!=NULL)
    switch(r->value.type)  {
      case VARIABLE : return expression_node_new("1");
      case OPERATEUR_BINAIRE :
        if (r->value.string[0]=='+' || r->value.string[0]=='-' ) {
           deriv=expression_node_new(r->value.string);
           deriv->left=expression_derivation(r->left);
           deriv->right=expression_derivation(r->right);
           return deriv;
        }

     } ......
```

5.2.2 Travail à réaliser

1. Complétez le fichier `expression.c` ave la fonction `expression_t expression_derivation(expression_t r)` qui dérive une expression arithmétique par rapport à la variable (x par exemple). 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 ($3+x, x+x, x+x+x–x, x+2–x+8+x-7$)
2. Le fichier `test_exp3.c`[2] lit une expression, calcule sa dérivée, l'affiche et trace la courbe initiale et la courbe de la dérivée dans une fenêtre graphique. Compiler avec la commande unix : `make test_exp3` et tester votre code en entrant des expressions préfixées.

2. ou bien `test_exp3_file.c`, en compilant avec la commande unix : `make test_exp3_file` si vous n'avez pas la bibliothèque graphique. Le fichier `donnees.csv` contient les valeurs x, $f(x)$ et $f_x(x)$.

Fonction	Dérivée
$f(x)+g(x)$	$f'(x)+g'(x)$
$f(x)-g(x)$	$f'(x)-g'(x)$
$f(x)*g(x)$	$f'(x)*g(x)+f(x)*g'(x)$
$\frac{f(x)}{g(x)}$	$\frac{f'(x)*g(x)-f(x)*g'(x)}{g^2(x)}$
$f(g(x))$	$g'(x).f'(g(x))$
$sin(f(x))$	$f'(x).cos(f(x)$
$cos(f(x))$	$-f'(x).sin(f(x)$
$exp(f(x))$	$f'(x).exp(f(x)$
$log(f(x))$	$\frac{f'(x)}{log(f(x))}$

TABLE 1 – Tableau des principales dérivées

3. Compléter ensuite la fonction `expression_t expression_derivation(expression_t r)` pour des expressions plus complexes à l'aide du tableau 1
4. Telle qu'est exprimé l'algorithme, l'expression dérivée crée des parties inutiles dans certains cas : par exemple, si l'expression est `1+x`, la dérivée sera `0+1`, ce qui peut facilement être évité à la construction.

   Pour cela, Vous pouvez écrire une fonction `binarytree_t expression_sum(binarytree_t f1, binarytree_t f2)` : cette fonction a pour rôle de construire et retourner l'arbre `(+ f1 f2)` dans le cas général, excepté dans les cas suivants :

   — `f1` est la constante 0 : on libère la mémoire occupée par `f1` et la fonction `expression_sum` retourne `f2`

   — `f2` est la constante 0 : on libère la mémoire occupée par `f2` et la fonction `expression_sum` retourne `f1`

   — `f2` et `f1` sont des constantes NON nulles : on réalise le calcul, on libère la mémoire occupée par `f1` et par `f2` et la fonction `expression_sum` retourne un noeud unique contenant la constante calculée.

   Idem pour les autres opérations pour lesquelles on peut prendre en compte dès la construction les éléments neutres et absorbants et écrire des fonctions :

   `binarytree_t expression_product(binarytree_t f1, binarytree_t f2)`,

   `binarytree_t expression_diff(binarytree_t f1, binarytree_t f2)` : si `f2` et `f1` sont identiques, la fonction retourne un unique noeud contenant la constante '0'

   `binarytree_t expression_div(binarytree_t f1, binarytree_t f2)` : si `f2` et `f1` sont identiques, la fonction retourne un unique noeud contenant la constante '1'

   `binarytree_t expression_pow(binarytree_t f1, binarytree_t f2).`

   Pour cela, vous aurez besoin d'une fonction `int expression_equal(expression_t r1, expression_t r2)` qui retourne 1 si les expressions sont égales, 0 sinon. Attention : 2 expressions sont égales si leurs racines portent la même valeur et si le fils gauche de `r1` est identique au fils gauche de `r2` et si le fils droit de `r1` est identique au fils droit de `r2`. Dans le cas où la racine est un opérateur commutatif (+ ou *), il y a aussi égalité si leurs racines portent la même valeur et si le fils gauche de `r1` est identique au fils droit de `r2` et si le fils droit de `r1` est identique au fils gauche de `r2`. Cette fonction doit être écrite dans le fichier `expression.c`

6 Troisième partie : Analyse syntaxique

Pour passer de l'écriture infixée à l'écriture préfixée (ou directement à l'arbre représentant les expressions), il faut utiliser un analyseur syntaxique dont le rôle est de vérifier si l'expression est correcte et de gérer les erreurs. Pour cela, on fait appel à la notion de grammaire formelle (voir CHOMSKY). Une grammaire comporte 4 objets différents :

1. l'alphabet A : Symboles terminaux ou unités lexicales qiu correspondent à tous les symboles de nos expressions : `'0'..'1','+','-','*','/','sin'`, etc...
2. l'alphabet X : Symboles intermédiaires ou catégories grammaticales qui correspondent à des états intermédiaires dans l'analyse que l'on va faire.
3. Règles de grammaire de type $x- > w$, avec $x \in X$ et $w \in (A \cup X)^*$
4. L'axiome à partir duquel on démarre l'analyse.

6.1 Grammaire des expressions arithmétiques simplifiée

Pour analyser des expressions ne comportant que des opérateurs `+` ou `*`, des variables `x` ou `y`, des constantes entières, la grammaire sera la suivante :

1. Axiome ou point de départ de l'analyse : `Expression` (voir dans les règles)
2. A = `'0'..'9','.','x','y', '+','*','(',')'`
3. X = Expression, Terme, Facteur, Nombre
4. Règles :
   - Règle 1 : $Expression \rightarrow Terme + Expression$
   - Règle 2 : $Expression \rightarrow Terme$
   - Règle 3 : $Terme \rightarrow Facteur * Terme$
   - Règle 4 : $Terme \rightarrow Facteur$
   - Règle 5 : $Facteur \rightarrow (Expression)$
   - Règle 6 : $Facteur \rightarrow Nombre$
   - Règle 7 : $Facteur \rightarrow Variable$
   - Règle 8 :$Nombre \rightarrow [\prime 0\prime..\prime 9\prime]^+$ [3]
   - Règle 9 : $Variable \rightarrow x$
   - Règle 10 : $Variable \rightarrow y$

6.2 Exemple d'analyse

Pour analyser notre expression $9 * (x + 6)$ qui est correcte, on applique successivement les règles suivantes en partant du symbole de départ.

Appliquer une règle signifie identifier la partie gauche et la partie droite de cette règle avec l'expression en cours d'analyse. Si nous analysons l'expression $x+6$, appliquer la règle 1 signifie que dans cette règle, on considère que $expression = x + 6$. On identifie alors dans la partie droite $Terme = x$, le $+$ de $x + 6$ correspond bien au $+$ de la règle 1, et on identifie $Expression$ de la partie droite par $expression = 6$. Cette règle peut s'appliquer si l'analyse de $Terme = x$ et l'analyse de $expression = 6$ sont toutes les 2 validées récursivement (ce qui sera effectivement vérifié avec l'application des règles 4 et 8 pour $Terme = x$ et des règles 2,4,9 pour $Expression = 6$.

L'analyse de l'expression $9 * (x + 6)$ se fait alors par l'application des règles suivantes :

1. Règle 2 = $Expression : 9 * (x + 6) \rightarrow Terme : 9 * (x + 6)$
2. Règle 3 = $Terme : 9 * (x + 6) \rightarrow Facteur : 9 * Terme(x + 6)$
3. Règle 6 = $Facteur : 9 \rightarrow Nombre : 9$
4. Règle 8 = $Nombre : 9 \rightarrow 9 : c'est\ bien\ un\ nombre.$
   On a bien trouvé un nombre, mais il reste la partie suivante (x+6) à analyser d'après la règle 3 en cours d'analyse à l'étape 2.
5. Règle 4 = $Terme : (x + 6) \rightarrow Facteur : (x + 6)$
6. Règle 5 = $Facteur : (x + 6) \rightarrow (Expression : x + 6)$

3. $[\prime 0\prime..\prime 9\prime]^+$ signifie au moins une fois un chiffre.

7. Règle 1 = $Expression : x + 6 \rightarrow Terme : x + Expression : 6$
8. Règle 4 = $Terme : x \rightarrow Facteur : x$
9. Règle 7 = $Facteur : x \rightarrow Variable : x$
10. Règle 9 = $Variable : x \rightarrow x$: c'est bien une variable autorisée.
    Mais il reste la partie suivante 6 à analyser d'après la règle 1 en cours d'analyse à l'étape 7.
11. Règle 2 = $Expression : 6 \rightarrow Terme : 6$
12. Règle 4 = $Terme : 6 \rightarrow Facteur : 6$
13. Règle 6 = $Facteur : 6 \rightarrow Nombre : 6$
14. Règle 8 = $Nombre : 6 \rightarrow 6$: c'est bien un nombre.
15. Fin de l'expression complète $9 * (x + 6)$, qui est correcte, la règle 2 de l'étape 1 a bien été complètement identifiée.

6.3 Exemple de codage

Pour implanter une telle analyse et réaliser une action (ici, construire l'arbre syntaxique représentant l'expression), avec une grammaire aussi simple, on peut utiliser un schéma récursif. On définit une fonction par règle, implantant directement cette analyse.

6.3.1 Fonction de découpage en lexeme

La premiere chose à réaliser est le découpage de la chaine en lexeme, c'est à dire en mots tels que `sin, log, x, + , 3.156, 2` . Pour cela, la fonction `char* strtok(char* s, char* separateur)` découpe la chaine `s` selon les séparateurs définis par `separateur` en l'utilisant de la manière suivante :

1. Le premier appel à la fonction sous la forme `strtok(s,sep);` initialise le découpage et le pointeur retourné par `strtok` pointe sur le premier mot de la chaine `s` ;
2. Les appels suivants de la forme `strtok(NULL,sep);` retournent successivement un pointeur sur les différents mots de la chaine. **Notez le parametre NULL à la place de s**.
3. Quand le dernier mot est lu, la fonction retourne le pointeur `NULL`.

Pour illustrer son utilisation, le programme ci dessous lit une chaine au clavier, puis affiche les différents mots de la chaine tapée au clavier.

```
#include <string.h>
#include <stdio.h>
int main() { int i=0;
  char* p;          /* Le pointeur vers les differents mots */
  char s[512];    /* Le tableau qui contient la chaine lue au clavier*/
  char *sep=" "; /* Les separateurs sont uniquement l'espace */
  puts("Entrer une chaine de mots au clavier");
  fgets(s,511,stdin); s[strlen(s)-1]=0; /* Lecture de la chaine de mots */
  p = strtok(s,sep); /* On initialise le decoupage : p pointe sur le premier mot, la chaine s est
   modifiee, les separateurs (espaces) sont remplaces par des marques de fin de chaines '\0' */
  do {
    printf("Le %d ieme mot est %s\n",i,p);  /* On affiche le mot */
    p = strtok(NULL, sep);        /* On passe au mot suivant */
    i=i+1;
  } while (p!=NULL);
}
```

L'execution de programme donne :

```
$ ./a.out
Entrer une chaine de mots au clavier
Test de la fonction strtok sur 8 mots
```

```
Le 0 ieme mot est Test
Le 1 ieme mot est de
Le 2 ieme mot est la
Le 3 ieme mot est fonction
Le 4 ieme mot est strtok
Le 5 ieme mot est sur
Le 6 ieme mot est 8
Le 7 ieme mot est mots
```

A retenir : la fonction `strtok` permet de passer au mot suivant dans une chaine de caractères.

6.3.2 Codage des fonctions

On définit une fonction par règle, implantant directement cette analyse. On écrit une fonction `Analyse` qui prend une chaine de caractère infixée et retourne l'arbre. Elle lance donc l'analyse et correspond à l'axiome.

Les fonctions correspondant à chaque groupe de règles (ou à chaque symbole intermédiaire) vérifient la syntaxe de ces symboles intermédiaires et construisent la partie de l'arbre correspondant, comme la fonction `Expression` ci-dessous qui correspond aux règles 1 et 2.

```
ARBRE  erreur(char** r) { printf("Erreur : %s \n",*r); return NULL; }

ARBRE Analyse(char* r){ ARBRE s;  char* t;
    /* strtok permet de separer les mots de la chaine, c'est a dire que si r="3.145 * x + 5.89" apres
    l'execution de t=strtok(r," "); t contient "3.145" et les prochains appels a strtok(NULL," ");
    retourneront alors les valeurs "*", puis ="x",  puis ="+" puis="5.89" */
    t=strtok(r," ");
    /* On cherche l'Axiome, qui est expression. La fonction expression retourne la chaine prefixee.*/
    s=Expression(&t);
    return s;
}

  /* La fonction Expression implemente les regles 1 et 2: on cherche un Terme ou un Terme + Expression
   */
ARBRE Expression(char** pr){
/*
pr est le pointeur sur la chaine a analyser: *pr est donc la chaine a analyser, modifiee par la
   fonction. **pr est donc le premier caractere de la chaine a analyser.
Si tout se passe correctement, *pr pointera la fin de la chaine analysee
et sera donc sur une marque de fin de chaine
Sinon, c'est qu'il y a une erreur. */
    ARBRE s1, s2, res;
    char s3[3];

/* Si la chaine est vide, c'est une erreur de syntaxe */
    if (*pr == NULL) return erreur(pr);

/*
Regle 1 et 2 : une expression comporte d'abord un Terme. Le resultat de ce terme sera dans s1.
pr est mise a jour par la fonction Terme, donc pr pointe maintenant apres la partie deja analysee par
   Terme :
donc soit un + , soit rien; puis on doit trouver une Expression, dont le resultat sera dans s2. Si tout
    est OK, on construit l'arbre + s1 s2.
Par exemple, si on analyse 9 * x + 6, 9*x est le Terme, x est l'expression
*/
  /* Y a t il un Terme en debut ? */
    s1=Terme(pr);
    /* Ici, s1 est l'arbre representant le Terme, sauf si on n'a pas trouve un Terme ce qui est alors
   une erreur, car les regles 1 et 2 imposent  un Terme en premiere position
    Pour 9 * x + 6,  s1 = * 9 x
    */
    if (s1==NULL) return erreur()
    /*  Si il y a un + ensuite, c'est la regle 1 */
    if (*pr && (*pr)[0] == '+'){
    /* s3 contient la chaine "+" qui se trouve dans *pr pour former l'expression finale */
```

```
        strcpy(s3,*pr);
    /* on avance dans l'analyse de la chaine : on passe au mot qui suit le +  */
        *pr=strtok(NULL," ");
    /* cette partie doit etre une expression d'apres la regle 1. */
        s2=Expression(pr);
    /* Si ce n'est pas une expression, il y a donc une erreur de syntaxe ; Pour 9*x+6,  s2=6*/
        if (s2 == NULL) return erreur(r);
    /*  On peut maintenant construire l'arbre final :  + Terme Expression */
    /* On cree le noeud +, avec s1 comme fils gauche et s2 comme fils droit */
        res = expression_node_new(s3); res->left=s1; res->right=s2;
        return ( res);
    }

         /* C'est la regle 2 : il y a juste un Terme, c'est s1 */
    return s1;
}
```

6.4 Travail à réaliser

1. Dans le fichier `analyse.c`, Implanter les fonctions `Analyse, Expression, Terme, Nombre, Fonction, Variable` sur le modèle ci dessus pour générer l'arbre représentant une expression à partir d'une chaine infixée.
2. Utiliser le fichier `test_exp4.c`[4] pour tester vos fonctions : il lit une expression infixée habituelle, trace l'expression et sa dérivée.

7 Grammaire plus complète

La grammaire précédente ne gère que les opérateurs `+` et `*`. On peut bien sûr ajouter les opérateurs `-` et `/` dans les premières règles :

— Règle 1b : $Expression \rightarrow Terme - Expression$

— Règle 4b : $Terme \rightarrow Facteur/Terme$

Attention : Cette grammaire ne gère pas correctement les opérateurs `/` et `-`, qui doivent être parenthésés pour être correctement gérés : `a-b+c` sera vu comme `a-(b+c)`, `a-b-c` sera vu comme `a-(b-c)`. Pour gérer correctement les expressions comportant des opérateurs `-` ou `/` avec cette grammaire, il faut les considérer comme des opérateurs binaires et utiliser les parenthèses comme par exemple `a-(b-c)` et `a-(b+c)` pour les expressions ci dessus.

La grammaire suivante[5] gère correctement les opérateurs `-` et `/`.

— Règles 1,2,3 : $Expression \rightarrow Expression + Terme \mid Expression - Terme \mid Terme$

— Règles 4,5,6 : $Terme \rightarrow Facteur * Terme \mid Facteur/Terme \mid Facteur$

— Règles 7,8,9,10 : $Facteur \rightarrow (Expression) \mid Nombre \mid Fonction\ Facteur \mid Variable$

— Règles 11,12 :$Nombre \rightarrow ['0'..'9']^+ \quad | \quad ['0'..'9']^+.['0'..'9']^+$

— Règles 13 : $Fonction \rightarrow sin \mid cos \mid tan \mid sqrt \mid exp \mid log$

— Règles 14 : $Variable \rightarrow x \mid y$

Malheureusement, cette grammaire est récursive à gauche : la règle 1 est du type $Expression \rightarrow Expression + Terme$. Son implantation récursive, selon l'exemple précédent, placerait un appel récursif infini en début de fonction comme cela :

4. ou bien `test_exp4_file.c` si vous n'avez pas la bibliothèque graphique.
5. Notations : dans les règles suivantes, le symbole | signifie OU et $['0'..'9']^+$ signifie au moins une fois un chiffre. Par exemple, la règle 1 $Expression \rightarrow Terme + Expression$, la règle 2 $Expression \rightarrow Terme - Expression$ et la règle 3 $Expression \rightarrow Terme$ sont regroupées en un groupe : $Expression \rightarrow Expression + Terme \mid Expression - Terme \mid Terme$.

```
ARBRE Expression(char** pr){    ARBRE s1, s2, res;
    char s3[3];
    if (*pr == NULL) return erreur(pr);
      /* CET APPEL RECURSIF EST INFINI !!!! */
    if ( (s1=Expression(pr))==NULL) return erreur()
    if (*pr && (*pr)[0] == '+'){  strcpy(s3,*pr);
        *pr=strtok(NULL," ");
        if ((s2=Terme(pr)) == NULL) return erreur(r);
        res=expression_node_new(s3);  res->left=s1; res->right=s2;
        return ( res);
    }
    return s1;
}
```

La solution consiste à utiliser la grammaire suivante, où le symbole $(+Terme)^*$ signifie la présence de $+Terme$ 0 fois ou plus. La règle 1 correspond ainsi à une suite d'opérations additives comme $a+b+c$, $a+b-c$, $a+b-c-d+e$,

- Règles 1,2 : $Expression \rightarrow Terme(+Terme)^* \mid Terme(-Terme)^*$
- Règles 3 et 4 : $Terme \rightarrow Facteur(*Facteur)^* \mid Facteur(/Facteur)^*$
- Règles 5,6,7,8 : $Facteur \rightarrow (Expression) \mid Nombre \mid Fonction\ Facteur \mid Variable$
- Règles 9 et 10 :$Nombre \rightarrow ['0'..'9']^+ \mid ['0'..'9']^+.['0'..'9']^+$
- Règles 11 : $Fonction \rightarrow sin \mid cos \mid tan \mid sqrt \mid exp \mid log$
- Règles 12 : $Variable \rightarrow x \mid y$

La grammaire peut alors se programmer avec des fonctions récursives (expression utilise facteur qui utilise expression) et itératives comme ci dessous. La boucle `while` implémente les parties $(+Terme)^*$ de la règle 1 et $(-Terme)^*$ de la règle 2. Chaque fois que l'on rencontre un opérateur '+' avec un terme `s1` devant, on cherche ensuite un nouveau terme `s2` et on crée ensuite un noeud '+' avec les termes `s1` et `s2` comme fils.

```
ARBRE Expression(char** r){ ARBRE s1=NULL, s2=NULL, res=NULL;
    char s3[32];
    if (*r == NULL) return erreur(r);
    if ( (s1=Terme(r))==NULL) return erreur(r);
    while (*r && ((**r == '+')|| (**r == '-'))){
        strcpy(s3,*r);
        *r=strtok(NULL," ");
        s2=Terme(r);
        res=expression_node_new(s3);  res->left=s1; res->right=s2;
        s1=res;
    }
    return s1;
}
```

7.1 Travail à réaliser

Modifier vos fonctions d'analyse syntaxique pour prendre en compte une grammaire gérant les 4 opérations et les fonctions mathématiques simples.