# Dérivation formelle

L'objet de ce problème[1] est la réalisation d'un programme de calcul formel, c'est-à-dire de lecture, représentation et transformation de fonctions mathématiques. Il s'agit de faire, principalement :

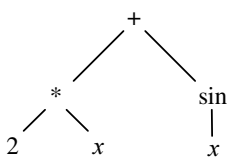
- la *lecture* d'une expression arithmétique représentant une fonction *f* d'une ou plusieurs variables et son codage en mémoire par un arbre binaire ;
- l'*évaluation* de *f* pour des valeurs données des variables qu'elle contient ;
- la construction d'un deuxième arbre qui représente la *fonction dérivée* de *f* par rapport à une de ses variables ;
- la *simplification* de ce dernier arbre, notamment dans le but de corriger certaines maladresses de la dérivation.

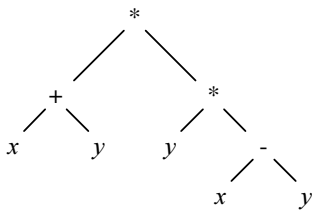
Exemple d'exécution (les interventions de l'utilisateur sont soulignées) :

```
A toi :  2 * x ;
f : 2 * x
Df/Dx : 0 * x + 2 * 1
Df/Dx (simplifiée) : 2
A toi :  (x + y) * z ;
f : (x + y) * z
Df/Dx : (1 + 0) * z + (x + y) * 0
Df/Dx (simplifiée) : z
Df/Dy : (0 + 1) * z + (x + y) * 0
Df/Dy (simplifiée) : z
Df/Dz : (0 + 0) * z + (x + y) * 1
Df/Dz (simplifiée) : x + y
A toi :  .
Au revoir.
```

1. SYNTAXE DES EXPRESSIONS. Les expressions qui nous intéressent sont définies par une grammaire BNF (analogue à celle du problème 3) qu'on explicitera. On prendra en considération les constantes (des nombres réels), les variables *x*, *y*, et *z*, les quatre opérations arithmétiques de base, l'opérateur – « unaire » (changement de signe) et, pour faire bonne mesure, quelques fonctions transcendantes comme *sin*, *cos*, *exp* et *log*.

2. CODAGE INTERNE DES EXPRESSIONS. Chaque expression est représentée par un arbre binaire dont les constantes et les variables occupent les feuilles et les opérateurs et fonctions les autres nœuds. Par exemple, les expressions $2 * x + \sin(x)$ et $(x + y) * y * (x - y)$ sont respectivement représentées par les arbres suivants :




De tels arbres sont faits de nœuds à deux ou trois champs :

- il y a toujours un champ *type*, de type **`char`**, ayant la valeur **`'c'`** si le nœud représente une constante, **`'v'`** s'il représente une variable, **`'f'`** s'il s'agit d'une fonction et l'opérateur en question dans les autres cas ;
- dans le cas d'une constante le nœud a un deuxième champ, de type **`float`**, avec le nombre qui en est la valeur ;
- dans le cas d'une variable, le nœud comporte un deuxième champ, contenant la lettre correspondante ;
- dans le cas d'un opérateur, il y a deux autres champs, contenant les adresses des deux sous-arbres qui représentent les opérandes ;
- enfin, dans le cas d'une fonction, le nœud comporte un champ qui identifie[2] la fonction dont il s'agit et un champ qui contient l'adresse du sous-arbre qui représente l'argument.

[1] Nous supposons ici que vous avez déjà traité le problème 3, qui met en œuvre certains concepts voisins de ceux illustrés ici.

[2] Un bon moyen pour identifier une fonction est son adresse.

3. EVALUATION D'UNE EXPRESSION, c'est-à-dire obtention de la valeur de l'expression pour des valeurs données des variables $x$, $y$ et $z$.

4. DERIVATION D'UNE EXPRESSION. Il s'agit de transformer l'arbre précédemment lu, selon les règles habituelles de la dérivation des fonctions.

5. SIMPLIFICATION DES EXPRESSIONS. Notre fonction de dérivation fournira un résultat correct mais souvent maladroit. Par exemple, l'application stricte des règles de dérivation nous fera trouver la dérivée de $2x$ sous la forme $0 \times x + 2 \times 1$, ce qui est une manière correcte, mais particulièrement lourde, d'exprimer la constante 2.

Nous aurons donc à transformer l'arbre représentant la dérivée d'une fonction, pour corriger ce genre de maladresses. Par exemple, un arbre qui représente la somme de deux constantes pourra être remplacé par une constante, le produit d'une expression quelconque par 0 pourra être remplacé par 0, etc.

5. AFFICHAGE DES EXPRESSIONS. A plusieurs endroits de notre programme nous aurons besoin d'une fonction produisant l'affichage de nos arbres. Ainsi, on affichera l'expression lue, ne serait-ce que pour constater qu'elle a été bien analysée et mémorisée, puis l'expression dérivée, avant et après simplification.

Synthétiser la forme écrite d'une expression à partir de son codage interne, est l'opération inverse de l'analyse faite au n° 1. Comme les fonctions d'analyse, les fonctions de synthèse s'obtiennent à partir de la grammaire, par une démarche systématique, facile à découvrir (il y aura une fonction *afficher_expression*, une fonction *afficher_terme*, une fonction *afficher_facteur*, etc.).

6. GESTION DE LA MEMOIRE. Les objets manipulés dans ce programme (les fonctions) sont représentés par des structures allouées dynamiquement en accord avec deux principes :

- le *partage des structures est autorisé* : une structure créée n'est jamais modifiée, ainsi toute structure peut être construite en utilisant des parties d'une autre structure préexistante ;
- on ne se préoccupe *jamais de libération des maillons* des structures devenues inutiles.

Bien entendu, la mémoire étant limitée, un programme ainsi conçu ne pourra fonctionner longtemps que si on lui adjoint un *mécanisme de recyclage de la mémoire* inutilisée. Celui-ci est ainsi conçu :

- chaque nœud comporte deux champs supplémentaires : un pointeur réalisant le chaînage de *tous les nœuds* alloués, et un booléen (i.e. un entier) utilisé à un certain moment pour marquer les maillons utiles,
- indépendamment de son rôle dans le programme, au moment de son allocation chaque nœud est ajouté à une certaine *liste de tous les nœuds* ;
- afin de pouvoir repérer les nœuds utiles, un tableau global contient les *adresses des structures utiles* (en réalité elles ne sont pas très nombreuses : il s'agit essentiellement de la fonction lue et ses trois dérivées).

Dans ces conditions, recycler la mémoire c'est :

- en parcourant les structures dont les adresses sont dans le tableau des structures utiles, *marquer* les nœuds utiles (un nœud peut être marqué plusieurs fois, cela n'a aucune importance),
- en parcourant la liste de tous les nœuds, *libérer* ceux qui ne sont pas marqués (et, au passage, effacer les marques sur les autres).

REMARQUES SUR LE PROGRAMME.

ANALYSE. L'analyseur syntaxique lit caractère par caractère l'expression tapée par l'utilisateur. Il se compose des fonctions classiques *reconnaitreExpression*, *reconnaitreTerme*, *reconnaitreFacteur*, etc. déduites de la grammaire. Chacune de ces fonctions construit l'arbre représentant l'entité qu'elle a analysé, et rend cet arbre comme résultat.

CODAGE DES EXPRESSIONS. Le plus gros effort à fournir concerne la définition d'une structure pour représenter les nœuds de l'arbre. Il faudra une déclaration du genre :

```
typedef struct noeud {
        ...champs à trouver...
} NOEUD, *EXPRESSION;
```

En même temps que cette structure on définira des fonctions spécialisées d'allocation de nœuds, ou *constructeurs* :

```
EXPRESSION constante(double valeur);
EXPRESSION variable(char nom);
EXPRESSION plus(EXPRESSION operandeGauche, EXPRESSION operandeDroit);
EXPRESSION moins(EXPRESSION operandeGauche, EXPRESSION operandeDroit);
etc.
```

DERIVATION. La fonction de dérivation se présente sous la forme

```
EXPRESSION derivee(EXPRESSION fon, char var);
```

Elle construit et renvoie l'arbre qui représente la dérivée de la fonction correspondant à l'arbre donné en argument, par rapport à la variable donnée en argument.

SIMPLIFICATION. Une manière de traiter le problème de la simplification des expressions consiste à écrire une fonction :

```
EXPRESSION simplifie(EXPRESSION exp);
```

qui construit et renvoie une nouvelle expression qui est la version simplifiée de **exp**.

Une *meilleure* manière de concevoir la simplification des expressions consiste à modifier les « constructeurs » des nœuds, afin qu'ils ne construisent pas des expressions maladroites, c'est-à-dire facilement simplifiables. Ainsi, la construction de l'arbre « 2 + 3 » renvoie la constante 5, la construction de « 0 × *expression* » renvoie la constante 0 quelle que soit l'*expression*, etc.

Notez que, si on choisit cette manière de faire, l'exemple d'exécution montré plus haut n'est plus adapté puisque les dérivées non simplifiées ne sont en réalité pas construites. Pour illustrer le travail à faire il faudrait ici montrer *deux* exécutions : celle d'une première version du programme, dans laquelle on ne s'est pas occupé de simplification :

```
A toi : 2 * x ;
f : 2 * x
Df/Dx : 0 * x + 2 * 1
A toi : (x + y) * z ;
f : (x + y) * z
Df/Dx : (1 + 0) * z + (x + y) * 0
Df/Dy : (0 + 1) * z + (x + y) * 0
Df/Dz : (0 + 0) * z + (x + y) * 1
A toi :
```

et celle d'une version améliorée, avec la simplification :

```
A toi : 2 * x ;
f : 2 * x
Df/Dx : 2
A toi : (x + y) * z ;
f : (x + y) * z
Df/Dx : z
Df/Dy : z
Df/Dz : x + y
A toi :
```

AFFICHAGE. Comme nous l'avons indiqué, l'affichage des arbres est pris en charge par une famille de procédures (*afficher_expression*, *afficher_terme*, *afficher_facteur*, etc.) qui se déduisent aisément de la grammaire.

EVALUATION. Cela se présente comme une fonction

```
double valeur(EXPRESSION expr, double valX, double valY, double valZ);
```

qui calcule et renvoie la valeur de l'expression **expr** lorsque les variables *x*, *y*, *z* ont respectivement pour valeurs **valX**, **valY** et **valZ**.

RECYCLAGE DE LA MEMOIRE. Cette partie du travail est programmée à travers deux fonctions :

```
void marquage(EXPRESSION expression);
```

qui parcourt (récursivement) l'*expression* donnée en marquant chacun de ses nœuds, et

```
void liberation();
```

qui parcourt la liste de tous les nœuds en libérant ceux qui n'ont pas été marqués. Ces fonctions sont appelées soit « sur incident » (lorsque la mémoire vient à manquer), soit périodiquement, à un moment où cela ne provoque pas de ralentissement sensible (exemple : immédiatement après l'affichage des réponses, pendant que l'utilisateur les lit).

☺☺☹