Workshop CAS et évaluation d'expressions

Raphaël *Shugo* Boissel (boisse_r)

23 Novembre 2013

Table des matières

1 Présentation du workshop

1.1 Définition du cas

Le CAS (Computer Algebra System ou système de calcul formel en français), est un système qui a pour objectif de travailler sur des expressions mathématiques qui contiennent par exemple des inconnues (x, y, ...) ou des symboles (π, e, ...). Le CAS peut faire des opérations telles que la dérivé d'une expression ou la recherche des solutions d'une équation.

Exemple d'expression sur laquelle un CAS peut travailler :

$$5 * \pi + 2 * x$$

1.2 Définition de l'évaluation

Contrairement au CAS l'évaluation de l'expression va consister à travailler sur des quantités connues et finies. On utilisera alors des variables dont la valeur a été préalablement définie et les constantes telles que π, e, ... seront approximées. Le but de l'évaluation est de fournir une valeur.

1.3 Objectif

L'objectif de ce workshop est de créer en C# un petit logiciel qui puisse à la fois faire divers traitements sur une expression comme la dérivée, ou la simplification et qui soit également capable de faire l'évaluation de cette expression à condition qu'on spécifie les valeurs des variables se trouvant dans l'expression.

2 Créer l'interface utilisateur

2.1 Création d'une application Windows Forms

Lorsque vous créez un nouveau projet dans *Visual Studio* vous vous trouvez face à une multitude de choix. Dans notre cas nous allons utiliser une *Application Windows Forms* que nous nommerons *CAS*. Une fois l'application créée une fenêtre nommée `form1` apparaît à l'écran. À l'aide de

la boite à outils (généralement située sur le côté de l'écran) nous allons insérer les composants suivants :

- un champ de texte (`TextBox`) qui nous servira à rentrer les lignes de calculs à traiter,
- un bouton qui déclenchera l'évaluation de l'expression,
- un bouton qui simplifiera l'expression,
- un bouton qui dérivera l'expression,
- une liste de choix (`ListBox`) qui contiendra l'historique de toutes les opérations effectuées,
- un bouton qui permettra d'effacer l'historique.

Une fois tous ces composants ajoutés, notre interface devrait ressembler à ceci :

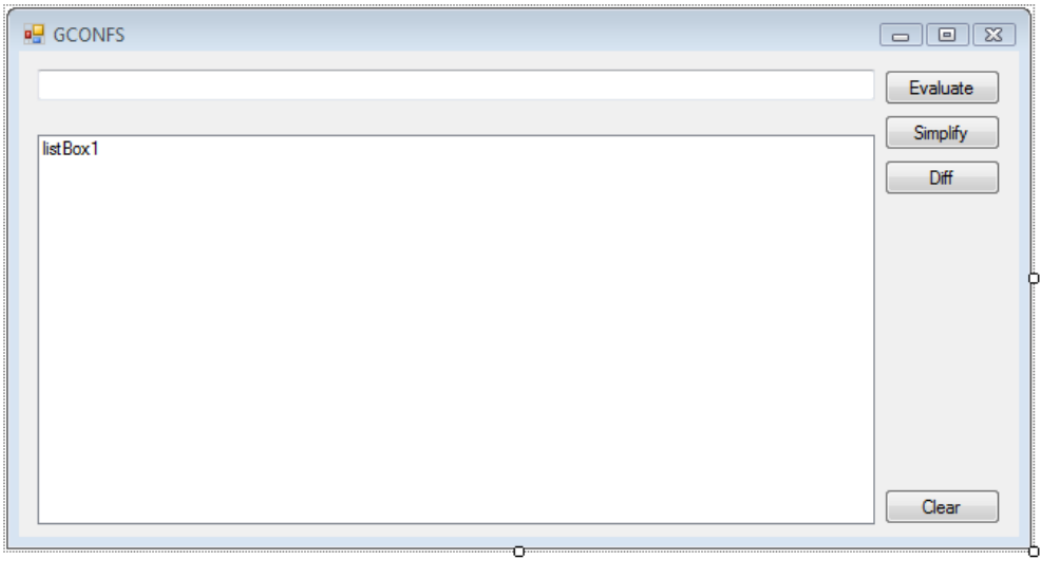

FIGURE 1 – Exemple d'interface

La disposition est bien entendu laissée à votre convenance : soyez créatif !

2.2 La fonction d'effacement

La première fonction que nous allons coder est la fonction qui permet d'effacer l'historique, c'est-à-dire celle qui permet de vider la liste de choix, lorsque l'on appuie sur *Clear*.

Aide : pour créer une fonction qui est appelée au moment où on clique sur un bouton, double-cliquez simplement sur le bouton en question.

3 Les valeurs

3.1 La création de la classe `Value`

Nous allons commencer par créer une classe `Value`. Cette classe représentera une valeur constante de la forme :

$$a + b * \pi + c * e$$

Classe dans laquelle nous ajouterons donc trois valeurs flottantes a, b, c. On demande ici uniquement la création de la classe : les méthodes qui vont modifier les valeurs de ces variables seront écrites plus tard.

Note : cette représentation va nous permettre par exemple d'effectuer des opérations telles que $2\pi + \pi$ sans perte de précision.

3.2 Les opérations

C# nous donne la possibilité de définir nos propres opérateurs ($+$, $-$, $*$, $/$, ...) sur des types que nous avons nous-même créés. C'est ce que nous allons faire avec `Value`. En effet, il serait agréable de pouvoir utiliser directement les opérateurs $+ - */$ sur nos valeurs plutôt que de faire appel à des fonction comme `Add`, `Sub`, ... qui alourdiraient vite le code.

La syntaxe est la suivante :

```
public static Value operator +(Value A, Value B)
{
  Value C = new Value();
  // Fix Me;
  return C;
}
```

Note : un opérateur doit obligatoirement être déclaré `public static`.

Nous allons donc coder les opérateurs $+$, $-$, $*$ et $/$ en considérant que si l'un des deux membres de l'opération est un flottant le résultat sera sous la forme d'un flottant.

3.3 Les conversions implicites

Parfois il serait plus agréable de pouvoir utiliser un `int` ou un `float` en tant que `Value` et inversement. Pour cela C# dispose des opérateurs de conversions implicites : c'est-à-dire que c'est le compilateur qui décidera quand est-ce-qu'il est nécessaire d'utiliser un `int` ou un `float` ou bien une `Value`.

La syntaxe est la suivante :

```
public static implicit operator Value(int A)
{
  Value C = new Value();
  // Fix Me;
  return C;
}
```

Note : l'extrait de code ci-dessus montre comment écrire une conversion implicite d'un `int` vers une `Value`.

Nous allons donc écrire les conversions suivantes :

- `int` → `Value`
- `float` → `Value`
- `Value` → `float`

3.4 Chaînes de caractères et `Value`

Une `Value` vers une chaîne

On se propose d'écrire la fonction `ToString()` qui retourne la chaîne de caractères correspondante à la valeur. Voici le prototype de la fonction :

```
public override string ToString()
{
  // Fix Me;
}
```

Note : le mot-clé `override` vient du fait qu'il existe une méthode `ToString` par défaut dans chaque classe.

Une chaîne vers une `Value`

On se propose d'écrire la fonction `FromString()` qui retourne la valeur correspondante à une chaîne de caractères.

Afin de simplifier la fonction et parce que les autres cas sont inutiles, nous ne considèrerons que les chaînes de caractères de la forme suivante :

- "10", "10.0" , "11.97", "−11.97", ...
- "10 π", "10.0 π", "11.97 π", "−11.97 π", ...
- "10 e", "10.0 e", "11.97 e", "−11.97 e", ...

On considérera également qu'il y a toujours une espace entre le nombre et π ou e.

Les opérations suivantes sur les chaînes de caractères pourront être utilisées :

```
string number = "10.0";
number = number.Replace(".", ",");
  // number contient maintenant "10,0"
float a = Convert.ToSingle("10,65");
  // le nombre doit contenir une , et non un . pour que ToSingle
       fonctionne
```

Voici le prototype de la fonction :

```
public static Value FromString(string str)
{
  // Fix Me;
}
```

> **Note :** La méthode est statique car elle retourne une nouvelle `Value`. Elle n'a donc pas besoin de dépendre d'une `Value` déjà existante (d'une *instance* de `Value`). On aurait également pu écrire cela comme un constructeur.

3.5 Est-ce que c'est zéro ?

Il faut maintenant créer la méthode `IsNull` qui retourne vraie seulement si la `Value` vaut 0 c'est-à-dire si $a = b = c = 0$ (on laissera de côté les cas pathologiques de la forme $a = 0$ $b = -e$ et $c = \pi$).

Voici le prototype de la fonction :

```
public static bool IsNull()
{
  // Fix Me;
}
```

4 Les expressions

4.1 La classe abstraite `Exp`

Nous allons créer la classe abstraite `Exp` qui va définir les méthodes qui devront être présentes dans chacune des classes qui vont hériter de `Exp`. Cette classe abstraite va se contenter de définir les prototypes des méthodes et non les implémenter. Afin d'éviter tout problème lors de la réalisation de cette classe le code basique de la classe est disponible ci-dessous :

```
abstract class Exp
{
  public abstract Exp Simplify();
  public abstract Exp Diff(string Var);
  public abstract Value Evaluate(Dictionary<string, Value>
     Variables);
  public abstract Value ToValue();
  public abstract bool IsNull();
  public abstract bool IsConst();
  public virtual new string ToString(){return "";}
}
```

Par la suite nous allons modifier cette classe mais pour le moment, nous pouvons continuer en la laissant en l'état.

4.2 L'expression constante

Commençons par coder une expression assez particulière, l'expression constante.

Pour déclarer une classe qui hérite d'une autre classe la syntaxe est la suivante :

```
class Cst : Exp
{
}
```

Toute classe qui hérite d'une classe abstraite doit impérativement redéfinir les méthodes marquées comme `abstract`, déclarées dans la classe abstraite dont il hérite. Dans notre cas `Cst` va devoir redéfinir `Simplify`, `Diff`, `Evaluate`, `ToValue`, `IsNull`, `IsConst`, `ToString` (même si cette méthode est marquée comme `virtual` et non comme `abstract`, nous allons quand même la redéfinir). Pour redéfinir une méthode dans une classe il faut utiliser la syntaxe ci-dessous :

```
public override bool IsConst()
{
  // Fix Me;
}
```

Maintenant que nous savons comment faire nous allons le faire. L'expression constante est une expression qui est construite avec une valeur. Il est donc nécessaire d'écrire un constructeur dont le prototype est le suivant :

```
public Cst(Value value)
{
  // Fix Me;
}
```

Note : vous devez définir la/les différentes variable(s) nécessaire(s) vous-même pour que ce constructeur puisse fonctionner.

La particularité de la classe constante est que quelque soit l'ensemble de variables passé en paramètre, `Evaluate` renverra toujours la valeur avec laquelle la classe a été construite. La dérivé renverra toujours la constante construite avec la valeur nulle (`new Cst(0)`). `IsNull` testera si la valeur est nulle, `IsConst` renverra toujours vrai, `ToValue` retournera la valeur avec laquelle la classe a été initialisée et `ToString` retournera la chaîne de caractères correspondant à la valeur avec laquelle la chaîne a été initialisée.

Implémentons donc ces méthodes.

`Symplify` est une commande qui renvoie l'expression simplifiée mais comme dans le cas d'une constante il n'y a rien à simplifier, nous renverrons simplement une nouvelle constante initialisée avec la même valeur que la première.

4.3 L'expression variable

La structure de l'expression variable reprend celle de la constante :

```
class Var : Exp
{
}
```

Nous commencerons donc par écrire le constructeur de l'expression variable qui prend en paramètre une `string` qui représentera le nom de la variable :

```
public Var(string value)
{
  // Fix Me;
}
```

Note : vous devez définir la/les différentes variable(s) nécessaire(s) vous même pour que ce constructeur puisse fonctionner.

Il devrait être assez facile maintenant d'implémenter les méthodes `Diff`, `ToValue`, `IsNull`, `IsConst`, `ToString`. On considérera que `IsNull()` ne peut jamais être vraie et que `ToValue` renvoie `null` (`ToValue` n'existe que s'il y a possibilité de transformer l'expression en valeur or ce n'est pas le cas ici).

Intéressons nous maintenant à `Evaluate`. Pour accéder à un dictionnaire, la syntaxe est la même que pour accéder à un tableau. En effet, pour récupérer la valeur de la variable `toto` dans le dictionnaire de variables, en supposant qu'elle existe, il suffit de faire `Variables[toto]` et nous obtiendrons sa valeur. En sachant cela nous pouvons donc coder la fonction `Evaluate` pour les expressions variables.

Une fois de plus pour `Symplify` il n'y a rien à simplifier, nous renverrons simplement une nouvelle expression variable initialisée avec la même valeur que la première.

4.4 Les autres expressions

Après avoir codé les deux expressions particulières nous allons maintenant pouvoir travailler sur des expressions plus conventionnelles. Nous

allons commencer par coder l'addition (`Add`), la soustraction (`Sub`), la division (`Div`) et la multiplication (`Mul`). Vous pouvez bien sûr rajouter par la suite autant d'expression qu'il vous plaira : cos et sin par exemple.

Toutes ces expressions ont un constructeur similaire qui prend en paramètre deux expressions (`A` et `B`) qui sont respectivement la partie gauche et la partie droite de l'opération :

```
public Add(Exp A, Exp B)
{
  // Fix Me;
}
```

Ainsi pour déclarer $Toto = 5 + 2$ nous déclarerons : `toto = new Add(new Cst(5), new Cst(2))`.

Nous pouvons à présent coder toutes les opérations nécessaires pour que ces expressions fonctionnent correctement.

> **Note :** Soignez particulièrement votre fonction de simplification pour chaque expression ; c'est elle qui fera la force de votre CAS.

5 Créer une expression à partir d'une `string`

5.1 Lecture d'une chaîne en NPI (Notation Polonaise Inverse)

Vous connaissez sans doute le principe de la notation polonaise inverse. Pour rappel, $5 + 8 * 6$ s'écrit en NPI $8\,6\,*\,5\,+$.

À vous de trouver un algorithme (en utilisant une pile de `string` : `Stack<string>` en C#) qui permet de transformer une chaîne en NPI vers une expression.

> **Note :** On pourra utiliser un `switch ... case`

5.2 Lecture d'une chaîne en notation infixe

La notation infixe est la notation que nous utilisons tous les jours. Par exemple :

$$(5 + x) * 10 * (\pi + 10)$$

La prise en charge de ce genre de notation est une étape plutôt délicate. C'est pourquoi nous allons procéder en deux temps. Dans un premier temps nous allons convertir la chaîne en notation infixe vers une chaîne en NPI puis nous utiliserons la fonction précédemment écrite pour en extraire une expression.

Un des algorithmes qui permet de passer d'une notation infixe à une notation en NPI s'appelle Shunting-yard algorithm (littéralement algorithme de la voie de garage). Le nom de cet algorithme vient de la façon dont on procède pour créer la chaîne en NPI.

Voici une version simplifiée de son fonctionnement :
On utilisera une pile de `string` (`Stack<string>` en C#) pour la voie de garage et une `string` pour la sortie.

- L'élément est (: on empile (
- L'élément est) : on dépile et on met les éléments dépilés dans la sortie jusqu'à trouver l'élément (. L'élément (ne sera pas ajouté à la sortie.
- L'élément est **un nombre ou une variable** : on le met directement dans la sortie
- L'élément est **un opérateur *opp1*** :
  1. Tant qu'il y a un opérateur *opp2* sur le sommet de la pile et que la priorité de *opp2* est supérieure ou égale à la priorité de *opp1*, on dépile *opp2* et on le met dans la sortie.
  2. On met *opp1* au sommet de la pile.

Quand il n'y a plus d'éléments on dépile les éléments de la pile jusqu'à ce que la pile soit vide.

Attention : cet algorithme ne marche que si tous les opérateurs sont associatifs à gauche. Voir le véritable Shunting-yard algorithm pour les autres cas.

6 Finir l'interface

Maintenant que notre CAS est capable de plein de choses, il faut finir l'interface afin de permettre à l'utilisateur de rentrer une expression puis d'effectuer tout un tas d'opérations dessus. Pour cette dernière étape vous êtes totalement libre !