M1 Informatique C++
Rapport de Mini-Projet phase 2

François Gaits

20 novembre 2019

1 Organisation de la gestion des fonctions

1.1 Traitement du string d'entrée

J'ai décidé de traiter les fonctions de la même manière que je traitais les variables lors de la phase précédente : elles sont interprétées et remplacées par leurs valeurs dans la string avant que cette dernière soit évaluée par la classe expression.

Lorsqu'une string est reçue en entrée, le programme la parcoure une première fois et fait la différence entre les caractères appartenant à une expression «pure» (réels, opérateurs, parenthèses...) et les parties d'identifiants (variables ou fonctions). Dans le premier cas il copie juste le caractère dans le string de sortie, sinon, il lit l'ensemble de l'identifiant et détermine s'il sait de quoi il s'agit.

Dans le cas d'une fonctions, le système va lire les arguments fournis, les évaluer de manière récursive et exécuter la fonction avec ces arguments avant de placer le résultat dans la string à évaluer.

```
} else if (!argbuf.empty()) {
    //if not in a nested func usage ( pow(1,pow(2,3)) )
    if (parCount <= 1) {
        double res = evaluateExpression(argbuf);
        if (std::isnan(res)) {
            std::cerr << "Expression : '" << argbuf << "'
               is invalid" << std::endl;
            return std::nan("");
        }
        args.push_back(res);
        argbuf = "";
        } else {
            argbuf += *it;
        }
}
```

Figure 1 – Appel récursif de *evaluateExpression*

1.2 Évaluation de fonctions

Une fois la fonction identifiée (et associée au membre correspondant d'une enum) et ses arguments isolés, ils sont passés à la fonction *execFunction*.

```
double execFunction(Function func,
                    const std::vector<double> &args) const;
```

FIGURE 2 – Déclaration de *execFunction* dans *Program.hpp*

Cette fonction vérifie la validité des arguments(log d'une valeur négative) et leur nombre avant d'appliquer la fonction a ces arguments et renvoyer le résultat.

```
case SQRT :
    if (args.size() != 1) {
        std::cerr << "Invalid number of arguments for
           function 'sqrt', should be 1" << std::endl;
        return std::nan("");
    } else if (args[0] < 0) {
        std::cerr << "Negative argument for function 'sqrt'
            : " << args[0] << std::endl;
        return std::nan("");
    }
    return sqrt(args[0]);
```

FIGURE 3 – Exécution de *sqrt()*

Cette organisation me permet de rajouter et modifier librement des fonctions, même si elles sont non standard (hypot, polynome...)

2 Choix d'implémentations

2.1 utilisation de la classe Expr

Un point présent depuis la phase 1 mais que j'ai peu détaillé est l'usage que je fais de ma classe Expr : elle a pour rôle de renvoyer le résultat d'une expression correctement formée (voir rapport phase 1), en gérant la priorité d'opérateurs, mais pas l'identification de variables ni l'exécution de fonction. C'est pour ça qu'elle n'est utilisée qu'à la fin de la fonction *evaluateExpression*

2.2 Agrégation de fonctions utilitaires dans *Utils.hpp*

Lors de la reprise du code pour la phase 2 il était déjà visible que plusieurs classes distinctes tenaient des définitions de fonctions utilitaires pouvant être employés par d'autres classes, et ce sans réelle logique de positionnement si ce n'est le premier endroit où j'en ai eu besoin.

Le projet étant naturellement appelé à grandir, ce problème n'en serait devenu que plus grand, c'est pourquoi j'ai rajouté un namespace *Utils* contenu dans *Utils.hpp* définissant les fonctions utilitaires pouvant se retrouver dans plusieurs classes, ces fonctions sont déclarées static et inline pour la plupart, permettant au compilateur d'au mieux s'adapter à leur utilisation et éviter plusieurs déréférencements inutiles.

```
namespace Utils {
    static inline bool isOp(const std::string &s) {
        return s.length() == 1 && (s == "+" ||
                                   s == "-" ||
                                   s == "*" ||
                                   s == "/");
    }
    [...]
};
```

FIGURE 4 – Le namespace *Utils*

3 Remarques

3.1 Evolution des tests

Le script de test a évolué : appeler ./test.sh <num> exécutera le test correspondant au num :

1– tests basiques
2– utilisation de variables
3– fonctions unaires
4– fonctions a arguments multiples
5– fonction représentant la plupart des cas limites
6– fonction générant des erreurs

Appeler ./test.sh testera le programme sur l'ensemble des fichiers test disponibles .

3.2 Éventuels refactors

Les fonctions *evaluateExpression* et *execFunction* sont longues et peu lisibles, elles mériteraient un re-découpage fonctionnel, cependant chaque partie de *evaluateExpression* est susceptible de modifier l'état de l'itérateur ce qui rend ce découpage potentiellement contre-productif. De la même façon *execFunction* répond à une logique simple de *switch case* et reste donc lisible même si longue.

3.3 Utilisation de l'énumération de fonctions

J'utilise l'enum *Function* pour identifier une fonctions associée à une string (par *getFunc()*) afin d'utiliser un *switch case*, ce qui est impossible sur des string (on pourrait les hasher et les comparer à des hash constexpr cela dit) mais ça ne fait que déplacer le problème dans la fonction *getFunc()*. Néanmoins l'énumération me permet d'utiliser la valeur UNDEFINED comme valeur «critique» dans ma gestion d'erreur (comme décrit dans le rapport phase 1) ou pour vérifier la validité d'un nom de variable.