M1 Informatique C++
Rapport de Mini-Projet phase 1

François Gaits

31 octobre 2019

1 Architecture et organisation du programme

Basé sur les consignes du TP, le programme s'articule autour de deux grands axes, l'évaluation d'expressions purement arithmétiques (sans littéraux) et la gestion de variables.

1.1 Évaluation d'expressions

L'évaluation d'expressions se fait au travers de la classe *Expr*, instanciée avec une chaine de caractère représentant une expression bien formée [1] contenant uniquement des nombres réels et des opérateurs.

Une instance de cette classe fournit deux méthodes :
- *eval* : renvoie la valeur de l'expression sous forme de **double**.
- *print* : affiche la notation polonaise inversée (RPN) correspondant à l'expression sur la sortie standard.

La première étape est la décomposition de la chaine de caractère en chaines plus petites comportant des éléments atomiques de l'expression :
- des nombres réels (e.g. 10, 10.6, 5., .5 ...)
- des opérateurs ('+', '-', '*', '/')
- des parenthèses ('(', ')')
- et l'opérateur '-' monadique

Étant donné cette décomposition la méthode *toRPN* transforme ces chaines en un vecteur de pointeurs uniques sur des instances de *ExprToken* organisé en respectant la priorité des opérations sous forme de RPN.

La fonction *eval* déroule simplement l'évaluation de la RPN pour donner un résultat.

Et la fonction *print* affiche le contenu du vecteur de RPN.

1. notion précisée dans la deuxième partie

1.2 Gestion de variables

La gestion des variables est réalisée en amont des expression dans la classe *Program* instancié avec un flux d'entrée, par défaut 'std ::cin'.

Trois types d'instructions sont possibles :
- une variable existante dont la valeur sera affichée sur la sortie standard
- une expression dont le résultat sera affiché sur la sortie standard
- une affectation de la forme ***var = expr ;***

Pour l'affichage d'une variable le programme vérifie simplement que la variable existe et affiche son contenu.

L'évaluation d'une expression se passe en deux étapes : premièrement le programme cherche la présences de variables dans l'expression et les remplace par leur valeurs stockées avant d'utiliser.

L'affectation isole la partie variable et la partie expression, évalue d'un côté l'expression et de l'autre traite le nom de la variable (suppression des espaces, vérification de la validité des caractères) pour ensuite affecter la variable à la valeur de l'évaluation (avec écrasement de l'enregistrement précédent s'il y a lieu).

2 Choix d'implémentations

Cette partie présente certaines spécificité du programme et interprétations personnelles du sujet.

2.1 Priorité des opérateurs

Pour la question de la priorité des opérateurs, la consigne du sujet (/ et * > - et +, de droite à gauche en cas d'égalité) est respectée bien qu'elle ne soit pas celle qui est utilisée couramment.

2.2 Forme des expressions

Une expression sera considérée bien formée si elle est correctement parenthésée, si elle ne contient pas de variables inconnues, si elle commence et finit par un nombre (ou commence par un '-'), et ne contient pas plusieurs opérateurs binaires à la suite.

Le cas où un opérateur est absent (entre des nombres ou des parenthèses (e.g. 10(5+3), 5 3 + 1)) il sera traité comme un '*' a priorité minimale (10(5+3) = 80), cette particularité est très peu intuitive, il ne s'agit pas d'un réel opérateur '*' et change l'ordre d'évaluation sur les expressions complexes, et ne devrait pas être utilisée. (5+3 4+5 $\neq$ (5+3)*(4+5))

La présence d'espaces en dehors de ce cas est omise.

3 Détails du code

Dans cette section certains passage du code sont détaillés.

3.1 Les tokens

La classe *ExprToken* représente les éléments atomiques d'une expression au travers de deux classes filles *ExprTokenNumber* et *ExprTokenOp*

```
class ExprToken {
private:
    TokenType _type;
public:
    explicit ExprToken(TokenType type = NONE);
    inline const TokenType type() const { return _type; }
    bool operator<(const ExprToken &t) { return false; }
    virtual const double number() = 0;
    virtual const char op() = 0;
};
```

Les accesseurs *number* et *op* sont définis en virtuels purs pour êtres redéfinis et utilisés par les instances des classes filles.

Cette particularité sert à tirer parti du polymorphisme dans les collections contenant des tokens comme par exemple :

```
using vecToken = std::vector<std::unique_ptr<ExprToken>>;
```

L'utilisation de pointeurs permet de passer deux problèmes : les slice des instances lors du passage par copie dans une collection, et l'impossibilité d'utiliser des références due à l'impossibilité d'assurer la durée de vie des instances référencées dans la collection.

Les pointeurs étant uniques il faut les *move* en mémoire et pas les copier :

```
std::unique_ptr<ExprToken> a = std::move(stack.top());
```

Cela permet d'accéder au type des tokens et d'accéder à leurs champs en conséquences alors qu'ils sont stockés dans une collection générique.

```
if (t->type() == OP) {
    std::cout << t->op();
} else if (t->type() == NUMBER) {
    std::cout << t->number();
}
```

3.2 Traitement des '-' unaires

Lors de la sépartation de la chaine de caractère en éléments atomiques, les '-' rencontrées hors de l'emplacement attendu d'un opérateurs sont considérés unaires (ou monadiques) et sont traités d'une façon particulière afin de garantir la bonne exécution de la suite du programme : ils sont simplement remplacés par '(-1)' et '*' dans la liste des éléments.

```
if (*it == '-') {
    vector.emplace_back("-1");
    vector.emplace_back("*");
    ++it;
    hasAdvanced = true;
}
```

3.3 Gestion des erreurs

Lorsqu'une méthode rencontre un cas non spécifié, elle affiche sur la sortie standard des erreurs un message adéquat et retourne un élément caractéristique (NaN pour l'évaluation, string vide pour le split, vecteur vide pour la RPN, etc...). C'est ensuite la classe *Program* qui utilise leurs services et qui à son tour vérifie leur valeur retour, pour continuer ou non l'instruction en cours avant de lire la ligne suivante.

```
if (!std::isnan(result) && !name.empty()) {
    variableMap[name] = result;
    std::cout << "result (" << result << ") saved under the
        name '" << name << "'" << std::endl;
} else {
    return false;
}
```

Cela permet de ne pas avoir recours au mécanisme lourd de *throw / catch* de c++, et de pas interrompre l'exécution du programme à la première faute de syntaxe.

4 Remarques

L'exécution prend fin lorsqu'elle rencontre EOF, "quit", "q" ou "exit" dans son flux d'entrée.

Le script *build.sh* permet de build l'exécutable avec cmake.

Certaines parties du code sont symptomatiques de tests au cours du développement et sont inutilement compliqués (inutile de stocker des tokens dans la stack de la RPN, elle ne contient que des nombres, par exemple)