Mini-Projet C++
Rapport de Phase 1

Organisation du projet

Racine
Compilation
↳ *CMakeLists.txt*
Classe Expr : représente une expression
↳ *Expr.h*
↳ *Expr.cpp*
Classe ExprToken : représente un token
↳ *ExprToken.h*
↳ *ExprToken.cpp*
Programme principal
↳ *main.cpp*
Classe Program : représente une séquence d'instructions
↳ *Program.h*
↳ *Program.cpp*
Classe StringOperations : classe annexe pour des opérations statiques sur des chaînes de caractères
↳ *StringOperations.h*
↳ *StringOperations.cpp*

Description du travail effectué

Les questions 1 et 2 du TP ont été effectuées.
Cependant il semble que ma version de la conversion d'une syntaxe infixe vers RPN contient encore des erreurs (bug découvert en fin de projet : lorsque le programme détermine l'ordre des opérations entre des opérateurs de même priorité, il mélange les opérandes ne donnant pas un bon résultat).
Le reste semble fonctionner correctement. L'application ne gère donc ni les parenthèses, ni les expressions sans espaces entre les tokens.

Gestion des nombres réels

Le programme a bien été modifié pour respecter cette contrainte. Le passage du type **int** en **float** a nécessité peu de changements :

- Dans la classe représentant un token :

- Lors de sa conversion (si c'est un littéral), la fonction *std::stoi* est remplacée par *std::stof*.
- L'attribut qui permet de conserver la valeur a changé de type.

Capacité à évaluer une séquence d'expressions

Classe Program

Cette classe peut être instanciée avec le constructeur de base (programme vide) ou bien avec une chaîne de caractères contenant un programme (peut être directement exécuté via la méthode *exec()*) et une instance peut être "remplie" via l'opérateur d'entrée >>. Pour quitter le mode interactif depuis le terminal, tapez '*exit;*'. Cela affichera toutes les représentations internes des expressions tapées.

Un exemple d'interaction avec l'application

```
4 * 2 + 3 * 6 / 13;
8 * 9 - 1
71
3 + 1
4
```

Pipeline des opérations

Une fois une séquence d'expressions passées à un objet Program, il découpe les expressions et les évalue une par une via la classe Expr écrite en TP. Puis il affiche le dernier résultat (sans point virgule). Cette partie fonctionne correctement.

Gestion d'une mémoire

Mémoire des variables

Un attribut de type *std::map* a été ajouté à la classe Program. Cette structure associative permet de stocker des couples de données. Pour chaque nom de variables affectées, un objet de type Expr contenant l'expression qu'on lui a affectée est enregistré dans cette structure. Cela permet de gérer les dépendances à l'exécution du programme (une entrée n'existant pas dans la map implique que la variable n'a jamais été attribuée)

Ajout de deux types de Token

Il a fallu gérer deux types de tokens supplémentaires :

- Les noms de variables (symboles)
- Le signe égal (pour affectation)

Pour cela les classes Expr et ExprToken ont été modifiées afin d'accepter ces nouveaux types. De plus, la méthode *Expr::eval()* reçoit désormais en paramètre une référence à la

map des symboles du programme dans lequel elle opère. Si une expression de type affectation est évaluée alors une entrée est ajoutée dans la map des symboles. A l'inverse, si on rencontre un symbole dans une expression, alors on va chercher son expression correspondant dans la table pour l'évaluer et l'inclure dans le calcul.
Remarque : Le choix de cette implémentation implique que le contexte est global. Une affectation contenant une variable dont la valeur est changée plus tard change donc aussi de valeur.

Un exemple d'interaction avec l'application

```
deuxpi = 2 * 3.1415926536;
rayon = 3 * 8;
circonference = deuxpi * rayon;
circonference
150.796
```

Cette partie fonctionne correctement.

Mini-Projet C++

Rapport de Phase 2

Organisation du projet

Racine
Compilation
↳ *CMakeLists.txt*
Classe Expr : représente une expression (lexeur)
↳ *Expr.h*
↳ *Expr.cpp*
Classe ExprToken : représente un token
↳ *ExprToken.h*
↳ *ExprToken.cpp*
Programme principal
↳ *main.cpp*
Classe Program : représente une séquence d'instructions
↳ *Program.h*
↳ *Program.cpp*
Classe StringOperations : classe annexe pour des opérations statiques sur des chaînes de caractères (parseur)
↳ *StringOperations.h*
↳ *StringOperations.cpp*
Classe Func : représente une fonction
↳ + *Func.h*
↳ + *Func.cpp*

Description du travail effectué

La majeure partie du travail effectué se trouve dans la remise à niveau de la calculette concernant la gestion des parenthèses, la non séparation (par des espaces) des tokens. Le parseur et la conversion en RPN ont été réécrites et fonctionnent bien mieux, tout en gérant les cas précédemment énoncés.

Gestion des fonctions

Cette partie du sujet est en partie traitée. Dû au retard de la première phase, aucune version n'a à ce jour abouti à un programme complet et fonctionnel. Cependant le développement est toujours en cours malgré l'impossibilité de démontrer le travail effectué jusqu'ici.

Classe Func

Cette classe a été introduite dans le but de gérer des fonctions, qui sont dans ce programme considérées comme la spécialisation d'une expression. Elle permet de contenir un nom de fonction, un ensemble d'arguments, et une fonction surchargée d'évaluation de cette expression. Elle sera donc parfaitement compatible avec la définition du programme, écrit pendant la phase numéro 1, pour l'évaluation.

Ajout de deux types de Token

Il a été nécessaire au bon fonctionnement de la calculette d'ajouter trois différents tokens non pris en charge lors de la phase 1 de ce projet:

- Les parenthèses
- Les séparateurs d'arguments

Permettant l'identification par le lexeur des fonctions de la forme:

<nom>(<argument_1>, ..., <argument_n>)

Gestion des fonctions à plusieurs arguments

Classe Func

La classe Func telle qu'elle est écrite au jour de l'écriture de ce rapport est prévue pour supporter les fonctions à plusieurs arguments car elle contient une liste d'arguments (expressions de type Expr) permettant une évaluation des expressions constituant les arguments avant l'évaluation globale de la fonction.
Elle permet aussi de supporter la déclaration de fonction, bien que cette étape ne soit pas encore demandée (dans le cas où la phase 3 nécessite cette fonctionnalité, l'initiative n'aura pas été vaine).

Les fonctionnalités demandées n'ont donc pas à ce jour été réalisées dans leur totalité. J'en assume le retard, cependant il me semble préférable de réaliser un produit bien pensé et facile à faire évoluer, mais d'abord fonctionnel. Je ferai de mon mieux pour finir en totalité la phase numéro 3.