

Travaux Dirigés sur Machine n°7 — Expressions arithmétiques postfixées en C++ — 4h noté

L'objectif est de calculer les expressions arithmétiques basées sur des opérateurs unaires et binaires contenant des variables ou des constantes.

Ces expressions seront exprimées en notation postfixe. Le principe de l'écriture postfixe est de mettre l'opérateur après les opérandes, par exemple `.7 exp` pour `exp(.7)` `4 3 /` pour `4/3` et `x 3 :=` pour `x := 3`.

La formule `2 x + 3 * log` correspond à l'expression `log((2+x)*3)` et pourra être évaluée dès que la variable `x` aura une valeur.

Ce TD est divisé en deux parties. La première partie consiste à construire des expressions arithmétiques. La deuxième partie est l'implantation d'une classe permettant de construire une expression arithmétique à partir d'une notation postfixe et de l'évaluer pour différentes valeurs des variables qu'elle contient.

Exercice 1. Expressions arithmétiques

On souhaite manipuler des expressions arithmétiques sur des `double` et les évaluer. Elles s'appuient sur les classes suivantes.

1. La classe `Expr` est une classe abstraite avec deux méthodes virtuelles pures `double eval()` et `string toString()` que devront définir les classes dérivées. Le champ `priority` sert à ajouter correctement les parenthèses pour écrire l'expression en format infixe sans parenthèses superflues. Par exemple, pour `x + 1 y *` on doit obtenir `(x+1)*y` et non `((x+1)*y)` ni `x+1*y`.
2. La classe `Constant` sert à représenter les constantes doubles (18.6 par exemple).
3. La classe `Variable` sert à représenter l'utilisation d'une variable dans une expression. L'expression ne leur donne aucune valeur ; celle-ci est fournie par un environnement d'évaluation. Celui-ci est un `Evaluation_Context` (ou une sous-classe de `Evaluation_Context` selon les besoins). Il faut compléter le module `evaluation_context` pour pouvoir utiliser des variables.
4. La classe `Op_Unary` sert à factoriser le code pour les fonctions `log (Log)` et `exp (Exp)`.
- [1] Compléter le code, la documentation (de `Op_Unary`) et expliquer cette construction (dans le CR).
5. La classe `Op_Binary` sert à factoriser le code pour les fonctions `+` (`Add`) , `-` (`Sub`) , `*` (`Mul`) et `/` (`Div`).
- [2] Compléter le code, la documentation (de `Op_Binary`) et expliquer cette construction (dans le CR).
6. La classe `Set` sert à définir l'affectation de variable. Elle correspond à l'expression `x 3 :=` en postfixe et `x :=3` en infixe. Son évaluation, en plus de l'affectation, renvoie la valeur qui vient d'être donnée à la variable.

Pour illustrer le lien entre `Variable`, `Evaluation_Context` et `Set`, dans le code suivant, l'expression `(y :=x+3)+y` (`exp`) est créée puis évaluée deux fois : avec `x` valant 7 puis 11.

```
Expr * add = new Add ( new Variable ( "x" ) , new Constant ( 3 ) ) ;
Expr * set = new Set ( new Variable ( "y" ) , add ) ;
Expr * exp = new Add ( set , new Variable ( "y" ) ) ;
Evaluation_Context_Simple ec ;
ec . valuate ( "x" , 7 ) ;
cout << exp -> toString () << " ==> " << exp -> eval ( ec ) << endl ;
ec . valuate ( "x" , 11 ) ;
cout << exp -> toString () << " ==> " << exp -> eval ( ec ) << endl ;
delete ( exp ) ;
```

La sortie est :

```
( y := x + 3 ) + y ==> 20
( y := x + 3 ) + y ==> 28
```

L'archive TD7.tgz contient différents modules pour représenter les expressions. Les fichiers `exparith*.hpp` et `exparith*.cpp` contiennent respectivement les déclarations des classes (et la documentation) et le squelette des méthodes à implémenter. Il faut terminer ces modules.

`make T1` vérifie le résultat de l'exemple ci-dessus.

Le programme de test `test_exparith.cpp` permet de tester l'implantation. Le fichier `test_exparith_out_expected.txt` donne les résultats à trouver pour le calcul et l'affichage. `make T2` vérifie le résultat.

- 3 Dessiner le diagramme UML d'héritage entre toutes les classes.
- 4 La construction se fait en utilisant des pointeurs pour les sous-expressions. Pourquoi ne pas avoir utiliser de références ?
- 5 Écrire l'expression correspondant au dernier test de `test_exparith.cpp` avec des variables en *format postfixe*.

Exercice 2. Notation postfixe et évaluation interactive

L'idée pour cette partie est d'une part de lire des expressions en notation postfixe et d'autre part d'évaluer en demandant les valeurs des variables non évaluées.

L'expression se termine par "." et les éléments doivent être séparés par des espaces. Par exemple à partir de l'entrée `1 x + 2 y + *`, l'expression à évaluer est $(1+x)*(2+y)$.

Tout cela est géré par le module `loader_evaluator` et la classe `Loader_Evaluator`. Le constructeur prend en argument un flux sur lequel est lue l'expression.

Une fois une instance obtenue, il est possible d'évaluer l'expression enregistrée à partir d'un `Evaluation_Context` ou à partir d'un flux d'entrée. Dans le cas où l'on donne un flux, on utilise une sous-classe de `Evaluation_Context` (définie dans le module) qui lit les valeurs sur le flux pour les variables non évaluées.

Le programme de test `test_evaluator.cpp` permet de tester l'implantation.