LSINF 1103 – Introduction à l'algorithmique

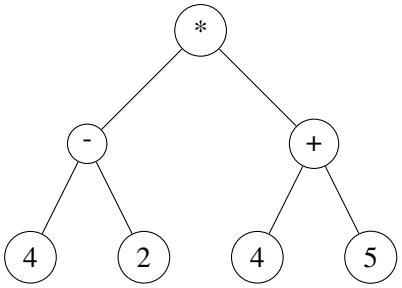
Mini-projet 1 - Manipulation d'expressions arithmétiques

Le but de ce projet est d'implémenter une calculette capable de manipuler une expression arithmétique et de l'évaluer de manière progressive. Le programme que vous allez construire devra être capable d'évaluer une expression arithmétique par simplification itérative jusqu'à obtenir sa valeur finale.

Exemple : `((4-2)*(4+5)) = (2*(4+5)) = (2*9) = 18`.

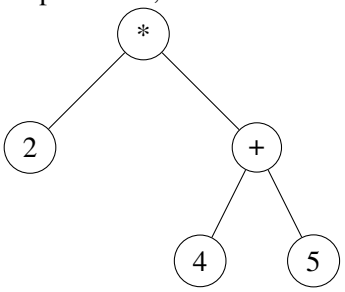
1 Comment faire ?

La tactique que nous allons utiliser est la suivante. Pour une expression donnée, il faut d'abord construire un *arbre syntaxique abstrait (AST)* qui la représente. Pour simplifier un peu la structure de données, on ne considère que des **opérations binaires**, c'est-à-dire des opérateurs avec deux opérandes. Les nœuds de l'AST seront donc composés d'une opération qui s'applique aux sous-arbres de gauche et de droite. Les feuilles contiendront les nombres. Par exemple, l'expression `((4-2)*(4+5))` sera représentée par l'arbre suivant :



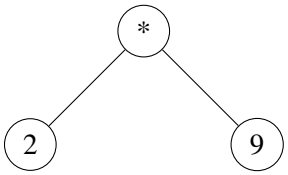
Notons que l'appel à une méthode `toString` sur cet arbre devrait renvoyer un `String` correspondant exactement à l'expression d'origine `((4-2)*(4+5))`.

Ensuite, pour la transformation de l'expression, il faudra faire un parcours postfixe en créant un nouvel arbre intermédiaire à chaque étape. Rappelons que dans un parcours postfixe, les fils du noeud courant sont traités avant le noeud lui-même. Le premier noeud interne traité lors d'un tel parcours contient un opérateur et l'arbre sera simplifié par évaluation de cet opérateur. A partir de l'arbre précédent, on obtiendrait donc :



= `(2*(4+5))`

Puis :



= `(2*9)`

Et enfin :

$= 18$

2 Concrètement

On vous laisse toute liberté pour l'implémentation, veuillez cependant respecter les éléments suivants :

- La/les classes qui représentent votre AST doivent implémenter l'interface `ExprIF` (cf. listing 1).
- La/les classes liées à la structure de l'arbre doivent se trouver dans le package `exprTree`.
- La classe qui crée l'AST à partir d'une expression s'appelle `TreeBuilder`. Elle implémente l'interface `TreeBuilderIF` (cf. listing 2) et se trouve dans le package `builder`.
- Le constructeur de cette classe prend comme seul argument un `String` qui représente l'expression arithmétique.
- Les opérateurs à considérer sont +, −, ⋆ et /.
- La méthode `getReducedTree`, qui simplifie l'arbre courant (en renvoyant un arbre simplifié), ou sa/ses méthode(s) auxiliaire(s) pour réaliser cette opération, ainsi que la méthode `toString`, qui permet de convertir un arbre en une expression complètement parenthésée sous forme de `String`, sont à appeler de manière **récursive** (utiliser un parcours adéquat).

Veillez aussi à ce que votre code soit clair et facilement maintenable. Une archive avec la structure de base et les interfaces précitées se trouve sur iCampus.

2.1 Construction de l'arbre

Pour simplifier l'algorithme de construction de l'arbre, les expressions arithmétiques en entrée sont données sous forme infixe (notation standard) *complètement parenthésée*, c'est-à-dire que, pour chaque opérateur, il y a une paire de parenthèses qui le groupe avec ces (deux) opérandes. Par exemple, l'expression `(5*(4/2))` est une entrée correcte alors que `5*(4/2)` ne l'est pas.

Étant donnée une expression en notation infixe complètement parenthésée, l'algorithme 1 permet de construire l'AST correspondant [GT10].

2.2 Entrées et sorties de votre programme

Votre programme sera exécuté en console de la façon suivante :

```
java EvalExpr exprs.in eval.out
```

où le premier argument est le chemin vers le fichier à lire en entrée, ici `exprs.in`. Ce dernier contient une expression complètement parenthésée par ligne. Le fichier `eval.out` est crée par le programme. Après l'exécution, il contient respectivement sur chaque ligne l'évaluation finale de l'expression correspondante ainsi que toutes les tranformations progressives de l'arbre. Ainsi, si une ligne dans `exprs.in` contient `((4-2)*(4+5))`, la ligne correspondante dans `eval.out` serait `((4-2)*(4+5)) = (2*(4+5)) = (2*9) = 18`.

Il faut donc que vous écriviez une classe `EvalExpr.java` avec une méthode `main` qui exécute le programme en tenant compte de ces arguments.

Pour lire et écrire les fichiers, les classes `java.io.BufferedReader` et `java.io.BufferedWriter` vous seront utiles. Pour parcourir les expressions données sous forme de

`String` en entrée, il est conseillé d'utiliser la classe `java.util.Scanner` dont la documentation se trouve ici : `http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html`.

Voici les spécifications du format des fichiers d'entrée et sortie :

— il y a une expression par ligne ;
— les expressions sont complètement parenthésées ;
— dans le fichier d'entrée, chaque élément de l'expression est séparé par un espace. Par exemple : `( 5 * ( 4 / 2 ) )`
— dans le fichier de sortie, il n'y a pas d'espace entre les éléments d'une expression. Il y a "=" entre 2 expressions consécutives avec un espace avant et un autre après. Par exemple : `(5.0*(4.0/2.0)) = (5.0*2.0) = 10.0`
— dans le fichier de sortie, la représentation en String des nombres est celle de `Double.toString(double d)`.

2.3 Bonus (optionel)

Prendre en compte certaines simplifications évidentes. Comme vous le savez, 0 est l'élément neutre de l'addition et 1 l'élément neutre de la multiplication. Ainsi, l'arbre `(0+X)` où `X` est un sous arbre quelconque peut directement se mettre sous la forme `X`. Il en est de même pour `(1*X)`. Par exemple : `(1*(4+5)) = (4+5)` et `(0+(8*12)) = (8*12)`.

2.4 Testez votre programme !

Il y a un exemple de fichier d'entrée sur iCampus. Cependant, nous vous conseillons fortement d'inventer vous-même plusieurs exemples. Vous pouvez vérifier vos résultats sur Inginious.

3 Modalités pratiques

Vous allez travailler **par groupe de deux étudiants**. Vous devez soumettre votre travail sur iCampus pour le **Vendredi 24 Avril 2015 à 23 :59** au plus tard sous la forme d'une archive (.zip ou .tar.gz). Cette archive doit contenir votre code et un petit rapport (en .pdf uniquement). Vous y présenterez le diagramme de classes de votre programme, vos choix d'implémentation, la complexité des différentes opérations des interfaces `TreeBuilderIF` et `ExprIF`, ainsi que les difficultés que vous avez rencontrées. Sans compter le diagramme de classes, le rapport ne doit pas dépasser une page. Pour soumettre votre travail sur iCampus, il est indispensable de vous enregistrer d'abord dans un groupe avec votre binôme. L'archive doit être déposée dans la section **Travaux/Mini-projet 1** et associé à votre groupe sur iCampus. Il est demandé aux bisseurs de travailler ensemble.

Différentes séances de consultance seront organisées pendant la durée du projet et communiquées via iCampus.

4 Annexe

Algorithm 1: buildExpressionTree(E)

Input: Une expression arithmétique E = $e_0\ e_1\ \ldots\ e_{n-1}$, où chaque e_i est un nombre, un opérateur ou un symbole de parenthèse.
précondition: E est bien formée et complètement parenthésée.
postcondition: Renvoie un arbre binaire représentant l'expression arithmétique E.

```
S ← une pile vide
for i ← 0 to n − 1 do
    if e_i est un nombre ou un opérateur then
        n ← new Node(e_i)
        S.push(n)
    else if e_i = '(' then
        Continuer à boucler
    else
        //e_i est une parenthèse fermante
        n_2 ← S.pop()
        n ← S.pop()
        n_1 ← S.pop()
        n.left ← n_1
        n.right ← n_2
        S.push(n)
return S.pop()
```

```java
package exprTree;

public interface ExprIF {

    /**
     * Simplification de l'arbre.
     *
     * @pre l'arbre représente une expression arithmétique
     *      bien construite
     * @post Si l'arbre contient au moins un opérateur,
     *       l'arbre renvoyé est obtenu après une simplification
     *       dans l'ordre d'un parcours postfixe
     *       Sinon, renvoit l'arbre original.
     */
    public ExprIF getReducedTree();

    /**
     * Conversion de l'arbre en un String.
     *
     * @pre l'arbre représente une expression arithmétique
     *      bien construite
     * @post le String renvoyé est la représentation
     *       complètement parenthésée de l'arbre
     */
    public String toString();

}
```

Listing 1 – Cette interface doit être implémentée par l'AST

```java
package builder;

import exprTree.ExprIF;

public interface TreeBuilderIF {

    /**
     * @pre L'expression arithmétique passée au constructeur est
     *      complètement parenthésée et bien formée. Les éléments
     *      de l'expression sont séparés par un espace.
     * @post renvoie un arbre représentant l'expression arithmétique
     *       passée au constructeur
     */
    public ExprIF build();

}
```

Listing 2 – Cette interface doit être implémentée par la classe TreeBuilder

Références

[GT10] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. Wiley, 5 edition, February 2010.