Algorithmique et structures de données

TD / TP 7

Les objectifs de ces séances de TD / TP sont :

- manipuler des arbres d'arité 1 ou 2
- parcourir ces arbres de diverses façons
- transformer et évaluer des expressions dans ce formalisme.

TD

1. Problématique

Dans ce TD nous allons gérer des expressions arithmétiques simples sous formes d'arbres. Les expressions considérées seront

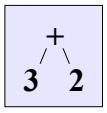
- des opérandes. Ce seront alors des nombres entiers relatifs (ex : 0, 12, -25...)
- des opérations unaires(!). leur racine sera un opérateur unaire (changement de signe ~) dont l'opérande sera une expression (ex : ~ -12, ~ (3+5)...)
- des opérations binaires(!). leur racine sera un opérateur binaire (+, -, * ou /) dont l'opérande gauche et l'opérande droit seront des expressions (ex : 3 + 5, (3+2) * (- 4)...)

De manière plus formelle, on pourrait représenter les expressions comme suit :

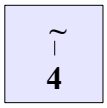
```
<Expr>   ::= <Opérande> | <OpérationUnaire> | <OpérationBinaire>| (<Expr>)

<Opérande>          ::= <entierRelatif>
<entierRelatif>     ::= ['-'] '1'..'9' ['0'..'9']*

<OpérationUnaire>   ::= <OpérateurUnaire> <Expr>
<OpérateurUnaire>   ::= '~'

<OpérationBinaire> ::= <Expr> <OpérateurBinaire> <Expr>
<OpérateurBinaire> ::= '+' | '-' | '*' | '/'
```

2. Exemples

Considérons l'expression `(3+2)`. C'est une opération binaire.
L'opérateur (binaire) `+` possède deux opérandes, l'opérande gauche `3` et l'opérande droit **2**.
On peut donc la représenter par l'arbre binaire ci-contre.



Considérons maintenant l'expression `(~ 4)`. C'est une opération unaire.
L'opérateur (unaire) - possède un opérande unique 4.
On peut donc la représenter par l'arbre unaire ci-contre.



Comment représenter graphiquement sous forme d'arbre les expressions `(3+2)*5` ; `3+2*5` ;
puis ~`(3+2)*(4+-1)` ?
Quelles valeurs numériques leur attribuez-vous ? Expliquez votre méthode d'évaluation.

Jouez au compilateur :
En utilisant l'expression `(3+2)*5` suivez la grammaire formelle et déduisez si l'expression est valide (O/N)

(!) unaire est ici pris au sens de : à un seul opérande (arité 1) ; binaire à celui de : à 2 opérandes (arité 2).

3. Affichages

Quelle que soit la notation, un Operande est représenté comme un entier signé.

a. Notation infixée

Vous utilisez habituellement, mais sans forcément le savoir, la notation dite "infixée" de représentation des expressions. Dans cette notation, les opérateurs unaires sont représentés **avant** leur opérande et les opérateurs binaires **entre** leurs 2 opérandes. Des parenthèses et/ou des règles de priorités sont souvent nécessaires pour lever les ambiguïtés d'évaluation, pour distinguer, par exemple, **`3+2*5`**, interprété comme **`(3+(2*5))`**, de **`(3+2)*5`**, interprété comme **`((3+2)*5)`**.

b. Notation préfixée

Dans cette notation, les opérateurs unaires sont représentés **avant** leur opérande et les opérateurs binaires **avant** leurs 2 opérandes. Aucune parenthèse n'est nécessaire.
Ecrivez les expressions **`(3+2)*5 et ~(3+2)*(4+-1)`** dans cette notation. Voyez-vous pourquoi les parenthèses sont inutiles ?

c. Notation postfixée

Dans cette notation, les opérateurs unaires sont représentés **après** leur opérande et les opérateurs binaires **après** leurs 2 opérandes. Ainsi, ~(3-1) sera représenté par : 3 1 - ~.
Ecrivez les expressions **`(3+2)*5 et ~(3+2)*(4+-1)`** dans cette notation.

4. Analyse

On souhaite pouvoir afficher n'importe quelle expression en notation infixée, préfixée ou postfixée. Par défaut, la méthode `toString()` de chaque classe fournira la notation infixée de l'instance.
De plus, on veut pouvoir aussi évaluer toute expression (c'est-à-dire lui attribuer une valeur).
L'encadré ci-contre indique les noms des méthodes.

```
boolean estOperande();
boolean estOperation();
boolean estOperationUnaire();
boolean estOperationBinaire();
int getValeur();
char getOperateur();
String toString();
String inFix();
String postFix();
String preFix();
```

a. Interfaces

Définissez le diagramme de classes avec `Operande`, `OperationUnaire`, `OperationBinaire` et `Expression` qui vous paraissent appropriées pour représenter ce problème.

Exemple d'utilisation souhaitée dans un main:

```
Expression exp1 =   new OperationBinaire('+',
                    new Operande(3),
                    new Operande(2) );
```

Etablissez une hiérarchie de ces classes et introduisez les "**`abstract`**" qui vous paraissent pertinents, tant au niveau des classes qu'à celui des méthodes.

Ecrivez les constructeurs pertinents.

Vous définirez une classe `TestExpressions` afin de valider les classes développées.
Votre validation devra être aussi complète et convaincante que possible.

5. Opérateurs supplémentaires

On souhaite ajouter maintenant deux nouveaux opérateurs :

- l'opérateur codé '**!**', unaire, retournant la valeur absolue de son opérande (qui est une expression)
- l'opérateur codé '**^**', binaire, retournant son opérande gauche élevé à la puissance indiquée par son opérande droit (qui doit être positif ou nul).

Modifiez les classes concernées et validez ces modifications.

6. Les variables

Introduire un nouvel élément Variable pouvant contenir le nom d'une variable associée à une expression.
Le main() se chargera de fabriquer les variables comme on l'a fait avec des opérations binaires, ...
*ex : (A*B)-5*

Remarque :
Si e=(A+3)-A ,les deux variables A ne doivent pas avoir des contenus différents.
Vous utiliserez donc un dictionnaire de type HashMap pour associer la variable à une expression unique (voir doc et exemples en ligne)
Ainsi, un premier constructeur permettra de stocker dans ce dictionnaire la nouvelle variable avec son contenu. (Si la variable existe déjà alors son contenu sera écrasé naturellement)
Un second constructeur avec seulement le nom d'une variable en paramètre pourra être utilisé dans les expressions pour rappeler la variable déjà existante.

Expression ex10 = new Variable('A', new Operande(10));
Expression ex11 = new OperationBinaire('+', ex10, new Variable('A')); // la valeur de ex11 est 20

7. Construction de l'arbre d'expression à partir d'une chaîne de caractère.

Construire automatiquement l'arbre d'expression à partir d'une chaîne de caractère décrivant une expression en pré-fixé simple. Ex : * 5 + 4 3

Pour ce faire il faut programmer sous forme de méthodes récursive la grammaire formelle.

Exp()
{
    lecture d'un nouveau caractère dans la chaine
    si caractère est + ou – ou * ou / implique operationBinaire alors retourner construction OpB(c , ,)
    sinon
        si caractère est ~ implique operationUnaire alors retourner construction OpU('-' , Exp())
    ...
}
Astuces techniques

- **caractère chiffre -> entier :**

```
c='1' ;
String s1=""+c;
int v= Integer.parseInt(s1)
```

- **extraire un caractère d'une chaîne**

```
char c = chaine.charAt(index );
```