m2103

TM confiné

Projet calculette (phase 2)

Table des matières

1 Présentation

1.1 Directions

- **Ce qu'on a fait** : expressions qui sont des sommes/différence de nombres `15 + 2 - 7 + 1`
- **Ce qu'on va faire maintenant :** somme/différence de produits/quotients de nombres : `2\*3 + 4\*5\*6 - 10/2`
- **Ce qu'on garde pour plus tard** : expressions parenthésées : `(2+3) \* (4+5)`

1.2 Terminologie

Fixons la terminologie

- la calculette lit et évalue une **expression**. Cette expression va jusqu'à la marque de fin de la ligne.
- une **expression** est une somme/différence d'un ou plusieurs **termes**
- un **terme** est un produit/quotient de **facteurs**,
- un **facteur**, pour l'instant, ne peut être qu'un nombre. Plus tard, il y aura aussi des expressions entre parenthèses.

1.3 Comment travailler

1. Sauvez votre projet existant sous GITLAB
2. Dupliquez-le sous un autre nom (calculette-v2, par exemple), dans le même répertoire GITLAB.

Vous travaillerez dans ce nouveau projet Netbeans.

2 Ré-usinage du code existant.

Avant de nous lancer dans l'extension des fonctionnalités, nous allons "ré-usiner" le code existant.

> Le **ré-usinage** consiste à retravailler le code source d'un programme, sans y ajouter de fonctionnalités ni en corriger les bogues, pour le restructurer, améliorer la lisibilité, et donc faciliter sa maintenance.

Notre objectif est qu'il corresponde mieux à la description :

- La calculette lit et évalue une **expression**.
- Cette expression va jusqu'à la marque de fin de la ligne.

et cela permettra d'envisager la maintenance évolutive (ajout de fonctionnalités) plus sereinement.

2.1 Le code existant

Pour l'instant, votre code doit ressembler à ceci.

```
public int evaluation(String line) ... {
    return evaluationSommeDifference(line);
}

private int evaluationSomme(String line) throws SyntaxErrorException {
    Tokenizer tokenizer = new Tokenizer(line);
    Token token = tokenizer.get();
    checkSyntax(token.isNumber(), "Number expected");
    int total = token.value();

    token = tokenizer.get();
    while (token.isSymbol("+")) {
        token = tokenizer.get();
        checkSyntax(token.isNumber(), "Number expected");
        total += token.value();
        token = tokenizer.get();
    }
    checkSyntax(token.isEnd(), ....);
    return total;
}
```

dans lequel vous avez ajouté le traitement de la soustraction.

2.2 Considérations diverses

1. la méthode `evaluation()` doit faire appel à une méthode chargée de **lire et évaluer une expression** grâce à un `Tokenizer`. Il y aura d'autres méthodes pour lire et évaluer des termes, des facteurs, des nombres, etc. Il est donc raisonnable de lui donner un nom comme `get_expr_value()`.

2. Toutes ces fonctions travaillent sur la même ligne reçue par la méthode `evaluation()`. Elles partagent le même "tokenizer" qui découpe la ligne, et la variable `token` qui contient le prochain élément à traiter. Ces variables `tokenizer` et `token` seront communes (attributs de `Calculator`), et initialisées par `evaluation()`.

3. Le rôle précis de `get_expr_value()` est de lire et évaluer une expression, et de laisser dans `token` le premier symbole qui **suit** l'expression. C'est donc `evaluation()` qui vérifiera qu'après l'expression, `token` contient la marque de fin.

2.3 Modification de `evaluation()`

Cette clarification des responsabilités conduit au nouveau code

```
// attributs, au lieu de variables locales à la méthode
private Tokenizer tokenizer;
private Token token;
```

```
public int evaluation(String line) ... {

    tokenizer = new Tokenizer(line);
    getNextToken();                          // voir ci dessous

    int value = get_expr_value();
    checkSyntax(token.isEnd(), ...);

    return value;
}

private void getNextToken() {
    token = tokenizer.get();
}
```

On en a profité pour introduire la méthode `getNextToken()` qui simplifie les écritures. Propagez cette modification.

2.4 Méthode `get_expr_value()`, première version

Ceci fait, la méthode `evaluationSomme` se transforme en `get_expr_value()`, en supprimant ce qui est pris en charge par `evaluation()` :

```
private int get_expr_value() ... {
    // (suppression)
    checkSyntax(token.isNumber(), "Number expected");
    int total = token.value();
    getNextToken();

    while (token.isSymbol("+")) {
            getNextToken();
            checkSyntax(token.isNumber(), "Number expected");
            total += token.value();
            getNextToken();
    }
    // (suppression)
    return total;
}
```

2.5 Introduction de `get_number_value()`

Dans le code, on trouve deux séquences similaires, qui consistent à

- vérifier que le `token` courant représente un *nombre*,
- récupérer sa valeur,
- faire avancer au token suivant.

On en fait naturellement une méthode `get_number_value()` :

```
private int get_number_value() ....
{
    checkSyntax(token.isNumber(), "Number expected");
    int value = token.value();
    getNextToken();
    return value;
}
```

2.6 Méthode `get_expr_value()`, seconde version

L'introduction de cette méthode `get_number_value()` simplifie énormément `get_expr_value()` :

```
private int get_expr_value() ... {
    int total = get_number_value();
    while (token.isSymbol("+")) {
            getNextToken();
            total += get_number_value();
    }
    return total;
}
```

2.7 À faire

1. Introduisez des tests, comme :

```
@Test
void testSums() {
    Calculator calc = new Calculator();
    assertEquals(123, calc.evaluation("  123 ");
    assertEquals(4, calc.evaluation("2+2"));
    assertEquals(1, calc.evaluation("421-20-400");
}
```

Ces **tests de non-régression** (dits aussi "tests de régression") auront pour but de vérifier que les modifications ne changent pas le comportement du programme.

2. Vérifiez qu'ils réussissent.
3. Modifiez votre propre code, en utilisant les tests de non-régression pour vérifier qu'il fonctionne toujours.

3 Maintenance évolutive

Si on restructure le code, c'est pour qu'il soit plus facile à **maintenir**. On distingue :

- **maintenance évolutive** = ajouter/améliorer des fonctionnalités,
- **maintenance corrective** = éliminer des erreurs.

Ici il s'agit de *maintenance évolutive*. Notre calculette doit maintenant traiter une expression qui est une somme/différence de **termes** avec des produits et des quotients de nombres.

3.1 Introduire des tests

À faire : Introduire quelques tests de non-régression supplémentaires, par exemple

- 2*3 = 7
- 7/2 = 3
- 2*3 + 4*5 = 26

Vérifiez aussi que la division par zéro lève une exception :

```
@Test
void testDivisionZero() {
     Calculator calc = new Calculator();
     try {
         calc.evaluation("1 + 2/0 + 3");
         fail();
     } catch (EvaluationErrorException ex) {
     }
 }
```

3.2 Transformation du code

Puisqu'une expression est maintenant une somme/différence de **termes** plutôt que de nombres, le code :

```
private int get_expr_value() ... {
    int total = get_number_value();
    while (token.isSymbol("+")) {
            getNextToken();
            total += get_number_value();
    }
    return total;
}
```

va être transformé en :

```
private int get_expr_value() ... {
    int total = get_term_value();
    while (token.isSymbol("+")) {
            getNextToken();
            total += get_term_value();
    }
    return total;
}
```

et de la même façon un terme sera un produit/quotient de facteurs, qui sont (pour l'instant) des nombres.

Écrivez les méthodes `get_term_value()`, `get_factor_value()`, `get_number_value()`.

4 Expressions parenthésées

4.1 Les expressions parenthésées sont des facteurs

Les expressions parenthèsées apparaissent comme des **facteurs** dans les termes

```
1+ 2*(3+4)*5 + 6
     *****
```

Un facteur est donc

- soit un nombre,
- soit formé d'une parenthèse ouvrante, une expression, et une parenthèse fermante.

4.2 Algorithme de reconnaissance

Il suffit de modifier légèrement l'algorithme qui évalue les facteurs

```
si le token est un nombre
    appeler get_number_value()
    et retourner sa valeur
sinon si c'est une parenthèse ouvrante
    avancer au token suivant
    appeler get_expr_value()
    verifier qu'on est sur la parenthèse fermante
    avancer au token suivant
    retourner la valeur de l'expression
sinon
    erreur de syntaxe
```

4.3 Travail

- introduire des tests
- modifier `get_factor_value`

5 Changement de signe

Pendant qu'on y est, pensons que le signe moins peut aussi servir pour indiquer un changement de signe. Exemples : `-3`, `8/-(2+3)`.

Une solution est de décrêter que

- un facteur peut aussi être composé d'un signe "-" suivi d'un facteur,
- l'évaluation d'un tel facteur retourne la valeur opposée.

et de le traduire dans le code.

6 Annexe : syntaxe des expressions

Description formelle de la syntaxe des expressions, dans une variante de l'EBNF (Extended Backus-Naur Form) :

```
expr    = term   { addOpp term }          // expression
term    = factor { multOp factor }        // terme
factor  =  number                         // facteur
        |  "(" expr ")"
        |  unaryOp factor
addOp   = "+" | "-"                       // opérateurs "additifs"
multOp  = "*" |  "/"                      // opérateurs multiplicatifs
unaryOp = "-"                             // opérateur unaire

number  = digit { digit }
digit   = "0" | "1" | ... | "9"
```

Conventions : définition d'une notion par `=`, alternative par `|`, repétition entre accolades (0, 1 ou plusieurs fois), texte entre guillemets.

C'est une description simplifiée, qui ne montre pas le rôle des espaces.