TP7

Romain MAURICE

Exercice1

1) On va créer Expr dans un fichier Expr.java avec le main suivant. public static void main(String[] args) { Expr expression = new Add(new Value(2), new Value(3)); Expr expression2 = new Sub(new Mul(new Value(2), new Value(3)), new Value(4)); } Et pour Value, Add, Sub et Mul, on utilisera des records chacun dans son propre fichier .java. Créer les records avec leurs composants nécessaires pour que le main compile. Remarquez que expression définit le première arbre présenté dans le sujet. Dessinez l'arbre à la main correspondant à expression2.

```
public interface Expr {}
```

```
public record Value(int value) implements Expr{}
```

```
public record Sub(Expr left,Expr right) implements Expr{
    public Sub{
        Objects.requireNonNull(left);
        Objects.requireNonNull(right);
    }
```

```
public record Mul(Expr left,Expr right) implements Expr{
    public Mul{
        Objects.requireNonNull(left);
        Objects.requireNonNull(right);
    }
```

```
public record Add(Expr left,Expr right) implements Expr{
    public Add{
        Objects.requireNonNull(left);
        Objects.requireNonNull(right);
    }
```

2) On souhaite maintenant pouvoir évaluer (trouver la valeur) d'une expression (Expr) en appelant la méthode eval comme ceci public static void main(String[] args) { Expr expression = new Add(new Value(2), new Value(3)); Expr expression2 = new Sub(new Add(new Value(2), new Value(3)), new Value(4)); System.out.println(expression2.eval()); } Modifier votre code en conséquence.

```
public interface Expr {

    public abstract int eval();
```

value

```
    @Override
    public int eval() {
```

```
        return value;
    }
```

Sub

```
    @Override
    public int eval() {
        return left.eval()-right.eval();
    }
```

Mul

```
    @Override
    public int eval() {
        return left.eval()*right.eval();
    }
```

Add

```
    @Override
    public int eval() {
        return left.eval()+right.eval();
    }
```

3) Écrire une méthode parse qui prend un Scanner en entrée et crée l'arbre d'expression correspondant sachant que l'arbre sera donné au scanner en utilisant la notation préfixe (opérateur devant). Par exemple, au lieu de 2 + 3 - 4, la notation préfixe est - + 2 3 4. Indication : la méthode parse est naturellement récursive. Si l'expression contient encore des symboles (et qu'elle est bien formée) alors: soit le prochain symbole est un opérateur et il faut appeler parse() 2 fois pour obtenir le fils gauche et le fils droit et les combiner avec l'opérateur pour faire une nouvelle expression, soit le prochain symbole est un entier et il suffit d'en faire une feuille de l'arbre d'expression. Enfin, pour rappel, scanner.next() renvoie le prochain mot, Integer.parseInt() permet de convertir une String en int et il est possible d'utiliser le switch (le switch qui renvoie une valeur) sur des Strings en Java. Pour cette question, on ne vous demande pas de vérifier que l'expression fournie en notation préfixe est bien formée. Pour les plus à l'aise, vous pouvez tenter de faire une gestion propre des cas où l'expression est mal-formée

```
    public static Expr parse(Scanner scanner) {
        var op = scanner.next();

        switch (op) {
            case "+": {
                return new Add(parse(scanner),parse(scanner));
            }
            case "-": {
                return new Sub(parse(scanner),parse(scanner));
            }
            case "*": {
                return new Mul(parse(scanner),parse(scanner));
            }
            default:
```

```
                return new Value(Integer.parseInt(op));
        }
    }
```

4) Il y a un bug dans le code que l'on a écrit, on permet à n'importe qui d'implanter Expr mais cela ne marchera pas avec la méthode parse qui elle liste tous les sous-types possibles. Comment corriger ce problème ?

-On va utiliser une interface scellé pour controler qui peut implémenter cette interface

```
public sealed interface Expr permits Sub, Value, Add, Mul {
```

5) Déplacer le main dans une nouvelle classe Main dans le package fr.uge.calc.main et faire les changements nécessaires.

var test = parse(scanner).eval(); -> var test = Expr.parse(scanner).eval();

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args){
        Expr expression = new Add(new Value(2), new Value(3));
        Expr expression2 = new Sub(new Mul(new Value(2), new Value(3)), new Value(4));
        System.out.println(expression.eval());
        System.out.println(expression2.eval());


        Scanner scanner = new Scanner("- + 2 3 4");
        var test = Expr.parse(scanner).eval();
        System.out.println(test);
    }
}
```

6) Noter que prendre un Scanner en paramètre ne permet pas de ré-utiliser la méthode parse si, par exemple, l'expression à parser est stockée dans une List de String. Quelle interface que doit-on utiliser à la place de Scanner pour que l'on puisse appeler la méthode parse avec un Scanner ou à partir d'une List.

-On va utiliser Iterator en paramètre à la place de Scanner

```
    public static Expr parse(Iterator<String> iterator) {
        var op = iterator.next();

        switch (op) {
            case "+": {
                return new Add(parse(iterator),parse(iterator));
            }
            case "-": {
                return new Sub(parse(iterator),parse(iterator));
            }
            case "*": {
                return new Mul(parse(iterator),parse(iterator));
            }
```

```
            default:
                return new Value(Integer.parseInt(op));
        }
    }
```

```
        Scanner scanner = new Scanner("- + 2 3 4");
        Scanner scanner2 = new Scanner("- + 2 3 4");
        List<String> list = new ArrayList<>();
        list.add("-");
        list.add("+");
        list.add("2");
        list.add("3");
        list.add("4");

        var test = Expr.parse(scanner);
        var test2 = Expr.parse(scanner2).eval();
        var test3 = Expr.parse(list.iterator()).eval();

        System.out.println(test);
        System.out.println(test2);
        System.out.println(test3);
```

7) Écrire la méthode d'affichage de l'arbre d'expression pour que l'affichage se fasse dans l'ordre de lecture habituel. Note : il va falloir ajouter des parenthèses et peut-être des paranthèses inutiles !

8) Enfin, on peut voir que le code de eval dans Add, Sub et Mul est quasiment identique, dans les trois cas : la méthode eval est appelée sur left et right. On souhaite factoriser ce code (on ne le ferait probablement pas dans la vraie vie car il n'y a pas assez de code à partager, mais ce n'est pas la vraie vie, c'est un exercice) en introduisant un type intermédiaire BinOp, sous-type de Expr et super-type de Add, Sub et Mul. Le type BinOp doit-il être un record, une classe ou une interface ?

-BinOp doit être une interface pour pouvoir réutiliser les champs des sous-type Add, Sub, Mul 9) Sachant que l'on veut écrire eval dans BinOp, comment eval doit être déclarée ? Et comment, dans eval de BinOp, peut-on accéder aux champs left et right, qui sont déclarés dans Add, Sub et Mul ?

-eval doit être déclaré comme "default" pour ne pas avoir besoin de le redéfinir dans les sous-classes. On peut accéder simplement aux champs left et right en définissant "Expr left(); Expr right();" dans l'interface.

10) Écrire le code de BinOp (dans BinOp.java) et modifier Add, Sub et Mul en conséquence.

```
public sealed interface BinOp extends Expr permits Add, Sub, Mul {

    Expr left();
    Expr right();

    default int eval() {

        int left = left().eval();
```

```
        int right = right().eval();

        switch(this.getClass().getSimpleName()) {
            case "Add":
                return left+right;
            case "Mul":
                return left*right;
            case "Sub":
                return left-right;
            default:
                throw new IllegalArgumentException();
        }

    }
}
```

```
import java.util.Objects;

public record Add(Expr left,Expr right) implements BinOp{
    public Add{
        Objects.requireNonNull(left);
        Objects.requireNonNull(right);
    }
}
```

```
import java.util.Objects;

public record Mul(Expr left,Expr right) implements BinOp{
    public Mul{
        Objects.requireNonNull(left);
        Objects.requireNonNull(right);
    }
}
```

```
import java.util.Objects;

public record Sub(Expr left,Expr right) implements BinOp{
    public Sub{
        Objects.requireNonNull(left);
        Objects.requireNonNull(right);
    }
}
```