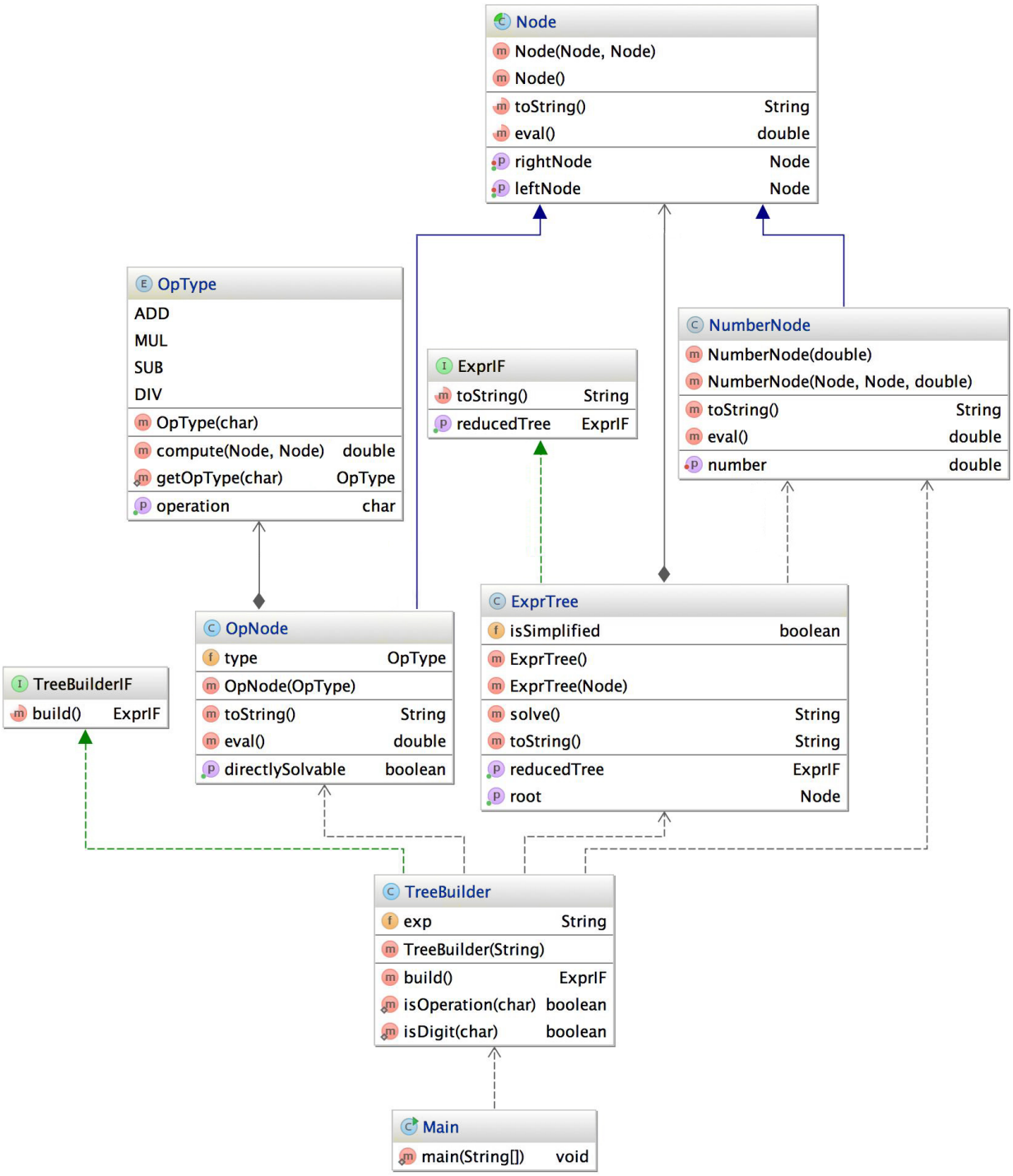
Mini-projet 1 - Manipulation d'expressions arithmétiques

1. Diagramme des classes



2. Choix d'implémentation

Afin d'implémenter l'arbre syntaxique abstrait, nous avons utilisé une structure chainée. Ce type d'implémentation a été préféré à un tableau dynamique car la complexité spatial avec un tableau est de $\Theta(2^n)$ alors qu'avec une structure chainée la complexité spacial est de $\Theta(n)$.

De plus, une structure chainée était un choix obligatoire vu la contrainte d'utiliser des méthodes récursives.

3. Complexité des différentes méthodes

Soit n, le nombre d'opérations et de nombres : n représente donc la taille du problème à résoudre.

3.1 TreeBuilderIF.build()

Complexité temporelle en $\Theta(n)$ car :

1. L'expression sera toujours parcourue entièrement 1 fois, à l'aide d'une boucle.
2. La boucle principale ne contient que des opérations de complexité $\Theta(1)$.

On en déduit donc que le temps d'exécution augmentera de façon linéaire par rapport à la taille du problème.

3.2 ExprIF.getReducedTree()

Complexité temporelle en $\mathcal{O}(\log(n))$ car :

1. Complexité temporelle en $\mathcal{O}(h)$, où h est la hauteur de larbre
2. L'arbre est équilibré donc h $\in \Theta(\log_2(n))$

Donc,

— Meilleur cas : $\Theta(1)$ (on tombe sur la racine)
— Pire cas : $\Theta(\log(n))$
— En général : $\mathcal{O}(\log(n))$

3.3 ExprIF.toString()

Complexité temporelle en $\mathcal{O}(n)$ car on passe, dans le pire des cas, 3x par chaque noeud de l'arbre (parcours d'Euler). On a donc une complexité en $\mathcal{O}(3n)$. On peut "simplifier" le 3 car il s'agit d'une constante. On obtient donc $\mathcal{O}(n)$

— Meilleur cas : $\Theta(1)$ (L'arbre est entièrement simplifié)
— Pire cas : $\Theta(n)$
— En général : $\mathcal{O}(n)$

4. Difficultés rencontrées

— Quelques difficultés à implémenter une méthode récursive
— Choix d'implémentations

Annexe

TreeBuilder

```
public ExprIF build() {
    Stack<Node> stack = new Stack<Node>();
    for (int i = 0; i < exp.length(); i++) {
        char currentChar = exp.charAt(i);
        if (isDigit(currentChar)) {
            StringBuilder number = new StringBuilder();
            while (i < exp.length() && isDigit(exp.charAt(i))) {   //Tant
                que le nombre n'est pas fini
                number.append(exp.charAt(i));
                i++;
            }
            Node Node = new
                NumberNode(Double.parseDouble(number.toString()));
            stack.push(Node);

        } else if (isOperation(currentChar)) {

            Node Node = new OpNode(OpType.getOpType(currentChar));
            stack.push(Node);

        } else if (currentChar == ')') {
            Node rightNode = stack.pop();
            Node root = stack.pop();
            Node leftNode = stack.pop();
            root.setLeftNode(leftNode);
            root.setRightNode(rightNode);
            stack.push(root);
        }

    }
    return new ExprTree(stack.pop()); // Retourne la racine de l'AST
}
```

ExprIF

```
public String toString() {
    String str = null;
    if (root.getLeftNode() == null && root.getRightNode() == null) {
        return root.toString();
    } else if (root.getLeftNode() != null && root.getRightNode() != null) {
        ExprTree LeftTree = new ExprTree(root.getLeftNode());
        ExprTree RightTree = new ExprTree(root.getRightNode());
        str = "(" + LeftTree.toString() + root.toString() +
            RightTree.toString() + ")";
    }
    return str;
}
```

```java
public ExprIF getReducedTree() {

    if (getRoot().getLeftNode() != null && getRoot().getRightNode() != null) {
        ExprTree subLeft = new ExprTree(root.getLeftNode());
        ExprTree subRight = new ExprTree(root.getRightNode());
        ExprTree eL = (ExprTree) subLeft.getReducedTree(); // Appel récursif
        ExprTree eR = (ExprTree) subRight.getReducedTree(); // Appel récursif

        if (getRoot() instanceof OpNode) {

            if (((OpNode) getRoot()).isDirectlySolvable() && (!eL.isSimplified || !eR.isSimplified)) {
                root = new NumberNode(root.eval());
                isSimplified = true;
                return this;
            }
        }

        if (eL.isSimplified) {
            this.getRoot().setLeftNode(eL.getRoot());
            return new ExprTree(this.getRoot());
        }

        if (eR.isSimplified) {
            this.getRoot().setRightNode(eR.getRoot());
            return new ExprTree(this.getRoot());
        }


    }
    return this;
}
```