Programmation et modélisation orientées objet
(USAL5A & USAL2H)

Structures arborescentes

La version 14 de Java a introduit le mot clé `record` permettant la définition concise d'un type enregistrement. Une telle définition correspond en fait à une classe avec des champs finaux privés pour les paramètres, et où le constructeur, les accesseurs et les méthodes `equals`, `hashCode` et `toString` sont générées automatiquement.

Utilisés de manière combinée pour implanter une interface, les *records* permettent à leur tour de définir un « type récursif » (aussi appelé « type algébrique », ou « type variant » s'il n'est pas récursif). Si l'interface est de plus « scellée » (*sealed*), les constructeurs des types *record* définis sont les seuls permettant d'implanter l'interface, autorisant ainsi des définitions de méthode par « filtrage » (*pattern matching*), introduit dans Java 21.

Dans ce TP, nous allons voir comment les types de données récursifs permettent de définir simplement des structures arborescentes. Nous considérons pour cela trois exemples de types récursifs, utilisés typiquement dans des cadres différents : en algorithmique (les arbres binaires), en compilation (les arbres de syntaxe abstraite) et en réseau (le format JSON pour échanger les données dans des applications web).

1 Arbres binaires

On choisit d'implanter les arbres binaires génériques (immutables) de la manière suivante :

```
sealed interface Tree<A> {
    record Leaf<A>(A value) implements Tree<A> {}
    record Node<A>(Tree<A> left, Tree<A> right) implements Tree<A> {}
}
```

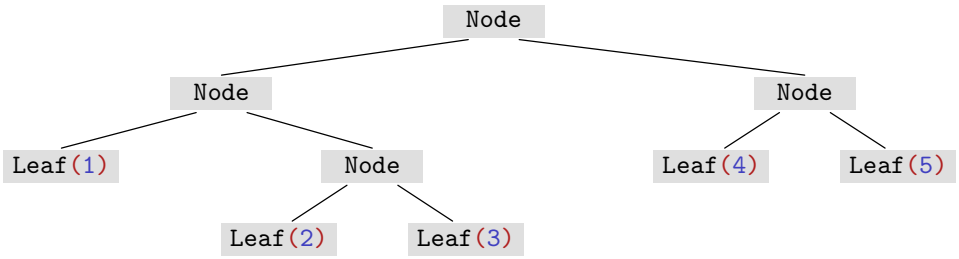
- Le constructeur `Leaf` permet de construire un arbre réduit à une feuille en donnant en paramètre la valeur associée (de type générique `A`).

- Le constructeur `Node` permet de construire un arbre dont la racine est un nœud en donnant en paramètre les deux sous-arbres.

Par exemple, on peut construire l'arbre binaire suivant :

```
new Node<>(
    new Node<>(
        new Leaf<>(1),
        new Node<>(
            new Leaf<>(2),
            new Leaf<>(3))),
    new Node<>(
        new Leaf<>(4),
        new Leaf<>(5)))
```

Cet arbre peut aussi être représenté sous forme graphique :



Questions.

1. Implanter la méthode suivante en utilisant le filtrage et la récursivité :

   ```
   static <A> Integer size(Tree<A> t)
   ```

   qui renvoie la taille (le nombre de feuilles) de l'arbre `t`.

2. (*optionnel*). Optimiser l'implantation de `size` en stockant la taille des sous-arbres dans les nœuds.

3. Implanter la méthode suivante en utilisant le filtrage et la récursivité :

   ```
   static <A> List<A> toList(Tree<A> t)
   ```

   `toList(t)` construit la liste de valeurs contenues dans les feuilles de `t` (de gauche à droite). On implantera au préalable une méthode statique `concat` :

   ```
   static <A> List<A> concat(List<A> l1, List<A> l2)
   ```

2 Expressions arithmétiques simples

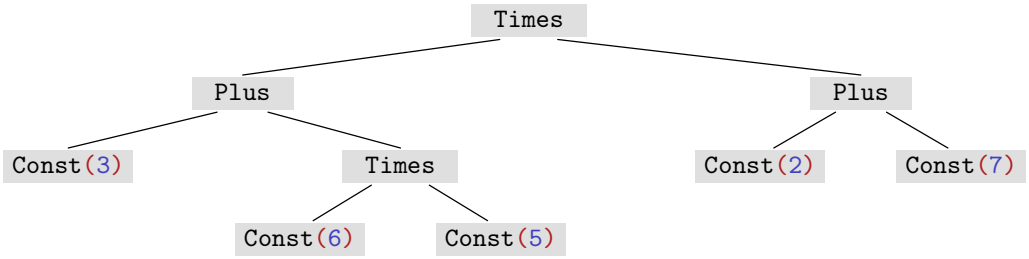
On considère la définition du type `Expr` des expressions arithmétiques simples suivantes :

```
sealed interface Expr {
    record Const(Integer i) implements Expr {}
    record Minus(Expr e1) implements Expr {}
    record Plus(Expr e1, Expr e2) implements Expr {}
    record Times(Expr e1, Expr e2) implements Expr {}
}
```

Par exemple, on peut construire l'expression `(3 + 6 * 5) * (2 + 7)` ainsi :

```
new Times(
    new Plus(
        new Const(3),
        new Times(
            new Const(6),
            new Const(5))),
    new Plus(
        new Const(2),
        new Const(7)))
```

L'*arbre de syntaxe abstraite* de cette expression peut aussi être représenté sous forme graphique :



Questions.

1. En utilisant le filtrage et la récursivité, écrire une méthode `stringOf` qui permet d'afficher une expression donnée :

   ```
   static String stringOf(Expr e)
   ```

2. En utilisant le filtrage et la récursivité, écrire une méthode `eval` qui calcule la valeur d'une expression donnée :

   ```
   static Integer eval(Expr e)
   ```

3 Expressions arithmétiques avec variables

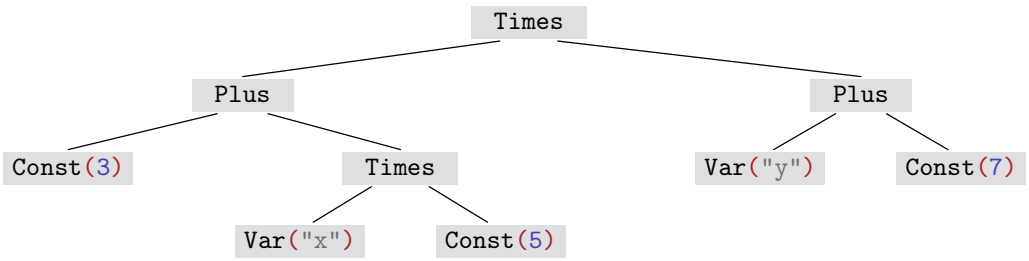
De manière à prendre en compte des variables dans les expressions arithmétiques, on complète la définition précédente de `Expr` avec :

```
record Var(String s) implements Expr {}
```

Par exemple, on peut maintenant construire l'expression `(3 + x * 5) * (y + 7)` :

```
new Times(
    new Plus(
        new Const(3),
        new Times(
            new Var("x"),
            new Const(5))),
    new Plus(
        new Var("y"),
        new Const(7)))
```

L'*arbre de syntaxe abstraite* de cette expression sous forme graphique devient :



Questions.

1. Compléter le code de la méthode `stringOf` qui permet d'afficher une expression :

   ```
   static String stringOf(Expr e)
   ```

2. Modifier la définition de `eval` pour pendre aussi en argument un paramètre `m` qui représente la mémoire et associe une valeur à chaque variable. Son prototype est maintenant le suivant :

   ```
   static Integer eval(Expr e, Map<String, Integer> m)
   ```

3. Implanter plusieurs tests unitaires pour cette méthode `eval` pour différentes valeurs de la mémoire.

4 Type de données en JSON

On choisit d'implanter un type de données pour le format JSON de la manière suivante[1] :

```
sealed interface JsonValue {
    record JsonString(String s) implements JsonValue {}
    record JsonNumber(double d) implements JsonValue {}
    record JsonNull() implements JsonValue {}
    record JsonBoolean(boolean b) implements JsonValue {}
    record JsonArray(List<JsonValue> values) implements JsonValue {}
    record JsonObject(Map<String, JsonValue> pairs) implements JsonValue {}
}
```

1. https://www.infoq.com/articles/data-oriented-programming-java

Par exemple, l'expression arithmétique `3 + 6 * 5`, dont l'AST de type `Expr` est le suivant :

```
Plus[e1=Const[i=3], e2=Times[e1=Const[i=6], e2=Const[i=5]]]
```

sera représentée en JSON sous la forme suivante (selon la spécification Jakarta EE) :

```
{
        "@type": "Plus",
        "e1": {
                "@type": "Const",
                "i": 3
        },
        "e2": {
                "@type": "Times",
                "e1": {
                        "@type": "Const",
                        "i": 6
                },
                "e2": {
                        "@type": "Const",
                        "i": 5
                }
        }
}
```

Questions.

1. En utilisant le filtrage et la récursivité, écrire une méthode statique `jsonStringOf` qui convertit en `String` une valeur JSON donnée (sur une seule ligne) :

   ```
   static String jsonStringOf(JsonValue v)
   ```

   Tester cette méthode en écrivant des tests unitaires jUnit.

2. (*optionnel*). Donner une autre version de votre méthode `jsonStringOf` qui génére une `String` muliligne et indentée comme dans l'exemple ci-dessus.

   Tester cette méthode en écrivant des tests unitaires jUnit.

3. En utilisant le filtrage et la récursivité, écrire une méthode statique `toJson` qui convertit une expression (vue dans l'exercice précédent) en `JsonValue`.

   ```
   static JsonValue toJson(Expr e)
   ```

   Tester cette méthode en écrivant des tests unitaires jUnit. On testera au moins l'exemple ci-dessus.

4. Ecrire une méthode statique `toJson` qui convertit une liste d'expression en `JsonValue` (on utilisera `JsonArray` pour cela).

   ```
   static JsonValue toJson(List<Expr> e)
   ```

   Tester cette méthode en écrivant des tests unitaires jUnit.

A Diagrammes de classes

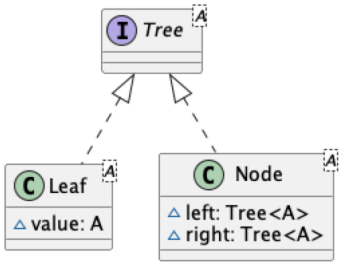


Figure 1. Tree

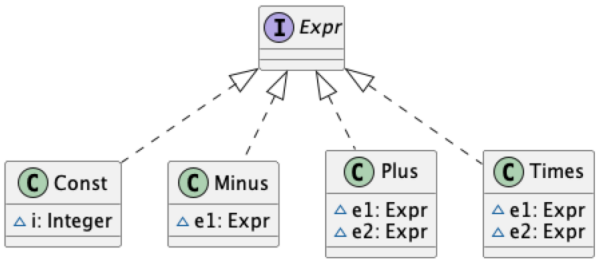


Figure 2. Expr

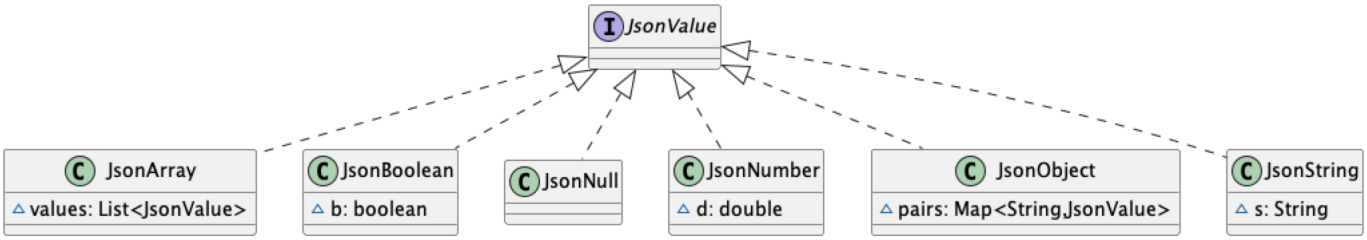


Figure 3. JsonValue