Conception de logiciels intranet : intergiciels à objets répartis (NSY102)

Les types algébriques en Java

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 de données algébrique » (aussi appelé « type récursif », 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.

1. On considère la définition du type `Form` des formules propositionnelles suivant (ces constructeurs correspondent respectivement aux opérateurs logiques « $\top, \bot, \neg, \wedge, \vee$ ») :

   ```
   sealed interface Form {}
   record Top() implements Form {}
   record Bottom() implements Form {}
   record Not(Form f1) implements Form {}
   record And(Form f1, Form f2) implements Form {}
   record Or(Form f1, Form f2) implements Form {}
   ```

2. Ecrire une méthode `stringOf` qui permet d'afficher une formule donnée :

   ```
   static String stringOf(Form f)
   ```

3. Ecrire une méthode `eval` qui calcule la valeur de vérité d'une formule donnée :

   ```
   static Boolean eval(Form f)
   ```

4. Considérez la méthode statique `oper` définie ci-dessous. A quoi correspond-elle ?

   ```
   static Form oper(Form f1, Form f2) {
       return new Not(new And(new Not(f1), new Not(f2)));
   }
   ```

   Définir l'implication `imply` (opérateur logique « $\Rightarrow$ ») de la même manière.

5. On souhaite simplifier une formule en supprimant les occurrences de `Not (Not ...)`. Ecrire la méthode `static Form simpl(Form f)` qui implante cette simplification.

6. On considère la définition du type `Expr` des expressions arithmétiques :

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

7. Ecrire une méthode `stringOf` qui permet d'afficher une expression donnée :

   ```
   static String stringOf(Expr f)
   ```

8. Ecrire une méthode `eval` qui calcule la valeur d'une expression donnée :

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

9. Ecrire une méthode `static Expr subtract(Expr e1, Expr e2)` en utilisant les constructeurs `Plus` et `Minus`.

10. Ecrire une méthode `static Expr translate(Form f)` qui traduit une formule en expression, en utilisant comme toujours les constantes 1 et 0 pour représenter `Top` et `Bottom`.

11. En déduire une autre fonction d'évaluation des formules `Boolean eval2(Form f)` utilisant l'évaluation des expressions.

12. 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 {}
    ```

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

    ```
    static Integer eval(Expr e, Function<String, Integer> v)
    ```

    Donner une dernière version de `eval` où `v` est de type `Map<String, Integer>`.