LIFPF – Programmation fonctionnelle
TD6 – Modules et foncteurs

Licence informatique UCBL – Printemps 2022–2023

Exercice 1 : Ensembles paramétrés On souhaite créer un foncteur pour représenter des ensembles codés par des listes ordonnées. Pour gérer les résultats de comparaison, on s'appuie sur le type suivant :

```
type cmp_r = Lt | Eq | Gt
```

1. Définir une signature `SEnsemble` pour représenter les modules qui définissent les ensembles, comprenant :
   - un type `elt` pour les éléments des ensembles ;
   - un type `ens` pour représenter des ensembles d'`elt`
   - une fonction `appartient` qui indique si un élément est dans un ensemble
   - une constante `vide` qui représente l'ensemble vide
   - une fonction `singleton` qui renvoie le singleton fabriqué à partir d'un élément
   - une fonction `union` qui calcule l'union de deux ensembles
   - une fonction `intersection` qui calcule l'intersection de deux ensembles
   - une fonction `card` qui donne la taille d'un ensemble
2. Définir un module `EnsInt` de signature `SEnsemble` où le type des éléments est `int`. On utilisera une structure de liste triée pour l'implémentation des ensembles. On se confronte à un problème de masquage, lequel ? Expliquer comment contourner ce problème en ajoutant une contrainte `with` dans la signature de `EnsInt`.
3. Définir une signature `SCmp` définissant un type `t` et une fonction de comparaison `cmp` qui renvoie une valeur de type `cmp_r` et qui indique si sont premier argument est strictement inférieur (`Lt`), égal (`Eq`) ou strictement supérieur (`Gt`) à son deuxième argument. En donner une implémentation avec des `int`.
4. Définir un foncteur `ListEns` qui prend en argument un module de signature `SCmp` et créée un module de signature `SEnsemble` où le type `elt` est le type `t` de l'argument du foncteur. Ce foncteur implémentera les opérations ensemblistes en s'appuyant sur une structure de liste triée.
5. Étendre la signature `SEnsemble` et le foncteur précédent de façon à ce que le module renvoyé contienne un sous-module qui soit compatible avec la signature `SCmp` et dont le type des éléments soit le type des ensembles du module principal. À quoi cela peut-il servir ?

Exercice 2 : Foncteurs pour les expressions On souhaite généraliser le traitement d'expressions arithmétiques composées d'`int` et des opérations $+$, $-$, $\times$ et $/$. Plus précisement, on souhaite découpler d'une part le traitement qui est fait des expressions et d'autre part la représentation de ces expressions.

On considérera deux traitements possibles : l'affichage (plus précisement la génération d'une chaîne de caractères représentant l'expression) et l'évaluation.

On considérera par ailleurs deux représentations possibles : une représentation à base d'un type inductif et une représentation polonaise (où les expressions sont représentées par une liste contenant des opérations ou des nombres).

1. Écrire une signature `SEval` décrivant les traitements comprenant un type résultat, une fonction de traduction des constantes `int` en résultat et une fonction d'évaluation pour chaque opération.
2. Écrire un module `StringEval` d'implémentation de `SEval` qui transforme une expression en une chaîne de caractères.
3. Écrire un module `IntEval` d'implémentation de `SEval` qui évalue sous forme `int` les expressions et qui peut déclencher des erreurs de division par zéro.
4. Écrire un foncteur qui prend un module de signature `SEval` et produit un module qui représente les expressions avec un type inductif et fourni une fonction d'évaluation des résultats.
5. Écrire un foncteur qui prend un module de signature `SEval` et produit un module qui représente les expressions en notation polonaise et fourni une fonction d'évaluation des résultats.
6. Discuter de la pertinence de la déclaration des types dans les foncteurs.