Série d'exercices #4

IFT-2035

4.1 Mini évaluateur

Soit les déclarations de type suivantes utilisées pour un mini-interpréteur d'expressions arithmétiques:

```
type Var = String

-- Expressions du code source en forme ASA.
data Exp = Enum Int               -- Une constante
         | Evar Var               -- Une variable
         | Elet Var Exp Exp       -- Une expr "let x = e1 in e2"
         | Ecall Exp Exp          -- Un appel de fonction

-- Valeurs renvoyées.
data Val = Vnum Int               -- Un nombre entier
         | Vprim (Val -> Val)     -- Une primitive
```

Les fonctions prédéfinies sont l'addition, la soustraction, la multiplication, et la division, liées aux variables "+", "-", "*", et "/", respectivement. Ces fonctions prennent deux arguments qui sont passés de manière currifiée. Par exemple une expression telle que "let $x = 3$ in $x + 4$" est représentée par la structure suivante de type *Exp*:

```
sampleExp = Elet "x" (Enum 3)
                 (Ecall (Ecall (Evar "+") (Evar "x")) (Enum 4))
```

L'environnement initial prédéfini les quatre fonctions:

```
mkPrim :: (Int -> Int -> Int) -> Val
mkPrim f = Vprim (\(Vnum x) -> Vprim (\(Vnum y) -> Vnum (f x y)))

-- L'environnement initial qui contient toutes les primitives.
type Env = [(Var, Val)]
pervasive :: Env
pervasive = [("+", mkPrim (+)), ("-", mkPrim (-)),
             ("*", mkPrim (*)), ("/", mkPrim div)]
```

Écrire la fonction *eval* qui prend un environnement qui décrit les variables liées (et leur valeur) ainsi qu'une expression et qui renvoie le résultat de l'évaluation de l'expression. I.e.

```
eval :: Env -> Exp -> Val
```

et

```
eval pervasive sampleExp
```

renvoie `Vnum 7`.

4.2 Renommage α

Soit le code ci-dessous qui est écrit en Haskell et utilise donc la portée lexicale:

```
x -> y ->
let f = x -> x + 2 in
let g x = g -> f (g x) in
let g (x, f) = f x
in f -> g (x, f)
```

Renommer toutes les variables (e.g. en y ajoutant un 0, 1, 2, ...) pour que chaque variable ait un nom différent des autres. Bien sûr ce renommage ne doit pas changer la sémantique du code.

4.3 Ordre et Portée

Définir dans un langage fonctionnel hypothétique une fonction qui renvoie:

- 0 si le langage utilisé obéi la portée statique et l'appel par valeur
- 1 si le langage utilisé obéi la portée statique et l'appel par nom
- 2 si le langage utilisé obéi la portée dynamique et l'appel par valeur
- 3 si le langage utilisé obéi la portée dynamique et l'appel par nom

4.4 Tracer la portée

Soit le code suivant dans un langage hypothétique dont la syntaxe est la même que celle de Haskell:

$$
\begin{array}{l}
\mathsf{let}\ x = 2 \\
\quad f_1\ y = z + x + y \\
\quad f_2\ x = f_1\ (x + 1) \\
\quad f_3\ z = f_2\ (z + 2) \\
\mathsf{in}\ f_3\ 5
\end{array}
$$

Montrer les étapes de l'évaluation dans chacun des deux cas: le cas où le langage utilise la portée dynamique et le cas où il utilise la portée statique. De même avec l'exemple suivant:

$$
\begin{array}{l}
\mathsf{let}\ m\ f\ [] = [] \\
\quad m\ f\ (x : xs) = f\ x : m\ f\ xs \\
\\
\quad i = 4 \\
\mathsf{in}\ m\ (\lambda x \to x + i)\ [2, 3]
\end{array}
$$

Utiliser une notation basée sur des environnements dénotés
$\{x_1 \mapsto v_1, x_2 \mapsto v_2, ...\}$.