L'église du Lambda-Calcul

On donne le type OCamL récursif suivant :

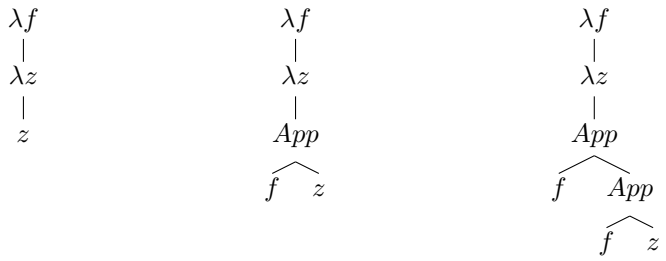
```
type id = string

type op = int -> int -> int function

type expr =
| Var of id
| App of expr * expr
| Lam of id * expr
| Operator of op * expr * expr
```

Il sert à représenter des fonctions, `App` représente la composition, `Lam(x, v)` représente la fonction prenant x et évaluant v en x.

1. Définissez la fonction $x \mapsto x$ en OCamλ.
2. Définissez la fonction $x \mapsto (y \mapsto x)$ en OCamλ.
3. On définit l'entier n par :
$$n = \lambda f.\lambda z.App(f, App(f, \dots z)$$
Ici, on applique f n fois. Les trois arbres ci-dessous sont les représentations dans le codage des entiers de Church de 0, de 1 et de 2.



Donnez une fonction OCamL qui renvoie la représentation d'un entier n.
4. Donnez une fonction OCamL `eval: expr -> (string * int) list -> expr = <fun>` renvoyant l'évaluation d'une expression en une liste de variables. On représentera les entiers comme à la question 3. On pourra commencer par écrire une fonction renvoyant la valeur du deuxième

Solution: L'église du Lambda-Calcul

1. `let id : expr = Lam('x', Var('x'));;`
2. `let p1: expr = Lam('x', Lam('y', Var('x')));;`
3. On procède de manière récursive avec une fonction auxiliaire :

```
let repr_n n =
        let rec aux = function
                | 0 -> Var('z')
                | k -> App(Var('f'), aux (k-1))
        in Lam('f', Lam('z', aux n));;
```

4. On donne déjà la fonction de recherche suivante (on suppose les fonctions correctement évaluées) :

```
let get_val i l =
        let vals = List.filter (fun m -> (fst m) = i) l in
        snd (List.hd vals)
```

```
let rec eval args = function
        |Var(x) -> repr_n (get_val x args)
        |Op(o, e1, e2) -> Op(o, (eval args e1), (eval args e2))
        |Lam(x, e) -> Lam(x, (eval args e))
        |App(e1, e2) -> App((eval args e1), (eval args e2))
```

Types Récursifs – Programmation OCamL 3
Matthieu Boyer
Planche B

Khôlle MP2I
14/10

Peano

On donne le type OCamL récursif suivant pour représenter les entiers :

```
type nat = Zero | Succ of nat;
```

1. Définissez le nombre 2 en OCamL.
2. Définissez des fonctions de conversion `to_nat: int -> nat = <fun>` et `of_nat: nat -> int = <fun>`.
3. Définissez l'addition et la multiplication sur le type `nat`. On n'utilisera pas les fonctions de la question précédente.
4. Définissez une fonction `fact: nat -> nat = <fun>`.

Solution: Peano

1. `let two: nat = Succ(Succ(Zero));;`
2. On donne :

```
let rec to_nat = function
        | 0 -> Zero
        | x -> Succ(to_nat (x - 1))
;;

let rec from_nat = function
        | Zero -> 0
        | Succ(x) -> 1 + (from_nat x)
;;
```

3. On donne :

```
let rec add a = function
        | Zero -> a
        | Succ b -> Succ (plus a b)
;;

let rec multiply a = function
        | Zero -> Zero
        | Succ b -> add a (multiply a b)
;;
```

4.

```
let rec fact = function
        | Zero -> Succ(Zero) (*Renvoyer 1*)
        | Succ n -> multiply (Succ n) (fact n)
;;
```

Types Récursifs – Programmation OCamL 3
Matthieu Boyer
Planche C

Khôlle MP2I
14/10

Première Introduction aux Foncteurs

Voici un type paramétré OCamL : `type 'a maybe = None | Some of 'a;;`

1. Donner un exemple de programme pour lequel le type `int maybe` pourrait être utile.

On va s'intéresser aux manières de faire des transformations sur des objets de type `'a maybe`.

2. Définir une fonction `map: ('a -> 'b) -> 'a maybe -> 'b maybe = <fun>` vérifiant les deux propriétés suivantes :
   (a) `map (fun x -> x) f = f;;`
   (b) `map foo (map bar f) = map (fun x -> foo (bar x)) f;;`

On appelle un type paramétré `f` muni d'une telle fonction `map` un *foncteur*[a]. `maybe` est un exemple de foncteur.

3. Définir le foncteur `lst` dont la fonction `map` est l'application point à point.

Une propriété importante des foncteurs est que la composition de deux foncteurs doit être un foncteur : étant donné un foncteur `'a foo` et un foncteur `'a bar`, on peut définir un foncteur `'a foobar = ('a bar) foo` et définir une fonction `mapFooBar`.

4. Définir le foncteur composition de `maybe` et de `lst`.

On définit de manière similaire la notion de monade. Une monade est un type paramétré `'a f` muni d'une fonction `fmap: ('a -> 'b f) -> 'a f -> 'b f`.

5. Donner les fonctions `fmap` pour les foncteurs précédents.

a. En réalité c'est plus compliqué que cela mais bon...

Solution: Première Introduction aux Foncteurs

1. Pour la fonction `div: int -> int -> int maybe`.
2. `let maybemap f = function None -> None | Some x -> Some (f x);;`
3. Comme en cours :

```
type 'a lst = Nil | Cons of 'a * 'a lst;;

let rec lstmap f = function Nil -> Nil | Cons(a, l) -> Cons (f a) (map l);;
```

4. Comme écrit au dessus :

```
type 'a maybelst = None | Some of 'a lst;;

let maybelstmap f ml = maybemap (lstmap f) ml;;
```

5. On donne :

```
let maybefmap f = function
        | None -> None
        | Some x -> f x
;;

let rec concat l1 l2 = match l1 with
        | [] -> l2
        | Cons(x, l') -> Cons(x, (concat l' l2))
;;

let rec lstfmap f = function
        | Nil -> Nil
        | Cons(x, l) -> concat (f x) (lstfmap f l)
;;

let maybelstfmap f ml = maybefmap (lstfmap f) m:w l;;
```