Programmation fonctionnelle 2024-2025

Feuille 3 : Notion de type

Exercice 3.1 Quel est le type de la fonction `fun x y -> (x, y)` ?

Exercice 3.2 Quel est le type de la fonction `compose` vue précédemment ?

```
let compose f g = fun x -> f (g x)
```

Exercice 3.3 On reprend la suite de Fibonacci définie par $u_0 = 1, u_1 = 1$ et pour tout $n \geqslant 2$, $u_n = u_{n-1} + u_{n-2}$. En utilisant une fonction auxiliaire qui calcule le couple (u_n, u_{n+1}), améliorer `fib` pour que l'appel `fib n` n'utilise qu'un nombre linéaire de sommes.

Exercice 3.4 Quel est le type et la valeur des expressions suivantes ?

```
'x', 2.1, (true, 0)
3 + 2, false || 2 = 3, "bonjour"
let p = 1, 2 in snd p, fst p
```

Exercice 3.5 Pour chacun des types suivants, donner une expression ayant ce type ainsi que la valeur de l'expression.

```
int * bool * string
(int * bool) * string
int * (bool * string)
```

Exercice 3.6 Soient les types `couleur` et `carte` définis comme suit :

```
type couleur = Pique | Coeur | Carreau | Trefle

type carte =
    As of couleur
  | Roi of couleur
  | Dame of couleur
  | Valet of couleur
  | Numero of int * couleur
```

- Écrire un accesseur `couleur_carte carte` de type `carte -> couleur` qui retourne la couleur d'une carte.
- Écrire un prédicat `est_de_couleur carte couleur` de type `carte -> couleur -> bool` qui retourne `true` si `carte` est de couleur `couleur`. On utilisera l'accesseur `couleur_carte`.
- Écrire un prédicat `est_une_figure carte` de type `carte -> bool` qui retourne `true` si `carte` est une figure, `false` sinon.

Exercice 3.7 1. Définir un type `carburant` ayant trois constructeurs `Diesel`, `Essence` ou `Electrique`.

2. Un *véhicule* est caractérisé par son carburant et son nombre de roues. Définir un type `vehicule`

répondant à ces critères.

3. Écrire le constructeur `make_vehicule` de type `carburant -> int -> vehicule`.
4. Écrire les accesseurs `carburant_of` de type `vehicule -> carburant` et `nb_wheels_of` de type `vehicule -> int` qui retournent respectivement le carburant et le nombre de roues d'un véhicule.
5. Lors des pics de pollution, les véhicules diesel à 4 roues au moins sont interdits. Écrire une fonction `can_run : vehicule -> bool` qui teste si un véhicule est autorisé.
6. Pour rouler 100km, un véhicule électrique consomme environ 10kWh, un véhicule diesel consomme environ 6L de carburant, et un véhicule essence consomme environ 8L. Sachant qu'1kWh coûte 0.25 EUR et qu'un litre de carburant coûte 1.5 EUR, écrire une fonction `consommation : vehicule -> int -> float` telle que `consommation v n` renvoie le coût d'utilisation du véhicule `v` sur `n` kilomètres.

Exercice 3.8 Rappels :

$$\begin{array}{ll} \frac{dc}{dx} & = 0, \\ \frac{dx}{dx} & = 1, \\ \frac{d(u+v)}{dx} & = \frac{du}{dx} + \frac{dv}{dx}, \\ \frac{d(uv)}{dx} & = u\frac{dv}{dx} + v\frac{du}{dx}, \\ \frac{d(exp(u))}{dx} & = \frac{du}{dx} exp(u). \end{array}$$

On représente des expressions arithmétiques utilisant les opérations $+, -, \times, exp$ par le type suivant :

```
type expr =
          Var of string
        | Number of float
        | Plus of expr * expr
        | Minus of expr * expr
        | Mult of expr * expr
        | Exp of expr
```

Ainsi, une variable x est représentée par l'expression `OCaML Var "x"` et $3x^2 + 2x + 1$ par

```
Plus (Mult (Number 3., Mult (Var "x", Var "x")),
  Plus (Mult (Number 2., Var "x"), Number 1.))
```

1. Définir deux variables `vx` et `vy` contenant respectivement des variables x et y.
2. Définir la variable `e1` contenant l'expression $2x + 1$.
3. Définir la variable `e2` contenant l'expression $3x^2 + 2x + 1$.
4. Implémenter une fonction `derivee var expr` par une traduction directe des cinq règles ci-dessus, i.e. par un simple filtrage avec cinq cas distincts.
   Exemple d'utilisation :

```
utop[83]> vx;;
- : expr = Var "x"
utop[84]> e1;;
- : expr = Plus (Mult (Number 2., Var "x"), Number 1.)
utop[85]> derivee vx e1;;
- : expr =
Plus (Plus (Mult (Number 2., Number 1.), Mult (Number 0., Var "x")),
 Number 0.)
utop[86]> e2;;
```

```
- : expr =
Plus (Mult (Number 3., Mult (Var "x", Var "x")),
 Plus (Mult (Number 2., Var "x"), Number 1.))
utop[87]> derivee vx e2;;
- : expr =
Plus
 (Plus
   (Mult (Number 3.,
     Plus (Mult (Var "x", Number 1.), Mult (Number 1., Var "x"))),
    Mult (Number 0., Mult (Var "x", Var "x"))),
  Plus (Plus (Mult (Number 2., Number 1.), Mult (Number 0., Var "x")),
   Number 0.))
```

5. Écrire la fonction `derivee_n var expr n` qui retourne une expression correspondant à la dérivée n-ème de `expr`.
   Exemples :

```
utop[91]> derivee_n vx e2 2;;
- : expr =
Plus
 (Plus
   (Plus
     (Mult (Number 3.,
       Plus (Plus (Mult (Var "x", Number 0.), Mult (Number 1., Number 1.)),
        Plus (Mult (Number 1., Number 1.), Mult (Number 0., Var "x")))),
      Mult (Number 0.,
       Plus (Mult (Var "x", Number 1.), Mult (Number 1., Var "x")))),
    Plus
     (Mult (Number 0.,
       Plus (Mult (Var "x", Number 1.), Mult (Number 1., Var "x"))),
      Mult (Number 0., Mult (Var "x", Var "x")))),
  Plus
   (Plus (Plus (Mult (Number 2., Number 0.), Mult (Number 0., Number 1.)),
     Plus (Mult (Number 0., Number 1.), Mult (Number 0., Var "x"))),
    Number 0.))
utop[92]>
```

   Les expressions auraient bien besoin d'être simplifiées. Proposer des pistes pour simplifier les expressions.