TD 1 : Récursivité et Typage

Semaine du 11/09/2023

Pour ce TD, nous allons utiliser la console OCaml de Basthon comme outil. Vous pouvez par ailleurs utiliser la console native OCaml si vous avez installé OCaml sur votre ordinateur personnel.

Exercice 1. Calcul

1. Une fonction de signature `int -> int` qui corresponde à la fonction $f(n) = 2n + 1$.
2. Une fonction de signature `float * float -> float` qui renvoie la norme d'un vecteur passé en entrée par le couple de ses deux coordonnées.
3. Écrire une fonction OCaml de signature `int -> int` qui à n associe le nombre de diagonales d'un polygone convexe à n côtés.
4. Une fonction de signature `float*float -> float*float -> float*float` qui avec une entrée `(x1, y1) (x2,y2)` les coordonnées de deux points renvoie le centre de ces deux points.

Exercice 2. Filtrage

Proposer une implémentation en OCaml pour chacune des spécifications suivantes :

1. Une fonction de signature `int -> bool` qui détermine si une année est bissextile.
2. Une fonction de signature `'a -> 'a -> 'a` qui renvoie l'élément le plus grand parmi les deux arguments.
3. Une fonction de signature `bool -> bool -> bool` qui renvoie `true` si et seulement si un nombre pair de ses deux entrées est égal à `true`.
4. Une fonction de signature `bool -> bool-> bool -> bool` qui renvoie `true` si et seulement si un nombre pair de ses arguments est égal à `true`.

   On veillera à parenthéser correctement les conditions pour éviter des problèmes d'associativité.
5. Une fonction `clamp` de signature `float -> float -> float ->float` telle que, pour $a < b$, `clamp a b x` soit égal à $f_{a,b}(x)$ défini de la manière suivante :

$$f_{a,b}(x) = \begin{cases} b \text{ si } x > b, \\ a \text{ si } a > x, \\ x \text{ sinon.} \end{cases}$$

Exercice 3. Récursivité

Proposer une implémentation récursive OCaml pour chacune des signatures et spécifications suivantes :

1. Une fonction de signature `int -> int` qui calcule 2^n où n est son entrée.
2. Une fonction de signature `int -> int` qui calcule $n!$ où n est son entrée.
3. Une fonction de signature `int -> int` qui sur une entrée n calcule la somme des entiers jusqu'à n, $\sum_{k=0}^{n} k$.
4. Une fonction de signature `int -> float` qui calcule la suite définie par récurrence suivante :

$$v_0 = 2$$
$$\forall n \geqslant 0, v_{n+1} = \frac{1}{2}\left(v_n + \frac{2}{v_n}\right)$$

   Attention à ne calculer qu'une seul fois v_n.
5. Une fonction de signature `int -> int` qui renvoie le plus petit $n \geqslant 0$ tel que 2^n soit plus grand que son entrée.

Exercice 4. Fonction auxiliaire

Possiblement à l'aide d'une fonction auxilaire, proposer une implémentation pour les fonctions suivantes :

1. Une fonction de signature `int -> int -> int` qui calcule le PGCD de ses deux arguments avec l'algorithme d'Euclide.

   Attention : vous ne pouvez pas utiliser un nom pour une variable ou une fonction qui commence par une majuscule.
2. Une fonction de signature `int -> int` qui renvoie le nombre de diviseurs de son entrée.

Exercice 5. Calcul des termes de la suite de Fibonacci

> Attention !
>
> Selon votre implémentation, il est possible que l'exécution des questions suivantes entraîne un plantage de votre navigateur en raison du temps de calcul. Il peut être intéressant d'augmenter graduellement la valeur d'entrée.

La suite de Fibonacci peut être définie de manière récursive de la manière suivante :

$$u_0 = 0$$
$$u_1 = 1$$
$$\forall n \in \mathbb{N}\ u_{n+2} = u_{n+1} + u_n$$

Proposer une fonction récursive OCaml de signature `int -> int` qui calcule le n-ième terme de la suite de Fibonnacci. À l'aide de cette implémentation, calculer si possible les valeurs suivantes :

1. u_{10}
2. u_{20}
3. u_{30}

4. u_{40}
5. u_{45}

Indication : on pourra utiliser une fonction auxilaire qui retourne un couple de termes consécutifs de la suite de Fibonacci.

Exercice 6. Trouver, vérifier, et expliquer les signatures des fonctions suivantes :

1. ```
   let f x = x *. 2.
   ```
2. ```
   let f x = match x with
   | true -> 1.
   | false -> 0.
   ```
3. ```
   let f x y = x ** y
   ```
4. ```
   let f x y = match x || y with
   | true -> 1
   | false -> 0
   ```
5. ```
   let f x = 1.
   ```
6. ```
   let f x = let a = (x +. 1.) in 1
   ```
7. ```
   let f x g = g x
   ```
8. ```
   let f g h x = h (g x)
   ```
9. ```
   let f g x = g (g x)
   ```

Correction :

1. ```
   ('a -> 'b) -> ('b ->'c) -> 'a -> 'c
   ```

Exercice 7. Pour chacune de ces signatures, trouver une fonction qui ait la signature suivante :

1. ```
   'a -> int
   ```
2. ```
   (int -> 'a) -> 'a
   ```
3. ```
   (int -> 'a) -> int
   ```
4. ```
   (int -> int) -> int
   ```
5. ```
   'a -> ('a -> 'b) -> 'b
   ```
   ```

6. `('a -> 'b) -> 'a -> int`

7. `('a -> int) -> 'a -> float`

8. `('a -> 'a) -> 'a -> int`

Correction :

1. `let valeur_en_zero f = f 0`

2. ```
   let f g =
       let g 0
       in 0
   ```

3. `let valeur_entiere_en_zero f = (f 0) + 0`

4. `let application x f = f x`

Exercice 8. Suite de Syracuse

Soit $N \in \mathbb{N}$, on défini la suite de Syracuse associée à N par la formule de récurrence suivante :

$$u_0 = N$$
$$u_{n+1} = \begin{cases} \frac{u_n}{2} \text{ si } u_n \text{ est pair,} \\ 3u_n + 1 \text{ sinon.} \end{cases}$$

1. Proposer une fonction `syracuse` de signature `int -> int -> int` de sorte à ce que `syracuse debut n` calcule le n-ième terme de la suite de Syracuse associée à `debut`.
2. La conjecture de Collatz est l'hypothèse selon laquelle, pour tout entier strictement positif, la suite de Syracuse associée atteint 1 à un moment.

   Pour tout entier N strictement positif, on nomme temps de vol associée à N le plus petit indice n tel que la suite associée soit égale à 1.

   Proposer une fonction `temps_de_vol` de signature `int ->int` telle que `temps_de_vol debut` renvoie le temps de vol en partant de `debut`.

Exercice 9. Structures des appels récursifs

On considère la fonction suivante :

```
let rec f x = match x with
| 0 -> 0
| _ -> x*x + f (x - 1)
```

1. Que fait la fonction `f` ?
```

2. Pour chaque appel de `f` lors du calcul de `f 4`, donner les arguments avec laquelle la fonction est appelée, ainsi que sa valeur de retour.

   On considère désormais la fonction suivante :

```
let g a b =
    let rec aux a b = match b with
    | 0 -> a
    | _ -> aux b (a mod b)
    in aux (max a b) (min a b)
```

3. Que fait la fonction `g` ?
4. Pour chaque appel de `aux` lors du calcul de `g 23 17`, donner les arguments avec laquelle la fonction est appelée, ainsi que sa valeur de retour.
5. On dit qu'une fonction est **récursive terminale** quand la dernière chose faite dans la fonction est l'appel récursif, et donc que le résultat du dernier appel récursif est le même que le premier appel.

   Proposer une implémentation de `f` qui soit récursive terminale.

Nous aurons l'occasion de voir pourquoi la récursivité terminale est importante en OCaml.

Exercice 10. Récursivité Croisée

On se donne les fonctions définies récursivement par les formules suivantes :

$$F(0) = 1$$
$$M(0) = 0$$
$$\forall n > 0\ F(n) = n - M(F(n-1))$$
$$\forall n > 0\ M(n) = n - F(M(n-1))$$

1. Montrer par récurrence que, pour tout $n \geqslant 0$, $F(n+1) - F(n) \in \{0,1\}$ et $M(n+1) - M(n) \in \{0,1\}$.
2. En déduire que $F(n)$ et $M(n)$ sont définies pour tout n.
3. Proposer une implémentation récursive mutuelle de F et M.

   Attention, il n'est pas possible d'utiliser de nom qui commence par une majuscule pour une fonction ou une variable en OCaml.
4. Trouver les 10 premiers entiers $n > 0$ tels que $M(n-1) \neq F(n-1)$.

Exercice 11.

1. Proposer une fonction OCaml `suivant` de signature `(int -> bool) -> int -> int` telle que `suivant f n` renvoie le premier entier `k` supérieur ou égal à `n` tel que `f k` soit égal à `true`.
2. En déduire une fonction OCaml `enieme` de signature `(int -> bool) -> int` telle que `enieme f n` renvoie le n-ième entier naturel qui vérifie `f`.
3. En déduire une fonction pour déterminer le n-ième nombre qui vérifie la propriété pour chacune des propriétés suivantes :

   (a) `n` est premier ;
   (b) `n` est parfait ;
   (c) `n` est un nombre d'Erdős–Nicolas

Exercice 12. Rondes suisses

Exercice 13. Opérateur infixe pour la composition de fonctions

Il est possible en OCaml de définir des opérateurs infixes, c'est-à-dire, des fonctions qui se placent entre leur deux premiers arguments.

```
let (>=) f g x = g (f x)
```

De cette manière, pour toute fonction `f` et `g`, on a `(f >= g) x` qui est équivalent à `g (f x)`.
On se donne quelques exemples de fonctions :

```
let f x = x + 1
let g x = 2 * x
let h x = -x
```

1. Justifier la signature de l'opérateur `>=`
2. À quelle fonction correspond `(f >= g) >= h` ? À quelle fonction correspond `f>= (g>= h)` ?
3. Écriver une fonction OCaml avec l'opérateur `>=` ainsi que les fonctions `f`, `g` et `h` qui corresponde à la fonction $s(x) = 2(-x+1)$
4. Montrer que pour toute fonction `f1`, `f2` et `f3`, `(f1 >= f2) >= f3` et `f1 >= (f2>= f3)` correspondent aux mêmes fonctions.

Il y a quelques restrictions à respecter pour nommer les opérateurs, mais il s'agit d'outils assez utiles pour simplifier certaines syntaxe en OCaml à l'échelle d'un projet.
Un autre exemple d'opérateur utilisé est l'application inverse :

```
let (|>) x f = f x
```

Cela permet d'écrire :

```
1 |> f |> g |> h
```

On évitera cependant d'avoir recours à ce genre d'opérateurs en DS, à moins qu'ils ne soient définis dans le sujet. La définition d'opérateur infixe n'est PAS au programme.

Exercice 14. Une autre manière de faire de la récursion

On se donne la fonction suivante :

```
let rec combinateur f x = f (combinateur f) x
```

1. Justifier la signature de la fonction `combinateur`.
2. Déterminer l'effet de la fonction `f` définie de la manière suivante :

```
let g c n = match n with
| 0 -> 1
| n -> n * c (n - 1)

let f = combinateur g
```

3. Proposer une implémentation qui utilise la fonction `combinateur` pour répondre aux questions à un seul argument du troisième et quatrième exercice.

Correction :

1. ...