Programmation fonctionnelle

Fiche de TP 2

L3 Informatique 2021-2022

Fonctions sur les entiers

Rappel : un fichier `File.ml` contenant du code CAML peut être chargé dans le toplevel par la commande

```
# #use "File.ml";;
```

Exercice 1. (La fonction factorielle)

La factorielle $n!$ d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs[1] à n. Écrire la fonction `fact` à un paramètre entier `n` qui renvoie $n!$.

```
# fact;;
- : int -> int = <fun>
# fact 0, fact 1, fact 2, fact 3, fact 4, fact 5;;
- : int * int * int * int * int * int = (1, 1, 2, 6, 24, 120)
```

Note : pour écrire une fonction récursive `fct`, et donc d'avoir le droit d'appeler `fct` dans la définition de `fct`, le mot clé `rec` est à placer juste avant l'identificateur lors de la liaison globale :

```
let rec fct arg_1 ... arg_n =
    (* Corps de la fonction dans lequel des appels a fct sont possibles. *)
```

Exercice 2. (La suite de Fibonacci)

La suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence généralement par les termes 0 et 1 (ou par 1 et 1 selon les conventions) et ses premiers termes sont

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89.$$

Écrire la fonction `fib` à un paramètre entier `n` qui renvoie le n^e terme de la suite de Fibonacci (cette fonction n'est pas efficace, nous verrons plus loin comment faire mieux).

1. « Inférieur » signifie bien « inférieur ou égal ».

```
# fib;;
- : int -> int = <fun>
# fib 0, fib 1, fib 2, fib 3, fib 4, fib 5;;
- : int * int * int * int * int * int = (0, 1, 1, 2, 3, 5)
```

Exercice 3. (Plus grand diviseur commun)

Écrire une fonction `pgcd` qui, étant données deux entiers, renvoie leur plus grand diviseur commun. On se servira des relations

$$\mathrm{pgcd}(m,n) = \begin{cases} n & \text{si } m = 0, \\ \mathrm{pgcd}(n,m) & \text{si } m > n, \\ \mathrm{pgcd}(n \mod m, m) & \text{sinon} \end{cases}$$

```
# pgcd;;
- : pgcd : int -> int -> int = <fun>
# pgcd 1 1;;
- : int = 1
```

```
# pgcd 20 30;;
- : int = 10
# pgcd 77 34;;
- : int = 1
```

Exercice 4. (Fonction d'Ackermann-Péter)

La fonction d'Ackermann-Péter

$$A : \mathbb{N}^2 \to \mathbb{N}$$

est définie récursivement par

$$A(m,n) := \begin{cases} n+1 & \text{si } m = 0, \\ A(m-1,1) & \text{si } m \geqslant 1 \text{ et } n = 0, \\ A(m-1, A(m,n-1)) & \text{sinon.} \end{cases}$$

Écrire la fonction `ackermann` à deux paramètres entiers `m` et `n` qui calcule $A(m,n)$.

```
# ackermann;;
- : int -> int -> int = <fun>
# ackermann 0 1, ackermann 1 1, ackermann 2 1, ackermann 3 1;;
- : int * int * int * int * int = (2, 3, 5, 13)
# ackermann 0 2, ackermann 1 2, ackermann 2 2, ackermann 3 2;;
- : int * int * int * int = (3, 4, 7, 29)
```

Exercice 5. (Coefficient binomiaux)

Les coefficients binomiaux, définis pour tout entier naturel n et tout entier naturel k inférieur ou égal à n donnent le nombre de sous-ensembles de k éléments d'un ensemble de n éléments. On les note $\binom{n}{k}$. Ces nombres sont sujets à la formule de Pascal

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}.$$

En utilisant la formule de Pascal, écrire la fonction `binom` à deux paramètres entiers `n` et `k` qui renvoie $\binom{n}{k}$. On exploitera le fait que $\binom{n}{0} = 1$ pour tout entier naturel n et que $\binom{n}{k} = 0$ si $n < k$.

```
# binom;;
- : binom : int -> int -> int = <fun>
# binom 0 0;;
- : int = 1
# binom 1 0, binom 1 1;;
- : int * int = (1, 1)
# binom 2 0, binom 2 1, binom 2 2;;
- : int * int * int = (1, 2, 1)
# binom 3 0, binom 3 1, binom 3 2, binom 3 3;;
- : int * int * int * int = (1, 3, 3, 1)
# binom 4 0, binom 4 1, binom 4 2, binom 4 3, binom 4 4;;
- : int * int * int * int * int = (1, 4, 6, 4, 1)
```

Remarque : on peut vérifier à l'aide de la fonction factorielle que

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Écrire, **en une seule (assez grosse) expression** un moyen de vérifier que la version précédemment écrite pour le calcul des coefficients binomiaux et la formule ci-dessus donnent les mêmes valeurs pour tous $0 \leqslant k \leqslant n \leqslant 3$.

Exercice 6. (Fonctions mutuellement récursives)

Écrire les fonctions récursives `is_even` et `is_odd` sans utiliser le modulo et **avec comme seule information que `0` est pair**. La fonction `is_even` (resp. `is_odd`) renvoie `true` si son argument entier est pair (resp. impair). Comme le titre de l'exercice l'indique, les deux fonctions doivent s'appeler l'une l'autre. Il faudra se baser sur le fait qu'un nombre n est pair si et seulement si $n = 0$ ou $n - 1$ est impair, et qu'un nombre n est impair si et seulement si $n \neq 0$ et $n - 1$ est pair.

```
# is_even;;
- : int -> bool = <fun>
# is_odd;;
- : int -> bool = <fun>
# is_even 0, is_even 1, is_even 2, is_even 3, is_even 4, is_even 5;;
- : bool * bool * bool * bool * bool * bool =
(true, false, true, false, true, false)
# is_odd 0, is_odd 1, is_odd 2, is_odd 3, is_odd 4, is_odd 5;;
- : bool * bool * bool * bool * bool * bool =
(false, true, false, true, false, true)
```

Pour rappel, pour définir deux fonctions mutuellement récursives `fct1` et `fct2`, le mot-clé `and` est nécessaire :

```
let rec fct1 a1 ... an =
    (* Corps de fct1 ou fct1 et fct2 peuvent etre appelees. *)
and fct2 b1 ... bm =
    (* Corps de fct2 ou fct1 et fct2 peuvent etre appelees. *)
```

Ceci se généralise naturellement à un nombre quelconque de fonctions mutuellement récursives pourvu que le `and` soit utilisé conformément à cet exemple pour deux fonctions.

Exercice 7. (Récursivité terminale 1/2)

Une fonction `fct` est récursive terminale si chaque appel récursif qu'elle effectue est la dernière expression à être évaluée. Cette instruction est alors nécessairement « pure », c'est-à-dire qu'elle consiste en un simple appel à la fonction, et jamais en un calcul faisant intervenir des appels récursifs (comme c'est le cas de la fonction `fib` de l'exercice 2) ou une composition de fonctions (comme c'est le cas de la fonction `ackermann` de l'exercice 4).

La récursivité terminale économise de l'espace mémoire car aucun état (sauf l'adresse de la fonction appelante) n'a besoin d'être sauvé sur la pile d'exécution. Cela signifie également que le programmeur n'a pas à craindre l'épuisement de l'espace de pile ou du tas pour des récursions très profondes[2].

Écrire une fonction `fact'` à un paramètre entier `n` qui calcule $n!$ en faisant appel à une fonction récursive terminale (à deux paramètres). Indice : le 1er paramètre est le rang `n` et le 2^{e} est un accumulateur qui contient la valeur associée au rang précédent.

```
# fact, fact';;
- : (int -> int) * (int -> int) = (<fun>, <fun>)
# fact 0 = fact' 0, fact 1 = fact' 1, fact 2 = fact' 2, fact 10 = fact' 10;;
- : bool * bool * bool * bool = (true, true, true, true)
```

Exercice 8. (Récursivité terminale 2/2)

Écrire une version récursive terminale `fib'` de la fonction `fib` à un paramètre entier `n` qui renvoie le n^{e} terme de la suite de Fibonacci. Indice : la fonction `fib'` fait appel à une fonction récursive terminale à trois paramètres. Le 1er est le rang `n`, le 2^{e} est un accumulateur qui contient la valeur au rang `n - 2` et le 3^{e} est un autre accumulateur qui contient la valeur au rang `n - 1`.

```
# fib, fib';;
- : (int -> int) * (int -> int) = (<fun>, <fun>)
# fib 0 = fib' 0, fib 1 = fib' 1, fib 2 = fib' 2, fib 10 = fib' 10;;
- : bool * bool * bool * bool = (true, true, true, true)
```

2. Ces notions seront détaillées en CM.

Exercice 9. (Conjecture de Syracuse)

On appelle suite de Syracuse une suite d'entiers naturels définie de la manière suivante.

On part d'un nombre entier strictement plus grand que 0*; s'il est pair, on le divise par* 2*; s'il est impair, on le multiplie par* 3 *et on ajoute* 1*. En répétant l'opération, on obtient une suite d'entiers positifs dont chacun ne dépend que de son prédécesseur.*

Par exemple, à partir de 14, on construit la suite des nombres

$$14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, \ldots$$

C'est ce qu'on appelle la suite de Syracuse de 14. Après que le nombre 1 a été atteint, la suite $4, 2, 1$ se répète indéfiniment en un cycle de longueur 3 (appelé *cycle trivial*). La conjecture de Syracuse (encore appelée conjecture de Collatz, ou conjecture d'Ulam) est la conjecture[3] selon laquelle la suite de Syracuse de n'importe quel entier strictement positif atteint 1.

Écrire la fonction `syracuse` à un paramètre entier qui renvoie la longueur de la suite de Syracuse de cet entier pour obtenir 1.

```
# syracuse
- : syracuse : int -> int = <fun>
# syracuse 1;;
- : int = 0
# syracuse 14;;
- : int = 17
```

```
# syracuse 100;;
- : int = 25
# syracuse 1000;;
- : int = 111
# syracuse 10000;;
- : int = 29
```

Exercice 10. (Exponentielle)

1. Écrire la fonction `exp` à deux paramètres entiers `x` et `n` qui renvoie x^n.
2. Écrire la fonction `exp'` à deux paramètres entiers `x` et `n` qui renvoie x^n en étant récursive terminale.
3. Écrire la fonction `fast_exp` à deux paramètres entiers `x` et `n` qui calcule x^n plus rapidement que le fait `exp`. Indice : se baser sur le fait que $x^{2n} = (x^n)^2$ et $x^{2n+1} = (x^n)^2 x$.
4. Ré-écrire la fonction `exp` pour qu'elle renvoie x^n ainsi que le nombre d'appels récursifs effectués pour obtenir ce résultat (renvoyer pour cela un couple `(res, nb)` où `res` est le résultat de l'exponentiation et `nb` est le nombre d'appels récursifs ainsi réalisés).
5. Faire de même pour `fast_exp` et comparer.
6. Refaire les deux questions précédentes mais en comptant cette fois le nombre de multiplications réalisées. Trouver un lien entre le nombre de multiplications faites par `fast_exp` et l'écriture en base deux de n.

3. Une conjecture est une phrase que l'on pense vraie mais qui n'a pas été démontrée.

```
# exp;;
- : int -> int -> int = <fun>
# exp';;
- : int -> int -> int = <fun>
# fast_exp;;
- : int -> int -> int = <fun>
# exp' 2 10;;
- : int = 1024
# exp 2 10;;
- : int * int = (1024, 10)
# fast_exp 2 10;;
- : int * int = (1024, 4)
# fast_exp 3 21;;
- : int * int = (10460353203, 5)
```

Exercice 11. (Sommes doubles)

1. Écrire une fonction `sum1` à deux paramètres `n` et `m` qui renvoie la somme suivante en fonction de n et m :
$$\sum_{a=n}^{m} \sum_{b=a}^{m} b.$$
2. Écrire une fonction `sum2` à un paramètre `n` qui renvoie la somme suivante en fonction de n :
$$\sum_{0 \leqslant k < j \leqslant n} k/j$$

```
# sum1, sum2;;
- : (int -> int -> int) * (int -> float) = (<fun>, <fun>)
# sum1 1 3, sum1 0 10, sum1 5 10, sum1 5 3;;
- : int * int * int * int = (14, 440, 175, 0)
# sum2 0, sum2 3, sum2 5, sum2 10;;
- : float * float * float * float = (0., 1.5, 5., 22.5)
```