Récursivité et Listes

I. Récursivité

1. Principes de base

Une fonction *récursive* est une fonction qui peut être amenée à s'appeler elle-même. On signale la présence d'une fonction récursive en OCaml à l'aide du mot clé **rec**.

Un exemple, déjà vu plus tôt, est la fonction factorielle, à la version récursive est la traduction directe de la définition mathématique par récurrence :

```
let rec fact n =
    if n <= 1 then n
    else
        n * fact (n - 1)
;;
```

Méthodologie Lors de l'écriture d'une fonction récursive, on distingue en général deux types de branches d'éxecution :

⋆ les branches **cas de base** qui ne comporte pas d'appel récursif ;
⋆ les branches avec **appels récursifs**.

Pour ces dernières, si l'on veut que l'appel de la fonction termine, on s'assurera que l'appel récursif se fait avec des arguments « plus petits » (nous préciserons cela dans le chapitre suivant).
Pour développer l'intuition, on pourra considérer pour l'instant comme arguments des entiers naturels, un appel récursif avec des arguments strictement plus petits ne pouvant se répéter infiniment.

2. Quelques exemples

On commence par deux exemples, simples, portant sur les entiers.

2.1. Multiplication

On peut commencer par une version récursive très simple (mais peu efficace) de la multiplication entre entiers naturels vue comme addition itérée.

```
let rec multiplication_naive x y =
    if x = 0 then 0
    else
        y + multiplication_naive (x - 1) y
```

```
;;
```

On peut faire mieux avec la *multiplication russe* :

```
let rec multiplication_russe x y =
    if x = 0 then 0
    else
        let x' = x / 2
        and y' = 2 * y in
        let prod = multiplication_russe x' y' in
        if x mod 2 = 1
        then
            prod + y
        else
            prod
;;
```

2.2. Exponentiation

```
let rec exponentiation_rapide a n =
    if n = 0 then 1
    else
        let r = exponentiation_rapide (a * a) (n / 2) in
        if n mod 2 = 1 then a * r else r
;;
```

On pourra noter qu'il s'agit du même algorithme que la multiplication russe (mais appliqué à un autre groupe).

Exercice 1 On note $a \wedge b$ le *plus grand commun diviseur* de deux entiers naturels a et b.

1. Pour $a > 0$, que vaut $a \wedge 0$?
2. Justifier que, si $b \neq 0$, on a $a \wedge b = b \wedge (a - b)$ et même $a \wedge b = b \wedge (a \bmod b)$ où mod désigne (comme c'est souvent l'usage en informatique) le reste de la division euclidienne.
3. En déduire une implémentation récursive du calcul du PGCD.

3. Fonctionnement, pile d'appel

On peut retenir, pour comprendre le fonctionnement d'appels récursifs (mais, en fait, de tout appel de fonction), que lorsqu'une fonction en appelle une autre, elle...

1. interrompt ce qu'elle fait,

2. note toutes les informations pour reprendre plus tard (où elle en est exactement, quelles sont les valeurs des différents arguments),
3. stocke ses informations dans la *pile d'appel*,
4. fait l'appel de fonctions proprement dit,
5. récupère les informations stockées le plus récemment dans la pile d'appel,
6. reprends où elle en était.

Ce déroulement ne diffère pas pour des appels récursifs.

Pour comprendre un peu comment se déroulent les appels récursifs, on peut utiliser la fonctionnalité de *tracage* d'OCaml (les indentations ont été ajoutées à la main).

```
# #trace fact ;;
# fact 5 ;;
fact <-- 5
  fact <-- 4
    fact <-- 3
      fact <-- 2
        fact <-- 1
        fact --> 1
      fact --> 2
    fact --> 6
  fact --> 24
fact --> 120
- : int = 120
```

4. Récursivité terminale

On peut parfois optimiser les appels récursifs, en omettant les phases 5 et 6 d'un appel de fonction. Cela est possible si le résultat de l'appel récursif est directement utilisé comme résultat de l'appel de la fonction courante.

Exemple Considérons une fonction qui, étant donné une fonction f, un entier n et un élément x, calcule l'itéré n-ème de f appliqué à x. Une première version est la suivante :

```
let rec itere f n x =
    if n = 0 then x
    else f (itere f (n - 1) x)
;;
```

Cependant, après avoir exécuté l'appel `itere f (n - 1) x`, il faut encore effectuer un post-traitement qui consiste ici à appliquer `f` une nouvelle fois. On

peut l'éviter en modifiant la fonction ainsi :

```
let rec itere2 f n x =
    if n = 0 then x
    else itere f (n - 1) (f x)
;;
```

Sans entrer dans des détails trop techniques, notons qu'une méthode usuelle pour obtenir une fonction terminale récursive est d'utiliser une fonction auxilliaire qui comporte un argument supplémentaire correspondant au résultat en cours de construction.

Exemple Une version terminale récursive de la fonction factorielle est :

```
let fact2 n =
    let rec aux n r =
        (* r est le résultat en cours de construction *)
        if n = 0 then r
        else
            aux (n - 1) (n * r)
    in
    aux n 1
;;
```

Une fonction récursive terminale est très efficace, elle est optimisée par OCaml et se comporte comme une boucle **while**. La fonction précédente pourrait se réécrire (avec des références) :

```
let fact2bis n =
    let r = ref 1
    and i = ref n in
    while !i > 0 do
        r := !r * !i ;
        i := !i - 1
    done ;
    !r
;;
```

II. Listes

1. Types somme

1.1. Déclaration

Un *type somme* est défini par une liste de *constructeurs* (qui commençnt par une majuscule, dont l'usage est réglementé en OCaml), énumérés en utilisant « | »

comme séparateurs. Par exemple,

```
type direction = Nord | Sud | Est | Ouest
```

On peut ajouter à ces constructeurs des données supplémentaires à l'aide du mot-clé **of**. Par exemple,

```
type infinint = Fin of int | Inf
type int_or_string = Int of int | String of string
type forme =
| Carre of float
| Rectangle of float * float
| Cercle of float
```

Le type que l'on définit peut dépendre d'autres types en paramètres. Voici par exemple la définition du type `option` qui est défini en OCaml :

```
type 'a option = Some of 'a | None
```

1.2. Utilisation

Il est très simple de définir un élément de type somme.

```
let a = Nord ;;
let b = Fin 12 ;;
let c = Rectangle (1., 2.) ;;
let d = Some Inf ;;
```

Pour savoir à quel type de constructeur on a affaire, et pour récupérer les éventuelles données supplémentaires, on utilise la construction **match ... with**.

```
let surface f =
    match f with
    | Carre c -> c *. c
    | Rectangle (h, l) -> h *. l
    | Cercle r -> 3.1415926535 *. r *. r
```

On peut utiliser « _ » comme jocker.

```
let est_carre f =
    match f with
    | Carre _ -> true
    | _ -> false
```

Exercice 2 Écrire une fonction

```
addition : infinint -> infinit -> infinit.
```

2. Listes en OCaml

On pourrait définir le type *liste* de la façon suivante : une liste, c'est soit la liste vide, soit une liste non vide constituée d'un élément et d'un liste. Cela se traduit par le type somme suivant :

```
type 'a liste = Vide | Cons of 'a * 'a liste
```

En fait, il s'agit d'un type tellement commun en OCaml (et, plus généralement, en programmation fonctionnelle) que l'on a des notations spéciales :

- ⋆ la liste vide est notée `[]` ;
- ⋆ sinon, le *cons* d'un élément e suivi d'une liste l est noté e`::`l.

De plus, on peut noter une liste dans son intégralité entre crochets, les différents éléments étant séparés par des `;`. Ainsi, `[1; 2; 3]` correspond à la liste que l'on peut aussi noter `1 :: 2 :: 3 :: []` (qui correspond à `1 :: (2 :: (3 :: []))`, on associe à droite).

Voici un exemple de fonction sur une liste :

```
let rec longueur l =
    match l with
    | [] -> 0
    | a :: l' -> 1 + longueur l'
;;
```

Elle est déjà définie, comme `List.length`, mais sa structure est typique des fonctions sur les listes basées sur un unique parcours.

Voici un exemple de fonction qui renvoit le deuxième élément d'une liste, et génère une erreur s'il n'est pas défini.

```
let second l =
    match l with
    | _ :: a :: _ -> a
    | _ -> failwith "second"
;;
```

Exercice 3 Écrire les fonctions suivantes :

1. `est_present: 'a -> 'a list -> bool`
2. `ieme: 'a list -> int -> 'a option`
3. `maximum: 'a list -> 'a`
4. `concat: 'a list -> 'a list -> 'a list`

5. `miroir: 'a list -> 'a list`