Types récursifs (ou inductifs)

La récursivité est utilisable dans la définition des types. Ceci permet en particulier de construire des types infinis.
On peut par exemple représenter les listes d'entiers[4].

```
# type intlist = NI | CI of int * intlist;; (* NI: liste d
type intlist = NI | CI of int * intlist
# CI(1, CI(2, CI(3, NI)));;
- : intlist = CI (1, CI (2, CI (3, NI)))
# NI;;
- : intlist = NI
# type 'a truclist = NT | CT of 'a * 'a truclist;;
type 'a truclist = NT | CT of 'a * 'a truclist
# NT;;
- : 'a truclist = NT
# CT('a', CT('b', CT('c', NT)));;
- : char truclist = CT ('a', CT ('b', CT ('c', NT)))
```

[4] Définition en fait inutile, puis qu'il existe un type prédéfini pour les listes que nous verrons page 82

Listes

Comment définir un type liste générique (liste d'éléments appartenant tous à un même type non fixé)?

```
type 'a mylist = Nil | C of 'a * 'a mylist
```

Exemples de fonctions utilisant ce type

- `length`
- `make_list`
- `concat`

Récursivité terminale

Un appel récursif est dit terminal si il est retourné directement par la fonction, c'est-à-dire qu'aucune opération n'est faite avec l'appel récursif mise à part le retour.
Une fonction récursive est dite récursive terminale[5] si tous ses appels récursifs sont terminaux.

La fonction récursive `fact` définie ci-dessous n'est pas récursive terminale car la multiplication par `n` est effectuée entre l'appel récursif `fact (n - 1)` et le retour de la fonction.

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1)
```

[5] tail recursive en anglais

Récursivité terminale

fonction pas récursive terminale
⇒ appels empilés (pile d'exécution)
lors de l'appel à `fact 4` seront empilés les appels: `fact 4`,
`fact 3`, `fact 2`, `fact 1`, `fact 0`.
Le dernier appel `fact 0` retourne `1`,
`fact 1` retourne `1 * 1 = 1`,
`fact 2` retourne `2 * 1 = 2`,
`fact 3` retourne `3 * 2 = 6`,
`fact 4` retourne `4 * 6 = 24`.
empilement d'appels ⇒ débordement de la pile (Stack overflow)
injustifié dans le cas d'un tel calcul qui dans un langage classique
se ferait avec une simple boucle.

```
def fact (n):
    p = 1
    for i in range(2, n + 1):
        p *= i
    return p
```

Récursivité terminale

avantage: pas nécessaire d'empiler les appels; l'appel récursif remplace l'appel précédent.
$\Rightarrow$ pas de débordement de pile

- pas toujours possible d'obtenir une fonction récursive terminale
- Souvent, passage d'une fonction non récursive terminale à une fonction récursive terminale par ajout d'un paramètre qui joue le rôle d'accumulateur et dans lequel on calcule la valeur à l'appel et non au retour de l'appel récursif.

- fonction auxiliaire `fact_aux n p` récursive terminale
- `fact` s'écrit en appelant `fact_aux n 1`, `1` étant l'élément neutre pour le produit.

```
let rec fact_aux n p =
  if n = 0 then p
  else fact_aux (n - 1) (n * p)

let fact n = fact_aux n 1
```

`fact_aux 4 1` remplacé par

`fact_aux 3 4` remplacé par

`fact_aux 2 12` remplacé par

`fact_aux 1 24` remplacé par

`fact_aux 0 24` retourne `24`.

`p` (resp. `n`) joue même rôle que `p` (resp. `i`):

```
def fact (n):
    p = 1
    for i in range(n, 0, -1):
        p *= i
    return p
```

fonction auxiliaire à l'intérieur de la fonction principale à l'aide d'un `let rec ... in ...` (sauf si fonction aux utile dans un autre contexte).

```
let fact n =
  let rec aux n p =
    if n = 0 then p
    else aux (n - 1) (n * p)
  in aux n 1
```

Exemples: `make_list_rt`, `length_rt`, `reverse_rt`

Type `'a list` prédéfini en `OCaML`

pas nécessaire de définir un type `'a mylist`
(comme vu précédemment)

type prédéfini `'a list` est fourni par le module `List`

listes homogènes (comme dans le cas du type `'a mylist`):
tous les éléments sont d'un même type.

fonctions de ce module seront accessibles avec le préfixe `List.`
(utiliser la complétion pour voir toutes les fonctions du module)

Par exemple, `List.length` (la longueur d'une liste)

Constructeurs et accesseurs pour le type `list`

- constructeur de liste vide est `[]` au lieu de `Nil` pour le type `mylist` .
- `constructeur` permettant de rajouter un élément à une liste est `::` mais contrairement au constructeur `C` du type `mylist` , il s'utilise en notation infixe.

  Ainsi on écrit `e :: l` au lieu de `C(e,l)` dans le type `mylist` .

Les `accesseurs` associés à ce constructeur sont `List.hd` et `List.tl` pour récupérer respectivement la tête (head) `e` et la queue (tail) `l` de la liste `e :: l` .
Ces accesseurs sont peu utilisés du fait que l'on utilisera le plus souvent la construction `match` pour déconstruire une liste.

La fonction de calcul de la longueur d'une liste qui s'écrivait avec le type `mylist`

```
type 'a mylist = Nil | C of 'a * 'a mylist
let rec mylist_length l =
    match l with
      Nil -> 0
    | C(_, t) -> 1 + mylist_length t
```

s'écrit comme suit avec le type prédéfini et la construction `match`

```
let rec list_length l =
    match l with
      [] -> 0
    | _ :: t -> 1 + list_length t
```

ou encore sans utiliser `match`

```
 let rec list_length l =
  if l = [] then 0
  else 1 + list_length (List.tl l)
```

Format externe des listes

La notation `e1 :: e2 :: e3 ... :: en :: []` n'étant pas très agréable à lire, `OCaML` utilise un *format externe* pour écrire et lire des listes: `[e1; e2; ...; en]` est le format sous lequel `OCaML` affiche une liste et un format que l'utilisateur peut utiliser pour entrer une liste.

Exemples:

```
# [1; 2; 3];;
- : int list = [1; 2; 3]
# [1.; 2.; 3.];;
- : float list = [1.; 2.; 3.]
# List.hd [1; 2; 3];;
- : int = 1
# List.tl [1; 2; 3];;
- : int list = [2; 3]
# 3 * 3 :: [1; 2; 3];;
- : int list = [9; 1; 2; 3]
# let x = 4;;
val x : int = 4
```

Concaténation de listes

La fonction prédéfinie `List.append l1 l2` retourne une liste constituée des éléments de `l1` suivis des éléments de `l2`.

Exemples:

```
# List.append [1; 2; 3] [4; 5];;
- : int list = [1; 2; 3; 4; 5]
# List.append [1; 2; 3] [];;
- : int list = [1; 2; 3]
# List.append [] [3; 4; 5];;
- : int list = [3; 4; 5]
```

Il existe une version infixe de cette fonction: l'opérateur `@`.

Exemples:

```
# [1; 2; 3] @ [4; 5];;
- : int list = [1; 2; 3; 4; 5]
# [1; 2; 3] @ [];;
- : int list = [1; 2; 3]
# [] @ [3; 4; 5];;
- : int list = [3; 4; 5]
```

à utiliser avec parcimonie.
En effet, sa complexité est en $O(len(l_1))$ car il entraîne une copie de la liste `l1`.
Il peut servir ponctuellement à ajouter[6] un élément `e` en fin d'une liste `l` en utilistant l'expression `l @ [e]`
Par contre l'ajout d'un élément `e` en tête d'une liste `l` se fait en temps constant $O(1)$ grâce à l'expression: `e :: l`.
Le plus souvent, il est préférable de construire une liste à l'envers puis de la retourner en utilisant la fonction `List.rev l` (linéaire par rapport à la longueur de la liste).

[6] en fait, construire une nouvelle liste ayant les mêmes éléments que `l` et `e` à la fin

Exemples de fonctions simples sur les listes

- `List.length`
- `List.mem`
- `List.append`
- `List.rev`
- `List.sort`
- `List.filter`
- `List.map`
- `List.find`, `List.find_all`
- ...

Et il y en a d'autres:

- utiliser `List.` *complétion* pour voir leur nom
- taper leur nom pour voir leur type