Listes

18 septembre 2023

Plan

Correction TP1

Listes

Administratif

- DM1 à rendre pour le 02/10 ;
- début des colles en TP d'informatique dès ce soir 18h25-19h20 : G1 avec M. Dziki, G2 avec M. Trucchi ;
- il me manque encore quelques e-mails d'information.

Correction TP1

La base d'OCaml, l'application

En OCaml, on évalue une fonction en mettant les arguments les uns à la suite des autres séparés par des espaces.

```
1 max 1 0
```

```
1 max (1, 0)
```

```
1 max (1 0)
```

Définition de fonctions

Pour définir une fonction, on doit mettre des espaces dans notre définition :

```
1 let f x = x + 1
```

Si on veut mettre plus d'arguments, on doit rajouter des arguments avant le signe égal séparés par des espaces :

```
1 let f x y = x + y
```

Variables locales à une fonction

Si on veut utiliser une variable locale dans une fonction, on doit le faire *dans* le corps de la fonction :

```
let x1, y1 = couple1 in
let x2, y2 = couple2 in
let f couple1 coupl2 =
    (x1 +. x2)/.2., (y1 +. y2)/.2.
```

Variables locales à une fonction

Si on veut utiliser une variable locale dans une fonction, on doit le faire *dans* le corps de la fonction :

```
let x1, y1 = couple1 in
let x2, y2 = couple2 in
let f couple1 coupl2 =
    (x1 +. x2)/.2., (y1 +. y2)/.2.
```

```
let f couple1 coupl2 =
    let x1, y1 = couple1 in
    let x2, y2 = couple2 in
    (x1 +. x2)/.2., (y1 +. y2)/.2.
```

Filtrage (1)

L'opération de base pour faire un test est le filtrage :

```
let f x = match x with
| 0 -> false
| _ -> true
```

Filtrage (2)

On peut filtrer un peu ce qu'on veut :

```
let f x = match x>0 with
| true -> x
| false -> 0
```

```
let f x y = match x, y with
| (0, 0) -> 2
| (0, _) -> 1
| (_, 0) -> 1
| _ -> 0
```

Attention pour les cas de filtrage

On ne peut pas se servir d'un nom de variable pour tester l'égalité dans un filtrage.

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

Attention pour les cas de filtrage

On ne peut pas se servir d'un nom de variable pour tester l'égalité dans un filtrage.

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

```
let rec f a b = match a with
| _ when a = b -> 0
| _ -> a + f (a + 1)
```

Filtrage imbriqués (1)

```
1  let f x y z = match x with
2  | true -> match y with
3      | true -> not z
4      | false -> z
5  | false -> match y with
6      | true -> z
7      | false -> not z
```

Filtrage imbriqués (1)

```
let f x y z = match x with
| true -> match y with
    | true -> not z
    | false -> z
| false -> match y with
    | true -> z
    | false -> not z
```

```
let f x y z = match x with
| true -> match y with
| true -> not z
| false -> z
| false -> match y with
| true -> z
| false -> not z
```

Filtrage imbriqué (2)

```
let f x y z = match x with
| true -> (match y with
| true -> not z
| false -> z
| false -> (match y with
| true -> z
| false -> not z))
```

Filtrage imbriqué (2)

```
let f x y z = match x with
| true -> (match y with
| true -> not z
| false -> z
| false -> (match y with
| true -> z
| false -> not z))
```

```
let f x y z = match x with
| true -> (match y with
    | true -> not z
    | false -> z)
| false -> (match y with
    | true -> z
    | false -> not z)
```

La récursivité : la seule manière de faire des boucles

Pour l'instant, la seule manière dont on dispose pour faire des boucles est d'utiliser une fonction récursive : c'est-à-dire une fonction qui s'appelle elle-même.
L'idée est de procéder comme par récurrence, on construit le résultat pour une entrée à partir d'entrées plus petites :

```
1 let rec puissance2 n = match n with
2 | 0 -> 1
3 | _ -> 2 * puissance2 (n-1)
```

Trouver une formule de récurrence

On note $u_n = \sum_{k=0}^{n} k$. On remarque que :

$$u_0 = 0$$

$$\forall n \geqslant 1, u_n = n + u_{n-1}$$

Par conséquent, on peut écrire :

```
let rec somme n = match n with
| 0 -> 1
| _ -> n + somme (n-1)
```

Fonctions auxiliaires (1)

```
1 let rec est_premier n = ...
```

Fonctions auxiliaires (1)

```
let rec est_premier n = ...
```

```
let rec est_premier n k = match k >= n with
| true -> true
| _ when n mod k = 0 -> false
| _ -> est_premier n (k+1)
```

Fonctions auxiliaires (1)

```
let rec est_premier n = ...
```

```
let rec est_premier n k = match k >= n with
| true -> true
| _ when n mod k = 0 -> false
| _ -> est_premier n (k+1)
```

```
est_premier 53 2
```

Fonctions auxiliaires (2)

```
1 let rec aux n k = match k >= n with
2 | true -> true
3 | _ when n mod k = 0 -> false
4 | _ -> aux n (k+1)
5
6 let est_premier n = aux n 2
```

Fonctions auxiliaires (2)

```
let rec aux n k = match k >= n with
| true -> true
| _ when n mod k = 0 -> false
| _ -> aux n (k+1)

let est_premier n = aux n 2
```

```
let est_premier n =
    let rec aux n k = match k>= n with
    | true -> true
    | _ when n mode k = 0 -> false
    | _ -> aux n (k+1)
    in
match x<= with
| true -> true
| _ -> aux n 2
```

Suite de Fibonacci (1)

$$u_0 = 0$$

$$u_1 = 1$$

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

Suite de Fibonacci (1)

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

```
1 let rec fibo n = match n with
2 |0 -> 0
3 |1 -> 1
4 |_ -> fibo (n-1) + fibo (n-2)
```

À n donné, quel ordre de grandeur en nombre d'appels à la fonction `fibo` pour calculer u_n ?

Suite de Fibonacci (2)

```
1 let rec fibo n = match n with
2 |0 -> 0
3 |1 -> 1
4 |_ -> fibo (n-1) + fibo (n-2)
```

On fait au moins u_n appels au deuxième cas : la complexité en nombre d'appels récursifs est au moins de l'ordre de u_n.

Suite de Fibonacci (2)

```
1 let rec fibo n = match n with
2 |0 -> 0
3 |1 -> 1
4 |_ -> fibo (n-1) + fibo (n-2)
```

On fait au moins u_n appels au deuxième cas : la complexité en nombre d'appels récursifs est au moins de l'ordre de u_n.
Avec un ordinateur qui 10^{12} opérations par secondes, il faut un temps plus long que l'âge estimé de l'univers pour calculer u_{150}.

Suite de Fibonacci (3)

Solution : utiliser une fonction auxiliaire.

```
let fibo n =
    let rec aux a b n = match n with
    | 0 -> a
    | _ -> aux b (a + b) (n-1)
in aux 0 1 n
```

Inférence de type en OCaml

OCaml fonctionne par unification de contraintes : grâce à la forme de l'expression, il obtient des contraintes qu'il essaye d'harmoniser.

Inférence de type

```
let f x = 1 + x
```

Inférence de type

```
let f x = 1 + x
```

```
let f x y = x + y
```

Inférence de type

```
let f x = 1 + x
```

```
let f x y = x + y
```

```
let f x = 1
```

Inférence de type

```
let f x = 1 + x
```

```
let f x y = x + y
```

```
let f x = 1
```

```
let f g x = g x
```

Listes

Listes en OCaml

En OCaml, les listes sont délimités par des crochets, et les éléments sont séparés par des point-virgules.

```
let l = [1 ; 2 ; 3]
```

```
int list
```

Tous les éléments d'une liste doivent être de même type.

Constructeurs pour la liste

La liste vide est polymorphe :

```
[]
```

```
'a list
```

On peut ajouter un élément en tête de liste avec l'opérateur :: :

```
0::[1; 2; 3]
```

```
[0; 1; 2; 3]
```

Attention : ce qui est à gauche de l'opérateur :: doit être un élément, et ce qui est à droite doit être une liste d'élément de même type.

Filtrage sur les listes

On peut filtrer une liste par la forme qu'elle prend :

```
let est_vide liste = match liste with
| [] -> true
| _ -> false
```

Filtrage récursif (1)

Grâce au constructeur :: on peut préciser la forme d'une liste qui n'est pas vide :

```
let rec longueur liste = match liste with
| [] -> 0
| p::q -> 1 + (longueur q)
```

Filtrage récursif (1)

Grâce au constructeur :: on peut préciser la forme d'une liste qui n'est pas vide :

```
let rec longueur liste = match liste with
| [] -> 0
| p::q -> 1 + (longueur q)
```

Exemple : trouver une fonction de type int list -> int qui fait la somme des éléments d'une liste d'entiers.

Filtrage récursif (1)

Grâce au constructeur :: on peut préciser la forme d'une liste qui n'est pas vide :

```
let rec longueur liste = match liste with
| [] -> 0
| p::q -> 1 + (longueur q)
```

Exemple : trouver une fonction de type int list -> int qui fait la somme des éléments d'une liste d'entiers.

```
let rec somme liste = match liste with
| [] -> 0
| p::q -> p + (longueur q)
```

Filtrage récursif (2)

Que fait la fonction suivante ?

```
let rec f x liste = match liste with
| [] -> false
| p::q when p = x -> true
| _::q -> f x q
```

Filtrage récursif (2)

Que fait la fonction suivante ?

```
let rec f x liste = match liste with
| [] -> false
| p::q when p = x -> true
| _::q -> f x q
```

Comment modifier la fonction pour vérifier si *tous* les éléments de la liste sont égaux à l'entrée ?

Filtrage récursif (2)

Que fait la fonction suivante ?

```
let rec f x liste = match liste with
| [] -> false
| p::q when p = x -> true
| _::q -> f x q
```

Comment modifier la fonction pour vérifier si *tous* les éléments de la liste sont égaux à l'entrée ?

```
let rec tous_egaux x liste = match liste with
| [] -> true
| p::q when p = x -> tous_egaux x q
| _ -> false
```

Filtrage plus complexe

Pour le filtrage, on peut bien sûr utiliser des filtrages plus complexes :

```
let rec est_croissant liste = match liste with
| [] -> true
| [_] -> true
| p::q::r when p<q -> est_croissant (q::r)
| _ -> false
```

Construction récursive de liste (1)

Le constructeur :: permet de construire une liste dans la sortie d'une fonction récursive :

```
let rec liste_de_zero n = match n with
| 0 ->[]
| _ -> 0::liste_de_zero (n-1)
```

Construction récursive de liste (2)

Que fait la fonction suivante ?

```
let rec f n = match n with
| 0 -> []
| _ -> n::f (n-1)
```

Construction récursive de liste (2)

Que fait la fonction suivante ?

```
let rec f n = match n with
| 0 -> []
| _ -> n::f (n-1)
```

On ne peut ajouter d'élément qu'en début de liste avec le constructeur `::`.
Comment peut on faire pour obtenir les éléments dans l'autre sens ?

Construction récursive de liste (2)

Que fait la fonction suivante ?

```
let rec f n = match n with
| 0 -> []
| _ -> n::f (n-1)
```

On ne peut ajouter d'élément qu'en début de liste avec le constructeur `::`.
Comment peut on faire pour obtenir les éléments dans l'autre sens ?

```
let range n =
    let rec aux k n = match k with
    | _ when k>n -> []
    | _ -> k :: aux (k+1) n
    in aux 1 n
```

Concaténation

Proposer une fonction de type `'a list -> 'a list -> 'a list` qui concatène deux listes passées en arguments.

Concaténation

Proposer une fonction de type `'a list -> 'a list -> 'a list` qui concatène deux listes passées en arguments.

```
let concat l1 l2 = match l1 with
| [] -> l2
| p::q -> p::(concat q l2)
```

Combien d'appels fait-on à la fonction ?

Opérateur de concaténation

OCaml nous fourni un opérateur pour concaténer deux listes : `@`. Cependant, sa complexité dépend de la taille de la première entrée et son usage doit donc être raisonnable.

```
let rec inverser liste = match liste with
| [] -> []
| p::q -> (inverser q) @ [p]
```

Combien de fois doit-on parcourir les éléments de la liste pour calculer l'inverse avec cette fonction ?

Inversion de liste grâce à un accumulateur

Pour inverser une liste, on peut utiliser une fonction qui dispose d'un **accumulateur**, c'est-à-dire d'un argument qui sauvegarde la réponse actuelle.

```
let inverse liste =
    let rec aux acc liste = match liste with
    | [] -> acc
    | p::q -> aux (p::acc) q
in aux [] liste
```

D'autres types d'éléments dans les listes

On peut mettre d'autre chose que des entiers dans les listes :

```
let l = [true; false; false]
```

```
bool list
```

D'autres types d'éléments dans les listes

On peut mettre d'autre chose que des entiers dans les listes :

```
let l = [true; false; false]
```

```
bool list
```

Comment programmer une fonction de type `bool list -> bool` qui renvoie `true` si et seulement si son entrée est une liste qui ne contient pas de `false` ?

D'autres types d'éléments dans les listes

On peut mettre d'autre chose que des entiers dans les listes :

```
let l = [true; false; false]
```

```
bool list
```

Comment programmer une fonction de type `bool list -> bool` qui renvoie `true` si et seulement si son entrée est une liste qui ne contient pas de `false` ?

```
let et_generalise = match liste with
| [] -> true
| false::_ -> false
| _::q -> et_generalise q
```

Listes de listes (1)

Rien ne nous empêche de faire des listes de listes :

```
let l = [[1; 2]; []; [3; 4; 5]]
```

```
int list list
```

Listes de listes (1)

Rien ne nous empêche de faire des listes de listes :

```
let l = [[1; 2]; []; [3; 4; 5]]
```

```
int list list
```

Comme d'habitude, tous les éléments doivent être de même type à l'intérieur d'une liste.

Listes de listes (2)

Et rien ne nous empêche de travailler avec des listes de listes :

```
let somme_listes l =
let rec aux1 l1 = match l1 with
| [] -> 0
| p::q -> aux2 p + aux1 q
and aux2 l2 = match l2 with
| [] -> 0
| p::q -> p + aux2 q
in aux1 l
```

Listes de listes (2)

Et rien ne nous empêche de travailler avec des listes de listes :

```
let somme_listes l =
let rec aux1 l1 = match l1 with
| [] -> 0
| p::q -> aux2 p + aux1 q
and aux2 l2 = match l2 with
| [] -> 0
| p::q -> p + aux2 q
in aux1 l
```

Malheureusement, on ne peut pas mélanger les profondeurs dans les listes, et on ne peut pas avoir des fonctions définies sur des profondeurs variables de cette manière.