Programmation Impérative en OCaml

25 septembre 2023

Plan

Remarques sur la semaine précédent

Compléments sur les listes

Programmation Impérative

Remarques sur la semaine précédent

Barre du filtrage

La barre du filtrage *doit* être présente pour un filtrage :

```
1 let f x = match x with
2 | 0 -> 0
3 | x -> x
```

Il ne faut pas mettre la barre verticale partout :

```
1 let f x =
2 | x + 1
```

Cas de Base

On cherche toujours à avoir le cas de base le plus général.

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

Cas de Base

On cherche toujours à avoir le cas de base le plus général.

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

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

Science du cas de Base

```
let min_liste liste = match liste with
| [] -> ...
| p::q -> min p (min_liste q)
```

Science du cas de Base

```
let min_liste liste = match liste with
| [] -> ...
| p::q -> min p (min_liste q)
```

```
let min_liste liste = match liste with
| [] -> max_int
| p::q -> min p (min_liste q)
```

Fonctions auxiliaire

On peut définir une fonction localement à l'intérieur d'une fonction en OCaml. On appelle cela une **fonction auxiliaire**.

```
let pgcd 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)
```

Rajouter des variables dans les arguments

L'un des intérêts des fonctions auxilaires est de pouvoir rajouter une variable sur laquelle on va itérer :

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

Ici, `k` est un argument supplémentaire que l'on modifie au fur et à mesure des appels.

Compléments sur les listes

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.

Trouver toutes les pairs d'une liste

Dans la version à jour du TD.

Programmation Impérative

Effet de bord

Définition 1 : Fonction à effet de bord

Une fonction est dite **à effet de bord** si elle modifie un état autre que son environnement local.

La plupart du temps, cela est une modification de la mémoire du programme, ou bien une interaction avec les entrée-sorties du programme.

Afficher quelque chose sur la sortie standard

La fonction `print_string` permet d'afficher une chaîne de caractère sur la sortie standard. Elle prend un argument la chaîne à afficher.

```
1 print_string "Bonjour !"
```

Type unit

Il existe un type spécial, le type `unit` qui indique qu'en expression ne renvoie aucune valeur. C'est le résultat de la plupart des fonctions qui n'ont que des effets de bord.

Expression () (1)

L'expression `()` ne fait rien et est de type `unit`.
Certaines fonctions ont le type `unit -> unit`, on a besoin de les appeler avec l'argument `()` pour les exécuter :

```
1 print_newline ()
```

Exprsession () (2)

On se donne la fonction suivante :

```
let f () = 1
```

Quelle est la différence entre les deux expressions suivantes :

```
f
```

```
f ()
```

Séquence

Étant donné que les expression peuvent avoir des effets de bord, on peut vouloir calculer des expressions les unes à la suite des autres. Le séparateur ; permet de séparer deux expressions qui seront calculées l'une après l'autre.

```
1 print_string "Bim" ; print_string "Bam"
```

Valeur des séquences

Lorsqu'une expression est une séquence, c'est le type et la valeur de la dernière sous-expression qui donne le type et la valeur de l'expression totale :

```
1 ; 1.0
```

```
- : float = 1.0
```

```
1.0 ; 1
```

Valeur des séquences

Lorsqu'une expression est une séquence, c'est le type et la valeur de la dernière sous-expression qui donne le type et la valeur de l'expression totale :

```
1 ; 1.0

- : float = 1.0
```

```
1 1.0 ; 1

- : int = 1
```

Valeur des séquences

Lorsqu'une expression est une séquence, c'est le type et la valeur de la dernière sous-expression qui donne le type et la valeur de l'expression totale :

```
1 ; 1.0
```

```
- : float = 1.0
```

```
1.0 ; 1
```

```
- : int = 1
```

En règle générale, OCaml affichera un avertissement si une expression à gauche d'un ; n'est pas de type unit.

Condition Si...Alors...Sinon...

En OCaml, comme dans la plupart des langages, on peut utiliser un élément conditionel.

```
1 if a = 0
2 then 1
3 else 0
```

Typage de la structure conditionnelle

Le type de sortie doit être le même quelque soit la sortie.

```
1 if a = 0
2 then 1
3 else 1.0
```

Cela implique que, la plupart du temps, on a besoin de mettre le `else`.

Utilisation dans une fonction

Naturellement, on peut utiliser le `if ... then ... else ...` dans une fonction :

```
1 let relu x = if x>0. then x else 0.
```

Utilisation dans une fonction

Naturellement, on peut utiliser le `if ... then ... else ...` dans une fonction :

```
let relu x = if x>0. then x else 0.
```

Proposer une fonction qui utilise `if ... then ... else ...` pour calculer la fonction max qui renvoie le maximum de ses deux arguments.

Utilisation dans une fonction

Naturellement, on peut utiliser le `if ... then ... else ...` dans une fonction :

```
let relu x = if x>0. then x else 0.
```

Proposer une fonction qui utilise `if ... then ... else ...` pour calculer la fonction max qui renvoie le maximum de ses deux arguments.

```
let max x y = if x>y then x else y
```

Condition et type unit

Il n'est pas nécessaire de mettre un `else` dans le cas d'une expression de type `unit`.

```
1 if a = 0
2 then print_string "a est nul."
```

(OCaml rajoute implicitement `else ()` après.)

Cas d'ambiguïté sur le else

Le else est associé avec le dernier if en cas de doute.

```
1 if 0 = 0 then if 0 = 1 then print_string "1" else
    print_string "2"
```

Cas d'ambiguïté sur le else

Le else est associé avec le dernier if en cas de doute.

```
1 if 0 = 0 then if 0 = 1 then print_string "1" else
    print_string "2"
```

Comme d'habitude, plus de parenthèses en cas de doute.

Variable mutable

Définition 2 : Variable Mutable

Une variable est dite **mutable** si elle peut être modifiée.

Référence

OCaml propose des variables mutables sous la forme des **références**. Il s'agit d'une case dans la mémoire à laquelle on peut accéder et la modifier.
Si on modifie la variable, elle est modifiée pour le reste du programme qui peut donc voir l'effet d'une fonction sans passer par sa variable de retour.

Syntaxe des Références

On peut définir des références en OCaml avec le mot-clef `ref` :

```
1 let a = ref 0
```

Syntaxe des Références

On peut définir des références en OCaml avec le mot-clef `ref` :

```
let a = ref 0
```

On peut modifier cette référence avec la syntaxe `...:=...` :

```
a:=1
```

Syntaxe des Références

On peut définir des références en OCaml avec le mot-clef `ref` :

```
let a = ref 0
```

On peut modifier cette référence avec la syntaxe `...:=...` :

```
a:=1
```

On peut accéder au contenu d'une référence avec l'opérateur `!` :

```
!a
```

Avec ces trois syntaxes, on peut utiliser les références pour faire des calculs :

```
let a = ref 1 in
a:= !a + 1 ;
a:= !a * !a ;
!a
```

Attention : il faut bien mettre le `!a` à la fin de la séquence pour que celle-ci soit de type `int` et ait bien une valeur autre que `unit`.

Boucle While

OCaml propose une manière de répéter des calculs avec la syntaxe `while ... do ... done` :

```
let a = ref 0 in
while !a <10 do
    print_int !a;
    print_newline ();
    a:= !a + 1
done
```

Boucle While

OCaml propose une manière de répéter des calculs avec la syntaxe `while ... do ... done` :

```
let a = ref 0 in
while !a <10 do
    print_int !a;
    print_newline ();
    a:= !a + 1
done
```

Proposer une expression pour afficher les multiples de 7 jusqu'à 70.

Boucle While

OCaml propose une manière de répéter des calculs avec la syntaxe `while ... do ... done` :

```
let a = ref 0 in
while !a <10 do
    print_int !a;
    print_newline ();
    a:= !a + 1
done
```

Proposer une expression pour afficher les multiples de 7 jusqu'à 70.

```
let a = ref 0 in
while !a <=70 do
    print_int !a;
    print_newline ();
    a:= !a + 7
done
```

Boucle For

De même, on peut utiliser une boucle `for ... to ... do .... done` quand on veut itérer sur une valeur.

```
1 for a = 0 to 10 do
2 print_int (a*7);
3 print_newline ();
4 done
```

Chaîne de caractère

Les chaînes de caractère sont délimités par des guillements double droits " en OCaml.

```
1 "Bonjour"
```

Nous verrons plus de détails sur les chaînes de caractère la semaine prochaine.

Afficher du texte

On peut afficher plusieurs types d'objets :

```
print_string "Boum"
```

```
print_int 23
```

```
print_float (5./2.)
```

```
print_newline ()
```

Fonction Printf.printf

La fonction `Printf.printf` permet de formatter des chaînes de caractères à afficher :

```
1 Printf.printf "Voici un entier %d dans une chaine de
      caracteres" 230
```

Fonction Printf.printf

La fonction `Printf.printf` permet de formatter des chaînes de caractères à afficher :

```
1 Printf.printf "Voici un entier %d dans une chaine de
      caracteres" 230
```

Il est possible d'afficher d'autres types :

```
1 Printf.printf "Voici un flottant %f dans une chaine
    de caracteres" 2.5
```

Fonction Printf.printf

La fonction `Printf.printf` permet de formatter des chaînes de caractères à afficher :

```
1 Printf.printf "Voici un entier %d dans une chaine de
      caracteres" 230
```

Il est possible d'afficher d'autres types :

```
1 Printf.printf "Voici un flottant %f dans une chaine
     de caracteres" 2.5
```

On peut afficher plusieurs variables :

```
1 Printf.printf "Voici un flottant %f et un entier %d"
      2.5 39
```

Syntaxe Printf.printf

d	Entier
f	Flottant
B	Booléen
s	Chaîne de caractère

La documentation [1] sur cette fonction permet d'avoir plus d'informations à ce sujet.

1. https://v2.ocaml.org/api/Printf.html

failwith

La fonction failwith de signature string -> 'a permet d'afficher un message d'erreur :

```
let diviser x y = match y with
| 0 -> failwith "Erreur : division par zero"
| _ -> x/y
```

failwith

La fonction `failwith` de signature `string -> 'a` permet d'afficher un message d'erreur :

```
let diviser x y = match y with
| 0 -> failwith "Erreur : division par zero"
| _ -> x/y
```

Pourquoi cette signature ?

Utilisation de failwith pour la programmation défensive

On peut utiliser `failwith` pour se protéger d'entrées non conformes aux spécifications de la fonction :

```
let clamp a b x =
    if a > b then failwith "a doit etre inferieur a b"
    else match a>x, b>x with
    | true, _ -> a
    | _, true -> x
    | _ -> b
```