

# cours5

December 11, 2019

## 1 Programmation fonctionnelle

### 1.1 Semaine 5 : Entrées-sorties et exceptions

Etienne Lozes - Université Nice Sophia Antipolis - 2019

---

```
[1]: let () = ()
```

---

### 1.2 Les exceptions

Vous savez qu'il peut y avoir des *erreurs* durant l'exécution du programme, une division par 0, un appel à `List.tl` sur la liste vide, etc. Nous avons vu aussi comment provoquer une erreur qui interrompt le calcul en affichant un message d'erreur avec la fonction `invalid_arg`. La fonction `failwith` est une variante d'`invalid_arg`.

```
[2]: let () =
  print_string "hello";
  print_newline ();
  failwith "exception!"; (* <- le calcul s'arrête ici *)
  print_string "world";
  print_newline()
```

File "[2]", line 4, characters 2-23:

```
4 |   failwith "exception!"; (* <- le calcul s'arrête ici *)
   |   ~~~~~
```

Warning 21: this statement never returns (or has an unsound type.)

hello

```
      Exception: Failure "exception!".
      Raised at file "stdlib.ml", line 29, characters 22-33
      Called from unknown location
      Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

D'autres exemples d'exceptions sont

- `Match_failure` : erreur levée lorsqu'une définition par cas est incomplète
- `Division_by_zero` : comme son nom l'indique...
- `Assert_failure` : erreur levée par un `assert`

```
[3]: match 2 with 0 -> true | 1 -> false (* pas de else ... *)
```

File "[3]", line 1, characters 0-35:

```
1 | match 2 with 0 -> true | 1 -> false (* pas de else ... *)
   ~~~~~
```

Warning 8: this pattern-matching is not exhaustive.

Here is an example of a case that is not matched:

```
2
```

File "[3]", line 1, characters 0-35:

```
1 | match 2 with 0 -> true | 1 -> false (* pas de else ... *)
   ~~~~~
```

Warning 8: this pattern-matching is not exhaustive.

Here is an example of a case that is not matched:

```
2
```

Exception: `Match_failure` ("[3]", 1, 0).

Called from file "toplevel/toploop.ml", line 208, characters 17-27

```
[4]: 1 / 0
```

Exception: `Division_by_zero`.

Raised by primitive operation at unknown location

Called from file "toplevel/toploop.ml", line 208, characters 17-27

```
[5]: assert (1 = 0 + 0)
```

Exception: `Assert_failure` ("[5]", 1, 0).

Called from file "toplevel/toploop.ml", line 208, characters 17-27

---

### 1.3 Rattraper une exception avec `try...with...`

Ces "erreurs" n'en sont pas toujours, elles peuvent être volontaires. Par exemple, dans une fonction, je peux demander un nombre à l'utilisateur et supposer qu'il fournira bien un nombre; si jamais il

tape autre chose, ma fonction fera une erreur, mais ce n'est pas grave, je vais la rattraper ailleurs. Je me concentre sur le cas *habituel* dans la fonction et je traite le cas *exceptionnel* ailleurs.

```
[6]: let affiche_inverse () =  
      let n = read_int () in print_int (1 / n); print_newline()
```

```
[6]: val affiche_inverse : unit -> unit = <fun>
```

Si j'appelle `affiche_inverse ()`, le toplevel me demande de saisir quelque chose.

- si je tape 1, le nombre 1 s'affiche
- si je tape 0, j'obtiens le message `Exception: Division_by_zero`
- si je tape toto, j'obtiens le message `Exception: Failure "int_of_string"`.

Créons maintenant une version "sans erreur" de la fonction `affiche_inverse`

```
[7]: let affiche_inverse_sans_erreur () =  
      try affiche_inverse () with  
      | Division_by_zero -> print_string "Vous avez tapé 0."  
      | Failure(msg) -> print_string "Vous n'avez pas tapé un entier."  
      | _ -> print_string "Une erreur inconnue s'est produite"
```

```
[7]: val affiche_inverse_sans_erreur : unit -> unit = <fun>
```

Si j'appelle maintenant `affiche_inverse_sans_erreur ()` et que je tape 0, le toplevel ne me signale plus d'exception. Normal, je l'ai rattrapée!

---

## 1.4 Lever une exception avec `raise`

Pour provoquer la levée d'une exception, la méthode la plus générale est d'utiliser la fonction `raise` avec pour argument l'exception que l'on souhaite lever. Les exceptions ont des noms qui commencent par une majuscule, par exemple `Failure`, `Invalid_argument`, `Not_found`, etc. On peut par exemple redéfinir `failwith` et `invalid_arg` comme suit.

```
[8]: let invalid_arg msg = raise (Invalid_argument msg)
```

```
[8]: val invalid_arg : string -> 'a = <fun>
```

```
[9]: let failwith msg = raise (Failure msg)
```

```
[9]: val failwith : string -> 'a = <fun>
```

```
[10]: let myassert test = if test then raise (Failure "assertion ... on line ...  
      ↪failed ")
```

```
[10]: val myassert : bool -> unit = <fun>
```

L'exception `Not_found` ne prend pas d'argument. On l'utilise typiquement quand une recherche échoue.

```
[11]: let index x liste = (* index x liste = index de la première apparition de x ↵
↳ dans liste *)
  let r = ref liste in
  let i = ref 0 in
  while (!r <> []) && (List.hd !r <> x) do
    r := List.tl !r;
    i := !i + 1
  done;
  if (!r <> []) then !i else raise Not_found
```

```
[11]: val index : 'a -> 'a list -> int = <fun>
```

```
[12]: let mem x liste = (* mem x liste = true si x apparait dans liste *)
  try
    ignore (index x liste); (* ignore:'a -> unit, nous évite un warning *)
    true
  with
  | Not_found -> false
```

```
[12]: val mem : 'a -> 'a list -> bool = <fun>
```

---

## 1.5 Quand rattraper une exception?

On n'est pas obligé de rattraper une exception à l'endroit où elle a été levée.

```
[13]: let f1 () = raise Not_found
let f2 () = print_string "hello," ; f1 ()
let f3 () = try f2 () with _ -> print_string " world!"; print_newline()
let () = f3 ()
```

```
[13]: val f1 : unit -> 'a = <fun>
```

```
[13]: val f2 : unit -> 'a = <fun>
```

```
[13]: val f3 : unit -> unit = <fun>
```

---

## 1.6 Jusqu'où remonte une exception?

L'exception remonte tous les appels de fonction jusqu'à trouver un bloc `try/with` qui la rattrape. Si le bloc `try/with` n'a pas prévu l'exception, elle passe au travers. Sinon, elle est rattrapée et elle n'est pas propagée plus loin.

```
[14]: try
      try
        try
          raise (Failure "boum")
        with
          | Not_found -> print_string "ce texte ne s'affichera pas"
        with
          | Failure(s) -> print_string s;print_string " rattrapé"; print_newline ()
      with
        | _ -> print_string "ce texte ne s'affichera pas"
```

hello, world!

```
[14]: - : unit = ()
```

---

## 1.7 Créer ses propres exceptions

`Failure`, `Invalid_argument`, ou encore `Not_found` sont des exceptions prédéfinies. On peut cependant déclarer ses propres exceptions et les utiliser ensuite.

```
[15]: exception Toto
```

boum rattrapé

```
[15]: exception Toto
```

```
[16]: Toto
```

```
[16]: - : exn = Toto
```

Le type `exn` des exceptions est un type énuméré un peu particulier. On peut l'étendre avec de nouveaux constructeurs d'exceptions, comme ici `Toto`. Ces constructeurs peuvent d'ailleurs prendre des arguments.

```
[17]: exception Myfailure of string
```

```
[17]: exception Myfailure of string
```

Examinons le type de `raise`

```
[18]: raise
```

```
[18]: - : exn -> 'a = <fun>
```

Le type de `raise` permet de lever une exception dans n'importe quel contexte tout en respectant les contraintes de typage: une expression comme `raise Toto` a le type polymorphe `'a`.

---

## 1.8 Utiliser une exception pour sortir d'une boucle

Réécrivons maintenant la fonction `index` avec un "return".

```
[19]: exception Return of int
let return x = raise (Return x)
```

```
[19]: exception Return of int
```

```
[19]: val return : int -> 'a = <fun>
```

```
[20]: let index x liste =
  let r = ref liste in
  let i = ref 0 in
  try
    while (!r <> []) do
      if List.hd !r = x then return !i;
      i := !i + 1;
      r := List.tl !r
    done;
    raise Not_found
  with
  | Return(n) -> n
```

```
[20]: val index : 'a -> 'a list -> int = <fun>
```

```
[21]: let _ = index 3 [0; 1; 3; 2]
```

```
[21]: - : int = 2
```

```
[22]: let _ = index 4 [0; 1; 3; 2]
```

```
Exception: Not_found.  
Raised at file "[20]", line 10, characters 10-19  
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

---

## 1.9 Les entrée-sorties

Tout programme qui s'exécute a une entrée standard (`stdin`) et une sortie standard (`stdout`).

Certaines instructions permettent de lire des données sur l'entrée standard et de les écrire sur la sortie standard, nous en avons déjà vu quelques unes:

- `print_int n` : écrit l'entier `n` sur la sortie standard
- `print_float x`, `print_string s`, `print_newline ()` écrivent aussi sur la sortie standard
- `read_int ()` : lit un entier sur l'entrée standard et renvoie sa valeur
- `read_float ()`, `read_line ()` : lisent aussi sur l'entrée standard

Écrivons un programme `salutation.ml` qui demande le nom de l'utilisateur et qui lui dit bonjour.

```
(* salutation.ml *)  
let salutation () = begin  
  print_string "Quel est ton nom, humain? ";  
  let nom = read_line () in  
  print_string ("bonjour, " ^ nom);  
  print_newline()  
end  
  
let () = salutation ()
```

Remarque: les entrées-sorties sont “bufferisées” en OCaml, si l'on omet le `print_newline()` on ne verra pas forcément apparaitre “bonjour ...” à la fin de la fonction `salutation`.

---

## 1.10 Comment tester un programme?

Pour tester mon programme, il y a trois étapes

1. je sauve le fichier (C-x C-s dans Emacs)
2. je le compile en tapant `ocamlc salutation.ml` depuis un terminal
3. je l'exécute en tapant `./a.out`

Je peux rediriger l'entrée standard et la sortie standard vers des fichiers ordinaires. Par exemple, si je tape

```
./a.out <nom.txt >msg.txt
```

et si `nom.txt` est un fichier que j'ai créé et qui contient "Etienne" sur la première ligne, j'obtiens "Bonjour, Etienne" dans le fichier `msg.txt`

Je peux aussi changer le nom du fichier exécutable généré par `ocamlc` en utilisant l'option `-o`.

```
ocamlc salutation.ml -o salut
./salut
```

---

## 1.11 L'exception `End_of_file`

Écrivons maintenant un programme qui calcule la somme des entiers qu'il lit sur l'entrée standard, et qui l'affiche sur la sortie standard. C'est l'exception `End_of_file` qui permettra de déterminer le moment auquel on a atteint la fin de l'entrée standard.

```
(* somme.ml *)
let calcul_somme () = begin
  let acc = ref 0 in
  try
    while true do (* boucle infinie! *)
      acc := !acc + read_int ()
    done
  with
    End_of_file -> print_int !acc
end

let () = calcul_somme ()
```

Pour tester ce programme, je crée un fichier `nombres.txt` qui contient des nombres, puis j'exécute `./a.out <nombres.txt`

Je peux aussi taper mes nombres dans l'entrée standard liée au terminal, et la fermer avec `Ctrl+D`.

---

## 1.12 Les canaux

Et si je veux spécifier dans mon programme le fichier `nombres.txt` qui m'intéresse? Il va me falloir créer un **canal d'entrée** (`in_channel`) pour ce fichier, puis utiliser des fonctions plus générales de lecture sur un canal.

Les principales étapes sont les suivantes:

1. `let ic = open_in "nombres.txt"` me permet d'ouvrir le fichier en lecture et d'obtenir le canal d'entrée `ic`
2. je lis les lignes de mon fichier une par une avec `input_line ic`
3. je ferme mon fichier avec `close_in ic`

```
[23]: (* fichier somme2.txt *)
```

```
let somme () =
```



```

let ic = open_in "nombres.txt" in (* <- OUVERTURE *)
let acc = ref 0 in

let rec iter () =
  let s = input_line ic in (* <- LECTURE *)
  let n = int_of_string s in
  acc := !acc + n;
  iter ()
in

try iter () with
| End_of_file ->
  begin
    print_int !acc;
    close_in ic (* <- FERMETURE *)
  end

(* pour lancer le programme: let () = somme () *)

```

[23]: val somme : unit -> unit = <fun>

Pour écrire dans un fichier, inversement, il me faut un `out_channel`. Les étapes sont sensiblement les mêmes:

1. ouvrir avec `open_out`
2. écrire avec `output_string`
3. fermer avec `close_out`

```

[24]: let comptine () =
  let oc = open_out "comptine.txt" in
  for i=1 to 10 do
    output_string oc (string_of_int i)
  done;
  close_out oc

```

[24]: val comptine : unit -> unit = <fun>

Notez que l'entrée standard et la sortie standard sont des canaux comme les autres

```
[25]: stdin
```

[25]: - : in\_channel = <abstr>

```
[26]: stdout
```

```
[26]: - : out_channel = <abstr>
```

---

### 1.13 Lecture et écriture simultanée

Écrivons maintenant un programme qui remplace chaque ligne de nombres prise dans un fichier “in.txt” par une ligne contenant la somme.

C’est l’occasion de découvrir quelques nouvelles fonctions bien utiles.

1. comment découper une chaîne de caractères?
2. comment appliquer une fonction à chaque élément d’une liste?
3. comment faire une somme généralisée d’une liste?

La fonction `String.split_on_char sep s` permet de découper une chaîne de caractères en une liste de chaînes de caractères en prenant `sep` comme caractère de séparation

```
[27]: String.split_on_char ' ' "21 823 87"
```

```
[27]: - : string list = ["21"; "823"; "87"]
```

La fonction `List.map f [x1; x2; ...; xn]` renvoie la liste `[f(x1); f(x2); ...; f(xn)]`

```
[28]: List.map int_of_string ["21"; "823"; "87"]
```

```
[28]: - : int list = [21; 823; 87]
```

La fonction `List.fold_left (++) x0 [x1; x2; ...; xn]` renvoie `((... (x0 ++ x1) ++ x2) ++ ... ++ xn)`

```
[29]: List.fold_left (+) 0 [21; 823; 87]
```

```
[29]: - : int = 931
```

Revenons à notre programme!

```
[30]: let somme () =  
  
    let ic, oc = open_in "alea.txt", open_out "somme.txt" in  
  
    let rec iter () =  
        let s = input_line ic in  
        let ls = String.split_on_char ' ' s in  
        let li = List.map int_of_string ls in  
        let n = List.fold_left (+) 0 li in
```

```

let s2 = (string_of_int n) ^ "\n" in
output_string oc s2;
iter ()
in

try iter () with
| End_of_file -> close_in ic; close_out oc
| exn -> (* on relance l'exception si ce n'est pas End_of_file *)
(close_in ic; close_out oc; raise exn)

```

[30]: val somme : unit -> unit = <fun>

---

### 1.14 L'opérateur |> d'application inversée

On peut rendre le programme précédent un peu plus stylé en utilisant l'opérateur |> d'application de fonction en ordre inversé.

$x \text{ |> } f \Leftrightarrow f \ x.$

[31]:

```

let somme () =
  let ic, oc = open_in "alea.txt", open_out "somme.txt" in

  try while true do

    input_line ic |> String.split_on_char ' ' |> List.map int_of_string
    |> List.fold_left (+) 0 |> string_of_int |> (fun s -> s^"\n") |>
    ↪output_string oc

  done with exn -> close_in ic; close_out oc; if exn<>End_of_file then raise exn

```

[31]: val somme : unit -> unit = <fun>

---

### 1.15 Les entrées-sorties formatées

Vous vous souvenez peut-être des chaînes de format de Python?

```

print("Il est {}:{}".format(15+2, 3+2))
# affiche "il est 17:05"

```

OCaml permet de faire quelque chose de similaire avec

```

Printf.printf "Il est %d:%d" (15+2) (3+2)

```

Le type de chacun des arguments qui suivent la chaîne de format est indiqué par un "marqueur" dans la chaîne de format

- %d pour un entier
- %s pour une chaîne de caractères
- %f pour un flottant
- %c pour un caractère
- ...

Le caractère retour à la ligne s'écrit `\n`, mais il ne force pas à vider le buffer. Pour vider le buffer, on peut utiliser `%!.`

```
[32]: let caractere_pi = "\xCF\x80" (* le codage sur 2 octets de en UTF8 *)
      let pi = 4. *. atan 1.
      let () = Printf.printf "le nombre %s vaut %.2f à %d decimales près.\n%!."
      ↪caractere_pi pi 2
```

```
[32]: val caractere_pi : string = ""
```

```
[32]: val pi : float = 3.14159265358979312
```

le nombre  vaut 3.14 à 2 decimales près.

La fonction `Printf.fprintf` généralise la `Printf.printf` en prenant un `out_channel` en argument.

```
let oc = open_out "toto.txt" in
Printf.fprintf oc "%s" "hello!";
close_out oc
```

La fonction `Printf.sprintf` écrit dans une chaîne de caractères et la renvoie.

```
[33]: let s = Printf.sprintf "le nombre %s vaut %.2f à %d decimales près."
      ↪caractere_pi pi 2
```

```
[33]: val s : string = "le nombre  vaut 3.14 à 2 decimales près."
```

Ce n'est qu'un aperçu, mais vous croiserez dans le manuel de référence:

- le module `Scanf`, qui permet de faire des *lectures* formatées

```
let read_int ()= Scanf.bscanf "%d" (fun n -> n)
```

- le module `Format`, qui généralise encore le module `Printf` en permettant de gérer la mise en page du texte

Ces deux modules sont hors programme! Si vous poursuivez en OCaml plus tard, vous aurez probablement l'occasion de vous y intéresser.