# Cours de programmation fonctionnelle et logique

# Cours 8 : les exceptions

## Table des matières



## 1 Erreurs et rattrapages d'erreurs

Toute fonction doit renvoyer une valeur (à moins qu'elle ne boucle indéfiniment).

Problème : il existe des fonctions qui ne peuvent retourner de valeurs sensées pour toutes les valeurs possibles de leurs arguments :

- division par 0 ;
- fonction qui renvoie la tête d'une liste appliquée à la liste vide.

Dans ce cas la fonction doit échouer. Il y a arrêt des calculs et renvoi d'un « signal d'erreur ».

### failwith

```
#failwith;;
 - : string -> 'a = <fun>
```

```
#let tete = function
    x::_ -> x
  |  _   -> failwith "tete";;
     val tete : 'a list -> 'a = <fun>
```

```
#tete [];;
 Uncaught exception: Failure("tete")
```

« Uncaught exception » signifie «exception non rattrapée ». On a déclenché une valeur exceptionnelle, ce qui a interrompu l'exécution du programme, mais on n'a pas essayé d récupérer cette valeur.

On a donc totalement interrompu l'exécution mais il se pouvait très bien que l'on était capable de continuer le calcul, même en cas d'erreur. Exemple : fonction qui calcule la somme des têtes d'une liste de listes.

```
#let rec somme_têtes_de_listes = function
    [] -> 0
 | t::r -> (somme_têtes_de_listes r) + (tete t);;
     val somme_têtes_de_listes : int list list -> int = <fun>
```

```
#somme_têtes_de_listes [[2;3];[];[1;4]];;
 Uncaught exception: Failure("tete")
```

Ici le calcul s'arrête car une des listes est vide. On pourrait simplement décider que si une des listes est vide elle ne contribue pas à la somme. La solution est de « rattraper » l'exception.

### `try ... with`

`try ... with` : signifie « essayer ... avec ». `try` expression `with` filtrage sur les valeurs exceptionnelles.

```
#let rec somme_têtes_de_listes = function
    [] -> 0
  | t::r -> (somme_têtes_de_listes r) +
          try (tete t) with Failure "tete" -> 0;;
      val somme_têtes_de_listes : int list list -> int = <fun>
```

```
#somme_têtes_de_listes [[2;3];[];[1;4]];;
 - : int = 3
```

Sémantique : on essaye de calculer l'expression

- Si tout se passe bien, on renvoie le résultat.
- Sinon, une valeur exceptionnelle est générée.
  - Si cette valeur exceptionnelle rentre dans un des cas du filtrage, on évalue l'expression qui lui est associée par le filtrage.
  - Sinon, on renvoie la valeur exceptionnelle, l'exception se propage. Tout se passe comme si le filtrage se terminait toujours par le filtre `x -> x`.

```
#let test l = try tete l with Failure "tete" -> 0;;
 val test : int list -> int = <fun>
```

```
#test [1];;
 - : int = 1
```

```
#test [];;
 - : int = 0
```

```
#let test l = try tete l with Failure "reste" -> 0;;
 val test : int list -> int = <fun>
```

```
#test [1];;
 - : int = 1
```

```
#test [];;
 Uncaught exception: Failure("tete")
```

## 2 Valeurs exceptionnelles

Les « erreurs » sont des valeurs à part entière du langage. On les appelle « valeurs exceptionnelles ». Elles appartiennent au type prédéfini `exn`.

L'échec signalé par la fonction tête est la valeur exceptionnelle `Failure "tete"`.

```
#let échec_de_tête = Failure "tete";;
 val échec_de_tête : exn = Failure("tete")
```

```
#échec_de_tête;;
 - : exn = Failure("tete")
```

`Failure` est en fait un constructeur du type `exn`.

Pour *lever* une valeur exceptionnelle, on utilise la fonction `raise`.

```
#raise;;
 - : exn -> 'a = <fun>

#raise échec_de_tête;;
 Uncaught exception: Failure("tete")
```

La fonction `raise` interrompt immédiatement les calculs en cours pour déclencher la valeur exceptionnelle qu'elle a reçu en argument. `raise` peut intervenir dans n'importe quel contexte et avec n'importe quel type. Comme les calculs ne seront de toutes façons jamais effectués, le contexte peut faire toutes les hypothèses qu'il veut sur la valeur renvoyée par « raise ». Il faut quand même que l'expression soit bien formée.

```
#let f x = x + (raise échec_de_tête);;
 val f : int -> int = <fun>

#let g x = (raise échec_de_tête) ^ x;;
 val g : string -> string = <fun>

#let h x y = x + (raise échec_de_tête) ^ y;;
 Characters 12-37:
This expression has type int but is here used with type string
```

# 3 Définition d'exceptions

De nombreuses fonctions `ocaml` déclenchent, quand elles échouent, l'exception construite au moyen de `Failure` avec leur nom en argument, grâce au déclencheur prédéfini `failwith`.

```
#let failwith s = raise (Failure s);;
 val failwith : string -> 'a = <fun>
```

Le type `exn` est un type somme dont la définition n'est jamais achevée : on peut toujours lui rajouter des constructeurs de types constants ou rationnels au moyen du déclarateur `exception`;;

```
#exception Stop;;
 exception Stop

#exception Erreur_fatale of int;;
 exception Erreur_fatale of int

#raise (Erreur_fatale 4);;
 Uncaught exception: Erreur_fatale(4)
```

# 4 Les exceptions comme moyen de calcul

Les exceptions permettent de transporter un résultat et permettent de signaler un événement attendu.

```
#exception Trouvé;;
 exception Trouvé

#let recherche_char_in_string s c =
   let rec recherche_aux i =
     if i = -1
     then false
     else if s.[i]= c
          then raise Trouvé
          else recherche_aux (i-1)
 in try recherche_aux ((String.length s)-1)
    with Trouvé -> true;;
                 val recherche_char_in_string : string -> char -> bool =
 <fun>
```

```
#recherche_char_in_string "qwerty" 'q';;
 - : bool = true
```

```
#recherche_char_in_string "qwerty" 'a';;
 - : bool = false
```

Cette fonction prend en entrée une liste de chaînes de caractères et renvoie, s'il en existe une, une liste qui contient le caractère recherché.

```
#exception Liste of string;;
 exception Liste of string
```

```
#let recherche_liste n l =
 try
   List.fold_right
     (function x -> function y -> if recherche_char_in_string x n
                                   then raise (Liste x)
                                   else y)
     l
     ""
 with Liste s -> s;;
                 val recherche_liste : char -> string list -> string = <
fun>
```

```
#recherche_liste 'e' ["abc";"def";"ghi"];;
 - : string = "def"
```

```
#recherche_liste 'z' ["abc";"def";"ghi"];;
 - : string = ""
```