PF chapître 5 : exceptions

Jean-François Monin

POLYTECH GRENOBLE

Que rendre en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

Que rendre en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

```
# 1/0;;
```

Que rendre en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

```
# 1/0;;

# List.hd [ ];;
```

Que rendre en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

```
# 1/0;;

# List.hd [ ];;
```

Solution envisageable : renvoyer un couple (valeur, booléen)

- (*a*, `true`) signifie : « la fonction a réussi et son résultat est *a* »
- (*a*, `false`) signifie : « la fonction a échoué »
  (et *a* n'a pas de sens)

Limite :

Que rendre en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

```
# 1/0;;

# List.hd [ ];;
```

Solution envisageable : renvoyer un couple (valeur, booléen)

- (*a*, `true`) signifie : « la fonction a réussi et son résultat est *a* »
- (*a*, `false`) signifie : « la fonction a échoué »
  (et *a* n'a pas de sens)

Limite : propagation.

Que rendre en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

Que rendre en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

```
# 1/0;;
```

Que rendre en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

```
# 1/0;;

Exception: Division_by_zero.
```

Que rendre en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

```
# 1/0;;
  Exception: Division_by_zero.
# List.hd [ ] ;;
  Exception: Failure "hd".
```

Que rendre en cas d'erreur ?

Toute fonction qui ne boucle pas renvoie une valeur.

```
# 1/0;;
  Exception: Division_by_zero.
# List.hd [ ] ;;
  Exception: Failure "hd".
```

Solution OCaml : les exceptions

Bonne solution : failwith

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

Bonne solution : failwith

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

Exemple

```
# let tete = fun l → match l with
  | []   -> failwith "tete vide"
  | x::s -> x ;;

val tete : 'a list -> 'a = <fun>

# tete [];;
Exception: Failure "tete vide".
```

Encore mieux : exception spécifique

raise
exception ListeVide

Encore mieux : exception spécifique

```
raise
exception ListeVide

let tete = fun l -> match l with
    | x :: s -> x
    | [] -> raise ListeVide
```

Encore mieux : exception spécifique

```
raise
exception ListeVide

let tete = fun l -> match l with
    | x :: s -> x
    | [] -> raise ListeVide


# tete []
Exception: ListeVide.
```

Encore mieux : exception spécifique

raise
exception ListeVide

```
let tete = fun l -> match l with
    | x :: s -> x
    | [] -> raise ListeVide
```

```
# tete []
Exception: ListeVide.
```

Interrompt l'ordre habituel de l'évaluation
Est propagée dans les fonctions appelantes

Le type des exceptions

ListeVide : de type exn
raise ListeVide : de type déterminé en fonction du contexte

Le type des exceptions

ListeVide : de type exn
raise ListeVide : de type déterminé en fonction du contexte

exn est :

- un type somme particulier
- étendu dynamiquement :
  exception ListeVide introduit un nouveau constructeur de exn.

Le type des exceptions

ListeVide : de type exn
raise ListeVide : de type déterminé en fonction du contexte

exn est :

- un type somme particulier
- étendu dynamiquement :
  exception ListeVide introduit un nouveau constructeur de exn.

Important

Une exception peut transporter de l'information :

exception Alarme of `int *bool`

Syntaxe des exceptions

```
# exception Erreurfatale of string;;
exception Erreurfatale of string
# raise (Erreurfatale "crash in computation of blabla");;
Exception: Erreurfatale "crash in execution of blabla".
```

Attention à la Majuscule

Syntaxe des exceptions

```
# exception Erreurfatale of string;;
exception Erreurfatale of string
# raise (Erreurfatale "crash in computation of blabla");;
Exception: Erreurfatale "crash in execution of blabla".
```

Attention à la Majuscule

Deux lanceurs d'exceptions prédéfinis

`failwith` "toto" ≡ `raise` (Failure "toto")

`invalid_arg` "toto" ≡ `raise` (Invalid_argument "toto")

Récupération d'exception (évaluation)

$T =$

```
try E with
    | ListeVide → E1
    | Alarme (n, b) → E2(n, b)
    | Not_found → E3
```

(une expression)

Récupération d'exception (évaluation)

T = `try` E `with`
| ListeVide $\rightarrow E_1$
| Alarme $(n, b) \rightarrow E_2(n, b)$
| Not_found $\rightarrow E_3$

(une expression)

Évaluation

E est évalué en premier, puis :

Récupération d'exception (évaluation)

T =

```
try E with
    | ListeVide → E1
    | Alarme (n, b) → E2(n, b)
    | Not_found → E3
```

(une expression)

Évaluation

E est évalué en premier, puis :

- si aucune exception levée : valeur de T = valeur de E

Récupération d'exception (évaluation)

T =

```
try E with
    | ListeVide → E1
    | Alarme (n, b) → E2(n, b)
    | Not_found → E3
```

(une expression)

Évaluation

E est évalué en premier, puis :

- si aucune exception levée : valeur de T = valeur de E
- si exception e levée, on la compare aux motifs du `with`

Récupération d'exception (évaluation)

T =

```
try E with
    | ListeVide → E1
    | Alarme (n, b) → E2(n, b)
    | Not_found → E3
```

(une expression)

Évaluation

E est évalué en premier, puis :

- si aucune exception levée : valeur de T = valeur de E
- si exception e levée, on la compare aux motifs du `with`
  - si filtrage réussi au i^{e} motif : valeur de T = valeur de E_i

Récupération d'exception (évaluation)

T =

```
try E with
    | ListeVide → E1
    | Alarme (n, b) → E2(n, b)
    | Not_found → E3
```

(une expression)

Évaluation

E est évalué en premier, puis :

- si aucune exception levée : valeur de T = valeur de E
- si exception e levée, on la compare aux motifs du `with`
  - si filtrage réussi au i^{e} motif : valeur de T = valeur de E_i
  - sinon l'exception e est transmise à la fonction appelante (comme s'il n'y avait pas de `try`)

Exemple

Écrire une fonction prenant une liste d'entiers et renvoyant la somme des ses éléments ; si la liste contient un négatif alors renvoyer -1.

Exemple

Écrire une fonction prenant une liste d'entiers et renvoyant la somme des ses éléments ; si la liste contient un négatif alors renvoyer -1.

Un premier programme

```
let rec somme = fun l → match l with
 | []     -> 0
 | x :: r -> if x < 0 then -1 else x + (somme r)
```

Exemple

Écrire une fonction prenant une liste d'entiers et renvoyant la somme des ses éléments ; si la liste contient un négatif alors renvoyer -1.

Un premier programme

```
let rec somme = fun l → match l with
 | []    -> 0
 | x :: r -> if x < 0 then -1 else x + (somme r)
```

ne fonctionne pas

Solution 1 : sans exception

Solution 1 : sans exception

```
let somme =  fun l →
  let rec testpos =  fun l → match l with
    | []      ->  true
    | x :: r ->  x > 0  &&  (testpos r)
  and let rec somme_aux = fun l → match l with
    | []      ->  0
    | x :: r ->  x  +  (somme_aux r)
  in if testpos l  then  somme_aux l  else  -1 ;;
```

Problème :

Solution 1 : sans exception

```
let somme =  fun l →
  let rec testpos =  fun l → match l with
    | []     ->  true
    | x :: r ->  x > 0  &&  (testpos r)
  and let rec somme_aux = fun l → match l with
    | []     ->  0
    | x :: r ->  x  +  (somme_aux r)
  in if testpos l  then  somme_aux l  else  -1 ;;
```

Problème : 2 parcours de la liste

Solution 2 : sans exception

```
let rec somme =  fun l → match l with
    | []      ->  0
    | x :: r ->  let s = somme r
                 in if  s = -1 || x < 0
                    then  -1
                    else  x + s
```

Problème :

Solution 2 : sans exception

```
let rec somme =  fun l → match l with
    | []      ->  0
    | x :: r ->  let s = somme r
                 in if  s = -1 || x < 0
                    then  -1
                    else  x + s
```

Problème : mélange entre code fonctionnel et gestion d'erreur
test `s = -1` systématique

Solution 3 : avec exception

```
exception Negatif;;
```

Solution 3 : avec exception

```
exception Negatif;;

let somme =  fun l →
  let rec somme_aux =  fun l → match l with
    | []                  ->  0
    | x::_ when x < 0 ->  raise Negatif
    | x::r                ->  x  +  somme_aux r
  in try
    somme_aux l
  with
    Negatif -> -1;;
```

Solution 3 : avec exception

```
exception Negatif;;

let somme =  fun l →
  let rec somme_aux =  fun l → match l with
    | []                  ->  0
    | x::_ when x < 0 ->  raise Negatif
    | x::r                ->  x  +  somme_aux r
  in try
    somme_aux l
  with
    Negatif -> -1;;
```

Séparation du code fonctionnel et de la gestion d'erreur

Application : utilisation d'exceptions en mise au point

Objectif : se ramener à des problèmes plus petits traités un à un

```
exception PasEncoreDef of string *string

let rec oppose = fun i →
  match i with
  | Ent (e) →
  | Reel (r) →
  | Cplx (r,i) →
```

Application : utilisation d'exceptions en mise au point

Objectif : se ramener à des problèmes plus petits traités un à un

```
exception PasEncoreDef of string *string

let rec oppose = fun i →
  match i with
  | Ent (e) →  raise PasEncoreDef ("oppose", "Ent")
  | Reel (r) →  raise PasEncoreDef ("oppose", "Reel")
  | Cplx (r,i) →  raise PasEncoreDef ("oppose", "Cplx")
```

Application : utilisation d'exceptions en mise au point

Objectif : se ramener à des problèmes plus petits traités un à un

```
exception PasEncoreDef of string *string

let rec oppose = fun i →
  match i with
  | Ent (e) →  raise PasEncoreDef ("oppose", "Ent")
  | Reel (r) →  raise PasEncoreDef ("oppose", "Reel")
  | Cplx (r,i) →  raise PasEncoreDef ("oppose", "Cplx")
```

On peut alors

- tester le programme global dès le départ
- proposer le vrai code pour chaque cas pris séparément

Application : tester ses programmes

Un test = une entrée possible de la fonction + le résultat attendu

```
let _ = assert (f entree = resultat)
```

Application : tester ses programmes

Un test = une entrée possible de la fonction + le résultat attendu

```
let _ = assert (f entree = resultat)
```

Couverture

- tests représentatifs de toutes les données acceptables
- permettant d'observer tous les résultats possibles
- ne pas oublier les cas extrêmes : liste vide, 0, etc.

Application : tester ses programmes

Un test = une entrée possible de la fonction + le résultat attendu

```
let _ = assert (f entree = resultat)
```

Couverture

- tests représentatifs de toutes les données acceptables
- permettant d'observer tous les résultats possibles
- ne pas oublier les cas extrêmes : liste vide, 0, etc.

Les données incorrectes (mauvais type, entier négatif...) relèvent de la *robustesse*, moins importante ici.

Application : tester ses programmes

Un test = une entrée possible de la fonction + le résultat attendu

```
let _ = assert (f entree = resultat)
```

Couverture

- tests représentatifs de toutes les données acceptables
- permettant d'observer tous les résultats possibles
- ne pas oublier les cas extrêmes : liste vide, 0, etc.

Les données incorrectes (mauvais type, entier négatif...) relèvent de la *robustesse*, moins importante ici.

Conserver les tests et les réévaluer à chaque modification

- Non régression
- Identification plus aisée des bugs