

Programmation Fonctionnelle

Cours 07

Michele Pagani



Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

pagani@pps.univ-paris-diderot.fr

26 octobre 2015

Les Exceptions, i.e. le type `exn`

À quoi ça sert les exceptions ?

① traiter application fonction en dehors du domain de définition

- toujours préférable à l'envoi des valeurs légales mais bidon
- arrêt du programme si l'exception n'est pas rattrapée

```
# 3/0;;  
Exception: Division_by_zero.
```

```
# List.hd [];;  
Exception: Failure "hd".
```

② marquer mauvaise interaction avec environnement extérieur

```
# let x = read_int ();;  
trois  
Exception: Failure "int_of_string".
```

③ rendre un calcul plus efficace (voir plus tard)

Exceptions (`exn`)

- les exceptions sont des valeurs d'un **type** `exn`:

```
type exn=  
  ...  
  | Division_by_zero  
  | Failure of string  
  | Invalid_argument of string  
  ...
```

- `exn` est un type **somme**:
 - `Division_by_zero`, `Failure`, ... sont les constructeurs,
 - parfois ils ont un argument pour transporter de l'information sur l'origine de l'exception

- en effet, `exn` est un type **extensible**:

```
# exception Echec ;;  
exception Echec  
# exception Int_exception of int;;  
exception Int_exception of int
```

Exceptions (raise)

- on peut lever une exception en utilisant `raise`:

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

- le type `'a` permet que la levée d'une exception soit compatible avec n'importe quelle autre expression:

```
# let rec fact = function  
  0 -> 1  
  | n -> if n>0 then n*(fact (n-1))  
         else raise (Int_exception n);;  
val fact : int -> int = <fun>  
  
# fact (-1);;  
Exception: Int_exception (-1).
```

- la levée d'une exception arrête l'évaluation d'une expression:

```
# (fun x -> 1) (fact (-1));;  
Exception: Int_exception (-1).
```

Exceptions (divide.ml)

```
1 let divide x =  
2   print_endline "donner le numérateur";  
3   let n = read_int () in  
4   print_endline "donner le dénominateur";  
5   let m = read_int () in  
6   n/m  
7  
8 let rec main () =  
9   try  
10    begin  
11     let r = divide () in  
12     print_string "la division est ";  
13     print_int r  
14     end  
15   with  
16   | Division_by_zero ->  
17     (print_endline "Grrr! pas choisir 0!"; main ())  
18   | Failure("int_of_string") ->  
19     print_endline "Tapez un entier"; main ()  
20 ;;  
21  
22 main ()
```

Exceptions (try ... with ...)

```
try expr with  
| excep1 -> expr1  
...  
| excep n -> exprn
```

- un filtrage par motif capable de rattraper les exceptions
- on évalue `expr`,
 - si pas d'exception, alors la valeur est la valeur de `expr`
 - sinon, on filtre la valeur de l'exception avec `excep1`, puis `excep2`, puis `excep3`, ...
 - si on trouve un motif compatible, alors on évalue l'expression correspondante (rattrapage de l'exception)
 - sinon on propage l'exception jusqu'à la prochaine `try...with...`
- les motifs doivent être du type `exn`
- les expressions dans le filtrage doivent être du même type

Exceptions (complet1.ml)

```
1 (* sans exceptions, avec parcours multiples *)  
2 type arbre = F | N of arbre * arbre;;  
3  
4 let rec hauteur a =  
5   match a with  
6   | F -> 0  
7   | N(g,d) -> 1 + max (hauteur g) (hauteur d);;  
8  
9 let rec complet a =  
10  match a with  
11  | F -> true  
12  | N(g,d) -> (complet g) && (complet d) (* parcours g,d *)  
13         && (hauteur g = hauteur d);; (* parcours g,d *)  
14  
15  
16 complet (N(N(F,F),N(F,F)));;  
17 complet (N(N(F,F),N(F,N(F,F))));;
```

Exceptions (complet2.ml)

```
1 (* avec exceptions et un seul parcours de l'arbre : *)
2 exception Incomplet
3
4 let complet a =
5   let rec haut_aux a = match a with
6     | F -> 0
7     | N(g,d) ->
8       let hg = haut_aux g
9       and hd = haut_aux d in
10      if hg = hd then (1 + hg) else raise Incomplet
11   in try
12     let _ = haut_aux a in true
13   with Incomplet -> false;;
14
15 complet (N(N(F,F),N(F,F)));;
16 complet (N(N(F,F),N(F,N(F,F))));;
```

Assertions (assert)

- Mot clef **assert**, suivi d'une expression booléenne
- Lève une exception `Assert_failure` quand l'expression donne `false`, avec nom du fichier, numéro de ligne et colonne
- Utile pour déclarer des invariants
- Utile pour tester des invariants pendant l'exécution
- On peut désactiver les assertions pendant la compilation d'un programme:
 `ocamlc -noassert`
- Premier langage avec assertions : [Eiffel](#)

Assertions (Exemple)

```
1 let rec fib n =
2   assert (n >= 0);
3   match n with
4   | 0 | 1 -> n
5   | n -> fib (n-1) + fib (n-2)
6
7 let main () =
8   let n = int_of_string (Sys.argv.(1)) in
9   print_string "factorial_ is_";
10  print_int (fib n);
11  print_newline ()
12 ;;
13
14 main ()
```

Input / Ouput

in_channel, out_channel

```
# stdin;;  
- : in_channel = <abstr>  
# stdout;;  
- : out_channel = <abstr>  
# stderr;;  
- : out_channel = <abstr>
```

- deux types correspondants aux **canaux de communication**:
 - `in_channel` pour les canaux d'entrée
 - `out_channel` pour les canaux de sortie
- tout canal est soit un canal d'entrée, soit un canal de sortie, mais jamais les deux à la fois !
- tout processus UNIX a trois canaux par défaut:
 - `stdin` : entrée "normale" du processus (usuellement clavier)
 - `stdout` : sortie "normale" du processus (usuellement écran)
 - `stderr` : sortie messages erreur (souvent confondue avec `stdout`)
- on peut les rediriger (par exemple à un tuyau ou un fichier)

Ouvrir/ fermer un fichier pour écriture

```
# open_out;;  
- : string -> out_channel = <fun>  
# close_out;;  
- : out_channel -> unit = <fun>
```

- `open_out` crée un canal de sortie pour écrire sur un fichier:
 - si le fichier n'existe pas, il sera créé
 - si le fichier existe
 - mais on n'a pas les droits d'écriture: exception `Sys_error`
 - sinon, le contenu précédent sera écrasé (**attention !**)
- `open_out_gen` offre plus d'options d'ouverture
(ne pas écraser contenu, écriture en binaire, ...) [voir manuel](#)
- `close_out` ferme un canal ouvert et en écrivant le contenu dans le buffer associé

Ouvrir/ fermer un fichier pour écriture (Exemples)

```
# let ch = open_out "toto";;  
val ch : out_channel = <abstr>  
  
# ch;;  
- : out_channel = <abstr>  
  
# close_out ch;;  
- : unit = ()  
  
(*ouverture fichier sans permission ecriture*)  
# let ch = open_out "titi";;  
Exception: Sys_error "titi: Permission denied".
```

Écrire vers un canal

```
output_char : out_channel->char->unit  
écrit un caractère (8-bits ASCII)
```

```
output_string : out_channel->string->unit  
écrit une chaîne de caractères
```

```
output : out_channel->string->int->int->unit  
écrit une sous-chaîne
```

```
output_byte : out_channel->int->unit  
écrit un int comme un octet
```

```
output_binary_int : out_channel->int->unit  
écrit un int en binaire (un ou plusieurs octets)
```

```
seek_out : out_channel->int->unit  
change la position d'écriture sur le fichier
```

```
pos_out : out_channel->int  
renvoie position actuelle d'écriture
```

Écrire vers un canal

- La sortie vers un canal est tamponnée (en angl. **buffered**)
- le contenu d'un tampon est vidé au moment de cloture du canal
- sinon, les fonctions suivantes vident le contenu de un/tous tampons
 - `flush : out_channel -> unit`
 - `flush_all : unit -> unit`

Écrire vers un canal (Exemples)

```
# let rec print_list canal = function
| [] -> ()
| h::r ->
    output_string canal (string_of_int h);
    output_char canal '\n';
    print_list canal r
;;
val print_list : out_channel -> int list -> unit = <fun>

# let ch = open_out "myfile" in
    print_list ch [3;5; 17; 2; 256];
    close_out ch;;
- : unit = ()
```

Écrire vers un canal (Exemples)

```
(* erreur d'execution *)
let c = open_out "myfile" in
    close_out c ;
    output_string c "toto"
;;

(* erreur de typage *)
let c = open_in "myfile" in
    output_string c "coocoo";
    close_in c
;;
```

Ouvrir/ fermer un fichier pour lecture

```
# open_in;;
- : string -> in_channel = <fun>
# close_in;;
- : in_channel -> unit = <fun>
```

- `open_in` crée un canal d'entrée pour lire un fichier
 - si le fichier ne peut pas être ouvert (par exemple parce qu'il n'existe pas): exception `Sys_error`
- `open_in_gen` offre plus d'options d'ouverture [voir manuel](#)
- `close_in` ferme un canal ouvert

Lire par un canal

```
input_char : in_channel->char
    lit un caractère (8-bits ASCII)

input_line : in_channel->string
    lit une ligne complète

input_byte : in_channel->int
    lit un int depuis un octet

input_binary_int : out_channel->int
    lit un int en binaire (depuis un ou plusieurs octets)
```

```
seek_in : in_channel->int->unit
    change la position de lecture sur le fichier

pos_in : in_channel->int
    renvoie position actuelle de lecture
```

- exception `End_of_file` quand on est à la fin du fichier.

Lire/Écrire (Exemple)

```
# let rec copy_lines ci co =
  try
    let x = input_line ci
  in
    output_string co x;
    output_string co "\n";
    copy_lines ci co
  with
    End_of_file -> ();;
val copy_lines : in_channel -> out_channel -> unit = <fun>

# let copy infile outfile =
  let ci = open_in infile
  and co = open_out outfile
  in
    copy_lines ci co;
    close_in ci;
    close_out co;;
val copy : string -> string -> unit = <fun>
```

Exemple (Ordre évaluation)

```
# (* risque d'entrer dans une boucle infinie !*)
let rec count_bytes ci =
  try
    String.length (input_line ci) + count_bytes ci
  with
    End_of_file -> 0
;;
val count_bytes : in_channel -> int = <fun>

# let c = open_in "myfile" in count_bytes c;;
Stack overflow during evaluation (looping recursion?).
```

Attention à l'ordre d'évaluation

- l'ordre d'évaluation des arguments dans une expression n'est pas spécifié
- les opérations de sortie ont un effet de bord, mais les opérations d'entrée aussi (!): elle font avancer la tête de lecture
- dans le cas des fonctions récursives: assurer que la tête de lecture est avancée avant d'entrer dans la récurrence !
- seulement les opérateurs booléen (`&&` et `||`) ont un ordre d'évaluation garanti de gauche vers la droite
- sinon on peut donner ordonner les étapes d'évaluation à travers:
 - déclarations locales
 - composition séquentielle ;

Exemple (correction)

```

(*OK*)
let rec count_bytes ci =
  try
    let bytes_this_line = String.length (input_line ci)
    in bytes_this_line + count_bytes ci
  with
    End_of_file -> 0;;
val count_bytes : in_channel -> int = <fun>

# let c = open_in "myfile" in count_bytes c;;
- : int = 8

```

Les modules Printf/Scanf

- les modules `Printf` et `Scanf` contiennent plusieurs fonctions pour écrire (lire) dans des formats précis, similaires aux instructions correspondantes en C.
- `printf` prend en premier argument une chaîne qui décrit le format, puis tant d'arguments que demandé par le format et l'écrit sur `stdout`
 - dans le format, `%i` dénote un entier, `%s` une chaîne de caractères, etc...
- `fprintf` permet d'écrire sur des canaux différents de `stdout`,
- `scanf` et `fscanf` sont les fonctions de lecture correspondantes.