Programmation Fonctionnelle
Cours 06

Michele Pagani

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

pagani@pps.univ-paris-diderot.fr

30 octobre 2014

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)

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)

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 crée
  - 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

```
# 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 crée
  - 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

```
# 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 crée
  - 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 chaine de caractères

output : out_channel->string->int->int->unit
écrit une sous-chaine

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

- 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

- 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 ouverte (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

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 ouverte (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

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 ouverte (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>
```

Lire/Écrire (Recursion terminale)

- sur des petits fichiers copy marche bien, mais en general...

```
# copy "words" "toto";;
Stack overflow during evaluation (looping recursion?).
```

- un appel récursif cesse d'être terminal si à l'intérieur d'un try:

```
try
....
copy_lines ci co
with
.....
```

copy_lines ci co n'est pas terminal car il faut mémoriser la position de son appel pour pouvoir traiter une exception éventuelle.

Lire/Écrire (Recursion terminale)

- sur des petits fichiers copy marche bien, mais en general...

```
# copy "words" "toto";;
Stack overflow during evaluation (looping recursion?).
```

- un appel récursif cesse d'être terminal si à l'intérieur d'un try:

```
try
....
copy_lines ci co
with
.....
```

copy_lines ci co n'est pas terminal car il faut mémoriser la position de son appel pour pouvoir traiter une exception éventuelle.

Lire/Écrire (Correction)

```
(*copy file: tail recursive*)
type 'a option = None | Some of 'a

let rec copy_lines ci co =
let x =
try
Some (input_line ci)
with
| End_of_file -> None
in
match x with
| Some l ->  output_string co l;
             output_string co "\n";
             copy_lines ci co
| None -> ()

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
```

- Voir module `Option` sur le Manuel.

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 ;

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 ;

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 ;

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'écrive 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.

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'écrive 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.

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'écrive 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.

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'écrive 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.