

Programmation Fonctionnelle

Cours 6

Entrées, sorties

Delia Kesner

Concept préliminaire: le type `unit`

- ▶ Le type `unit` contient un seul élément, noté `()` et prononcé également *unit*.
`();;`
`# - : unit = ()`
- ▶ Utile par exemple pour des fonctions qui ne n'envoient pas de résultat intéressant.
- ▶ Utile aussi pour les instructions d'affichage.

Les canaux de communication

- ▶ Les entrées et sorties d'un programme OCaml utilisent des *canaux de communications* (sauf cas spéciaux tels que le graphisme).
- ▶ Tout canal est soit un canal d'entrée, soit un canal de sortie, mais jamais les deux à la fois.
- ▶ Certains de ces canaux sont créés par défaut (voir transparent suivant), d'autres peuvent être ouverts et fermés par le programme (voir plus tard).

Les canaux de communication

Tout processus UNIX a trois canaux de communication (voir le cours de *Systèmes*) :

- ▶ *stdin* : entrée « normale » du processus, normalement associée au clavier. Peut aussi être une redirection d'un tuyau ou d'un fichier.
- ▶ *stdout* : sortie « normale » du processus, normalement associée à l'écran. Peut aussi être redirigée vers un tuyau ou un fichier.
- ▶ *stderr* : sortie pour les messages d'erreur. Normalement confondue avec *stdout* et associée à l'écran, mais peut aussi être redirigée.

Les canaux de communication I

```
stdin;;
```

```
# - : in_channel = <abstr>
```

```
stdout;;
```

```
# - : out_channel = <abstr>
```

```
stderr;;
```

```
# - : out_channel = <abstr>
```

Sortie vers *stdout*

- ▶ Fonctions de sortie vers *stdout* pour tous les types de base
- ▶ La sortie vers *stdout* n'est pas effectuée tout de suite : il y a un tampon. Un saut de ligne (p.ex. via `print_newline`) force la sortie du contenu du tampon.
- ▶ Pour des valeurs dont le type n'est pas un type de base (p.ex. listes , types sommes, etc), c'est à nous d'écrire des fonctions d'affichage vers *stdout*. Même si en fait l'interpréteur OCaml sait en afficher la plupart quand il nous présente un résultat !
- ▶ Si on veut sortir des telles valeurs vers *stdout* alors il faut écrire une fonction spéciale pour le faire.

Fonctions pour imprimer sur *stdout* |

```
print_int;;  
# - : int -> unit = <fun>  
print_float;;  
# - : float -> unit = <fun>  
print_string;;  
# - : string -> unit = <fun>  
print_char;;  
# - : char -> unit = <fun>  
(*usage*)  
print_string "toto";;  
# toto- : unit = ()  
print_newline();;  
#  
- : unit = ()
```

Fonctions pour imprimer sur *stdout* II

(*ou directement*)

```
print_string "toto\n";;
```

```
# toto
```

```
- : unit = ()
```

(*ou on peut enchaîner deux actions*)

```
print_string "toto"; print_newline ();;
```

```
# toto
```

```
- : unit = ()
```


Le module Printf

- ▶ Ce module définit une fonction `printf` qui prend en premier argument une chaîne qui décrit un format, puis tant d'arguments que demandé par le format.
- ▶ Dans le format, `%i` dénote un entier, `%s` une chaîne de caractères, etc.
- ▶ Il y a des variantes pour écrire sur un canal de sortie quelconque ou dans une chaîne de caractères.
- ▶ Cette fonction « triche » au niveau typage (car le nombre et les types des arguments dépendent du premier argument).
- ▶ Similaire à `printf` dans C, C++, Java, ...

Utilisation de Printf |

```
(* type dependant, sera vu en M1 *)
```

```
Printf.printf;;
```

```
# - : ('a, out_channel, unit) format -> 'a = <fun>
```

```
(* il faut fournir des arguments selon le format*)
```

```
Printf.printf "La longueur de %s est %i\n";;
```

```
# - : string -> int -> unit = <fun>
```

```
(* usage *)
```

```
Printf.printf "La longueur de %s est %i\n" "toto" 4;;
```

```
# La longueur de toto est 4
```

```
- : unit = ()
```

Sortie vers *stderr*

- ▶ Il y a des fonctions analogues pour la sortie vers *stderr*.
- ▶ La distinction entre *stdout* et *stderr* est importante: un utilisateur peut avoir besoin de séparer la sortie normale des messages d'erreur.
- ▶ Fonctions [prerr_int](#), etc. (Voir le manuel)

Les types des canaux de communication

- ▶ Il y a deux types prédéfinis en OCaml pour les canaux de communication:
 - ▶ `in_channel` pour les canaux d'entrée
 - ▶ `out_channel` pour les canaux de sortie
- ▶ `stdin` est de type `in_channel` tandis que `stdout` et `stderr` sont de type `out_channel`.
- ▶ Des fonctions spécialisées permettent de créer un nouveau canal en l'associant à un fichier, à une connexion réseau, etc.

Ouvrir et fermer un fichier pour l'écriture

- ▶ Fonction `open_out` pour ouvrir un fichier en écriture, de type `string -> out_channel`. Si le fichier n'existe pas il est créé.
- ▶ Peut lever une exception `Sys_error`, par exemple quand on n'a pas les droits nécessaires pour créer ou ouvrir le fichier.
- ▶ Fonction `close_out` de type `out_channel -> unit` pour fermer un fichier.

Ouvrier et fermer un fichier pour l'écriture I

```
let c = open_out "myfile";;  
# val c : out_channel = <abstr>  
close_out c;;  
# - : unit = ()
```

```
(* erreur d'exécution *)  
let c = open_out "/blurbel";;  
# Exception: Sys_error "/blurbel: Permission denied".
```

Écrire vers un canal

- ▶ Fonctions `output_string`, `output_char` pour écrire dans un canal de sortie. Le premier argument est le canal.
- ▶ La fonction `print_string` vue auparavant est équivalente à `output_string stdout`.
- ▶ Il n'y a pas de `output_int`, en revanche il y a une variante de `printf` (`Printf.fprintf`) pour écrire dans un canal.
- ▶ La sortie vers un canal est tamponnée (`buffered`).
- ▶ Fonctions `flush` pour vider un tampon et fonction `flush_all` pour les vider tous.

Écrire une liste dans un fichier I

```
let rec print_list canal l = match l with
| [] -> ()
| 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 c = open_out "myfile" in
    print_list c [3; 5; 17; 42; 256];
    close_out c;;
# - : unit = ()
```


Erreurs d'écriture dans un fichier I

(* erreur de typage *)

```
let c = open_in "myfile" in
  output_string c "coocoo";
  close_in c;;
  output_string c "coocoo";
  ^
```

Error: This expression has **type** in_channel
but an expression was expected **of type** out_channel

(* erreur d'exécution *)

```
let c = open_out "myfile" in
  close_out c;
  output_string c "toto";;
# Exception: Sys_error "Bad_file_descriptor".
```

Entrée par *stdin*

- ▶ La fonction `read_line` attend sur *stdin* une ligne terminée par retour-chariot, et envoie comme résultat le contenu de cette ligne (sous forme d'un string) mais **sans** le retour-chariot.
- ▶ Il y a également `read_int` et `read_float`.
- ▶ Le module `Scanf` permet de lire des lignes dans un format précis (analogue à `Printf`, mais d'usage délicat).
- ▶ Pour des lectures plus complexes, il existe des outils dédiés tels que *ocamllex* et *ocamlyacc* ou *menhir* ... et des cours entiers pour les apprendre (Compilation).

Exemples d'entrées I

```
let rec read_and_add x =  
  let y = read_int () in  
    if y = 0 then x else read_and_add (x+y);;  
# val read_and_add : int -> int = <fun>
```

```
(* read_and_add 120;; *)
```

Ouvrir et fermer un fichier pour la lecture

- ▶ Fonction `open_in` pour ouvrir un fichier en lecture. Lève l'exception `Sys_error` si le fichier ne peut pas être ouvert (par exemple parce qu'il n'existe pas).
- ▶ Fonction `close_in` pour fermer le canal.
- ▶ Fonction `input_line` pour lire une ligne complète. Lève l'exception `End_of_file` quand on est à la fin du fichier.

Copier un fichier I

```
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>
```

Copier un fichier II

```
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>  
  
copy "source-file" "target-file";;  
# - : unit = ()
```

Entrées/sorties et effet de bord

- ▶ Les opérations de sortie sont l'exemple typique d'**effets de bord**:
 - ▶ Leur type résultat `unit` n'indique pas l'action faite en chemin
 - ▶ L'ordre d'évaluation des opérations de sorties importe !
- ▶ Les opérations d'entrée sont aussi des effets de bord : elles font avancer la tête de lecture. Faire deux lectures de suite ne donnera sans doute pas le même résultat!
- ▶ Dans le cas des fonctions récursives, s'assurer que la tête de lecture est avancée avant d'entrer dans la récurrence !

Risque d'entrer dans une boucle infinie I

```
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?).
```


Version correcte I

```
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 "myfile2" in count_bytes c;;  
# - : int = 25
```

Attention à l'ordre d'évaluation

- ▶ Le souci précédent : l'expression à droite du '+' est évalué **avant** l'expression à gauche.
- ▶ L'ordre d'évaluation des arguments dans un appel de fonction n'est officiellement pas spécifié en OCaml.
- ▶ En fait, il calcule les arguments d'une expression de la droite vers la gauche!
- ▶ Sauf les opérateurs booléens `&&` et `||`, qui évaluent de la gauche vers la droite (et peuvent parfois ignorer l'argument de droite!)
- ▶ Utiliser des `let ... in ...` pour forcer l'ordre d'évaluation.