

# Programmation Fonctionnelle

## Cours 7

### Entrées, sorties, graphisme

October 20, 2020

## Table de matières

### Entrée et sortie

- Canaux

- Sortie

- Entrée

### Graphisme

- Fonctionnalité de Base

- Événements

## Les canaux de communication vus de l'intérieur

- ▶ 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, et jamais les deux à la fois.
- ▶ Certains de ces canaux sont créés par défaut (voir le transparent suivant), d'autres peuvent être ouverts et fermés par le programme (voir plus tard).

## Les canaux de communication qui existent toujours

Tout processus Unix a au moins 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.

## 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.
- ▶ L'interpréteur sait afficher des valeurs de type autre que les types de bases (par exemple listes, types sommes).
- ▶ Par contre, si on veut sortir une telle valeur vers *stdout* alors c'est à nous d'écrire une fonction pour le faire.

## Exemples (print.ml)

```
(* fonctions pour imprimer sur stdout *)
print_int;;
print_float;;
print_string;;
print_char;;
print_string "toto";;
print_newline ();;
print_string "toto\n";;
print_string "toto"; print_newline ();;
```

## Le module Printf

- ▶ Ce module définit une fonction `printf` qui prend en premier argument une chaîne qui décrit le format, puis autant 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, ...

## Exemples (printf.ml)

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

```
Printf.printf ;;
```

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



## 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 canaux de communication vus de l'intérieur

- ▶ La bibliothèque standard OCaml propose deux types prédéfinis pour les canaux de communication :
  - ▶ `in_channel` pour les canaux d'entrée
  - ▶ `out_channel` pour les canaux de sortie
- ▶ Des fonctions spécialisées permettent de créer un nouveau canal en l'associant par exemple à un fichier, ou à une connexion réseau.

## Exemples ([canaux.ml](http://canaux.ml))

```
stdin;;  
stdout;;  
stderr;;
```

## Ouvrir et fermer un fichier pour l'écriture

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

## Exemples (file1.ml)

```
let c = open_out "myfile";;  
close_out c;;
```

```
(* erreur d'exécution *)  
let c = open_out "/blabla";;
```

## Écrire vers un canal

- ▶ Fonctions `output_string`, `output_char` pour écrire dans un canal de sortie. Le premier argument est le canal.
- ▶ Il n'y a pas de `output_int`, en revanche `Printf.fprintf` est une variante de `printf` écrivant dans un canal.
- ▶ La sortie vers un canal est tamponnée.
- ▶ Fonctions `flush` pour vider un tampon, et `flush_all` les vider tous.

## Exemples (file2.ml)

```
let rec print_list canal = function
| [] -> ()
| h::r ->
    output_string canal (string_of_int h);
    output_char canal '\n';
    print_list canal r
;;
```

```
let c = open_out "myfile" in
print_list c [3; 5; 17; 42; 256];
close_out c;;
```

## Exemples (file3.ml)

```
(* erreur d'exécution *)
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;;
```



## 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`, 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 (read.ml)

```
let rec read_and_add x =
  let y = read_int () in
  if y = 0
  then x
  else read_and_add (x+y)
;;
```

```
(read_and_add 0);;
```

## Ouvrir et fermer un fichier pour la lecture

- ▶ Fonction `open_in`. 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.

```

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 -> ()

let copy infile outfile =
  let ci = open_in infile in
  let co = open_out outfile in
  copy_lines ci co;
  close_in ci;
  close_out co

```

## Entrées/sorties et effet de bord

- ▶ Les opérations de sortie sont l'exemple même 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 !

## Exemples (rec1.ml)

```
(* 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

let c = open_in "myfile" in count_bytes c;;
```

## Exemples (rec2.ml)

```
(* OK *)
let rec count_bytes ci =
  try
    let line_size = String.length (input_line ci)
    in line_size + count_bytes ci
  with End_of_file -> 0

let c = open_in "myfile" in count_bytes c;;
```

## Attention à l'ordre d'évaluation

- ▶ Le souci précédent : dans `rec1.ml` la droite du `+` est évalué *avant* la 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 de la droite vers la gauche!
- ▶ Sauf les opérateurs booléens `&&` et `||`, qui évaluent de gauche vers droite (et peuvent parfois ignorer l'argument de droite!)
- ▶ Utiliser des `let ... in` pour forcer l'ordre d'évaluation.
- ▶ Exercice : avec les fonctions vues aujourd'hui, comment tester en pratique cet ordre d'évaluation des arguments ?



## Le graphisme

- ▶ On utilisera ici la bibliothèque `graphics`
- ▶ Elle est assez “datée”, mais d'usage simple et encore largement disponible

## Disponibilité de graphics

- ▶ Par défaut, l'interprète OCaml n'a pas de primitives graphiques. Plusieurs solutions:
  - ▶ Charger la bibliothèque dans l'interpréteur :  
`#load "graphics.cma";;`
  - ▶ Inclure la bibliothèque dès le lancement de l'interpréteur :  
`ocaml graphics.cma` au lieu de `ocaml`
  - ▶ Créer une nouvelle instance de l'interpréteur à l'aide de la commande `ocamlmktop` (voir le manuel)
- ▶ Pour compiler un programme qui utilise le graphisme, le plus simple est d'utiliser l'outil `done` et d'indiquer une dépendance envers la bibliothèque `graphics`.

## La fenêtre graphique

- ▶ `open Graphics` met toutes les fonctions (types, exceptions) de cette bibliothèque disponible (on n'a plus besoin de la notation pointée `Graphics.quelquechose`)
- ▶ Ouverture de la fenêtre graphique (unique) :  
`open_graph " 800x600"`. Ajuster si besoin la largeur et la hauteur (en pixels). Attention à l'espace devant la largeur : obligatoire en Unix mais à enlever sous Windows.
- ▶ `close_graph: unit -> unit` ferme la fenêtre graphique.
- ▶ `clear_graph: unit -> unit` efface le contenu de la fenêtre graphique.

## Exemples (graphics1.ml)

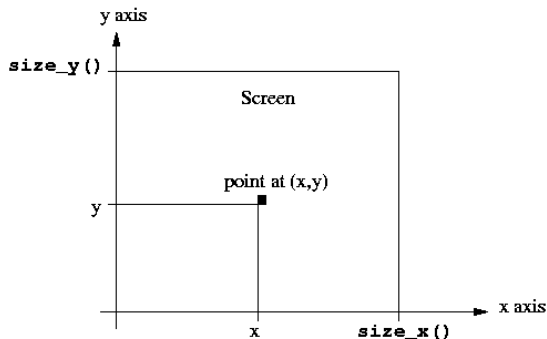
```
(* uniquement dans un toplevel: *)
#load "graphics.cma";;

open Graphics;;

open_graph "□800x600";;

close_graph ();;
```

## Coordonnées sur le canvas graphique



L'origine (0,0) est en bas à gauche

# Dessiner

- ▶ Il y a un curseur, qui au début se trouve à l'origine  $(0,0)$ .
- ▶ `(plot x y)` dessine un point à la position  $(x,y)$  et positionne le curseur graphique en ce point ;
- ▶ `(moveto x y)` positionne le curseur graphique en  $(x,y)$  ;
- ▶ `(lineto x y)` dessine un trait du curseur graphique de la position actuelle du curseur à  $(x,y)$ , et positionne le curseur graphique en  $(x,y)$  ;
- ▶ `(set_line_width n)` sélectionne  $n$  pixels comme épaisseur des lignes.

```
open Graphics;;
```

```
open_graph "_800x600";;
```

```
moveto 200 200;;
```

```
lineto 400 200; lineto 400 300;
```

```
lineto 200 300; lineto 200 200;;
```

```
set_line_width 5;;
```

```
moveto 150 150;;
```

```
lineto 450 150; lineto 450 350;
```

```
lineto 150 350; lineto 150 150;;
```

```
close_graph ();;
```

## Textes

- ▶ (`draw_char c`) affiche le caractère `c` à la position actuelle du curseur graphique ;
- ▶ (`draw_string s`) affiche la chaîne `s` à la position actuelle du curseur graphique ;
- ▶ (`set_text_size n`) devrait permettre de choisir une taille de police de caractère, mais ... ne marche pas.
- ▶ (`text_width s`) renvoie la paire (*largeur*, *hauteur*) de la chaîne `s` quand elle est affichée dans la fonte courante.



```

open Graphics;;
open_graph "□800x600";;
let pi = 3.1415927 ;;
let dessine (xo,yo) radius l =
  let rec des alpha i = function
    | [] -> ()
    | h::r ->
      moveto
        (xo - (int_of_float (cos(alpha *. i)*.radius)))
        (yo + (int_of_float (sin(alpha *. i)*.radius)));
      draw_char h;
      des alpha (i +. 1.) r
  in des (pi /. (float ((List.length l)-1))) 0. l
;;
dessine (300,200) 50. ['b';'o';'n';'j';'o';'u';'r'];;

```

## Couleurs

- ▶ Il y a un type `color` représentant les couleurs.
- ▶ Les constantes prédéfinies du type `color` sont `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan`, `magenta`.
- ▶ `(rgb r v b)` renvoie la couleur (du type `color`) avec les composantes rouges *r*, verte *v* et bleue *b*. Les valeurs légales pour les arguments sont de 0 à 255.
- ▶ `(set_color c)` sélectionne *c* comme la couleur courante ;
- ▶ `(fill_rect x y l h)` remplit le rectangle de largeur *l*, de hauteur *h* et de point inférieur gauche  $(x, y)$  par la couleur courante.

## Exemples (graphics4.ml)

```
open Graphics;;
```

```
open_graph "┐600x400";;
```

```
let shocking_pink = rgb 255 105 180;;
```

```
set_color shocking_pink;;
```

```
fill_rect 100 100 200 200;;
```

```
close_graph ();;
```

# Événements

- ▶ Un **événement** se produit quand l'utilisateur clique sur un bouton de la souris, déplace la souris ou presse une touche du clavier. Le type `event` contient les formes différentes des événements :

```
type event =
```

```
  Button_down | Button_up | Key_pressed | Mouse_motion ;
```

- ▶ La fonction `wait_next_event` prend comme argument une liste `l` d'événements et attend le prochain événement appartenant à la liste `l` (les autres événements seront ignorés). Quand le premier événement se produit une description détaillée est renvoyée, du type `status`.

## Le type status

```
type status =
{
mouse_x      : int;  (* coordonnée x de la souris          *)
mouse_y      : int;  (* coordonnée y de la souris          *)
button       : bool; (* un bouton de la souris est enfoncé ? *)
keypressed   : bool; (* une touche du clavier a été pressée ? *)
key          : char; (* touche pressée du clavier le cas échéant *)
}
```

Remarque : il n'y a aucune distinction entre les différents boutons de la souris.

```

open Graphics;;
open_graph "□500x500";;
exception Quit;;
let rec loop t =
    let ev = wait_next_event [Mouse_motion;Key_pressed] in
    if ev.keypressed
    then match ev.key with
        | 'b'  -> set_color black; loop t
        | 'r'  -> set_color red; loop t
        | 'g'  -> set_color green; loop t
        | 'q'  -> raise Quit
        | '0'..'9' as x ->
            loop (int_of_string (String.make 1 x))
        | _    -> loop t
    else (fill_circle ev.mouse_x ev.mouse_y t; loop t)
in
try loop 5 with Quit -> close_graph ();;
    
```