Programmation Fonctionnelle
Cours 7
Entrées, sorties, graphisme

November 7, 2012

Table de matières

Disponibilité

- L'interprète OCaml n'a par défaut pas les fonctionnalités graphiques. Il y a plusieurs possibilités :
  - Charger la bibliothèque dans l'interpréteur : `#load "graphics.cma;;"`
  - Inclure la bibliothèque au moment du 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 : voir le manuel (bibliothèque `gr phics`).

Programmation Fonctionnelle Cours 7 Entrées, sorties, graphisme
└Graphisme

La fenêtre graphique

- `Open Gr phics;;` met toutes les fonctions (types, exceptions) de cette bibliothèque disponible (on n'a plus besoin de la notation pointée)
- On peut avoir une (seule) fenêtre graphique, crée par `open_gr ph " lxh"` où *l* et *h* sont le nombre de pixels pour la largeur (l) et hauteur (h). Par exemple:

  `open_gr ph " 600x400"`

  Attention l'espace au début de l'argument est obligatoire. Ceci pour UNIX, voir le manuel pour le cas Windows.
- `close_gr ph: unit -> unit` ferme la fenêtre graphique.
- `cle r_gr ph: unit -> unit` efface le contenu de la fenêtre graphique.

Exemples (graphics1.ml)

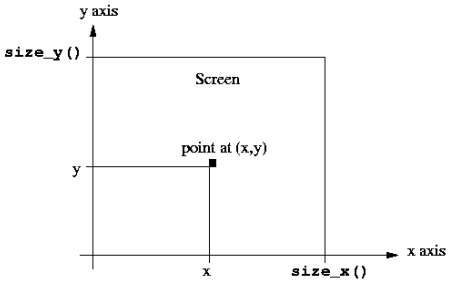
```
open Graphics;;

#load "graphics.cma";;

open_graph " 600x400";;

close_graph ();;
```


Coordonnés sur le canvas graphique



L'origine (0,0) est en bas à gauche

Dessiner

- Il y a un curseur, qui au debut 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 à (x, y) et positionne le curseur graphique en (x, y) ;
- `(set_line_width n)` sélectionne n comme épaisseur des lignes.

Exemples (graphics2.ml)

```
open Graphics;;

open_graph " 600x400";;

moveto 200 200;;
lineto 400 200; lineto 400 300; lineto 200 300; line

set_line_width 5;;
moveto 150 150;;
lineto 450 150; lineto 450 350; lineto 150 350; line

close_graph ();;
```

Textes

- `(dr w_ch r c)` affiche le caractère *c* à la position actuelle du curseur graphique ;
- `(dr w_string s)` affiche la chaîne *s* à la position actuelle du curseur graphique ;
- `(set_font_size n)` sélectionne *n* comme taille de fonte ;
- `(text_width s)` renvoie la paire (*largeur*, *hauteur*) de la chaîne *s* quand elle est affichée dans la fonte courante.

Programmation Fonctionnelle Cours 7 Entrées, sorties, graphisme
└Graphisme

```
open Graphics;;
open_graph " 600x400";;
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_of_int ((List.length l)-1))) 0. l

dessine (300,200) 50. ['b'; 'o'; 'n'; 'j'; 'o'; 'u'; 'r'
```

1d72 0.7 G 0 Td [(

nvoie la cofonte

color

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 `w it_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 `st tus`.

Le type `status`

```
type st tus =
{
mouse_x    : int;  (* coordonnée x de l  souris
mouse_y    : int;  (* coordonnée y de l  souris
button     : bool; (* vr i si un bouton de l  souris est en
keypressed : bool; (* vr i si une touche du cl vier   été p
key        : ch r; (* touche pressée du cl vier le c s éché
}
```

Les boutons différent de la souris ne sont pas dinstingués.


```
open Graphics;;
open_graph "␣500x500";;
exception Quit;;
let rec loop t =
  let eve = wait_next_event [Mouse_motion;Key_pressed]
  in
  if eve.keypressed
  then
    match eve.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.ma
      | _    -> loop t
  else begin
    fill_circle (eve.mouse_x-t/2) (eve.mouse_y-t/2) t;
    loop t
  end
in
try loop 5
with Quit -> close_graph ();;
```


Les canaux de communication vus de l'intérieur

- Les entrées et sorties d'un programme OCaml utilisent des canaux de communications (sauf bibliothèques spécialisées, comme pour 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 existent toujours (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 trois canaux de communication (voir un 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 rédirigée 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 rédirigée.

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. La fonction `print_newline` force la sortie du contenu du tampon de sortie.
- 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 il faut écrire une fonction pour le faire.

Les canaux de communication vus de l'intérieur

- Il y deux types définis dans la bibliothèque de noyaux qui correspondent aux canaux de communication :
  - `in_channel` pour les canaux d'entrée
  - `out_channel` pour les canaux sortie
- On peut créer un nouveau canal en l'associant (par des fonctions spécialisées) par exemple à un fichier, ou une connexion réseau.

Exemples (canaux.ml)

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

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 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.
- Le typage de cette fonction est « triché » (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.
- Merci de bien faire la distinction entre stdout et stderr car un utilisateur peut avoir besoin de séparer la sortie normale des messages d'erreur.
- Fonctions `prerr_int` etc. (voir le manuel)

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 un 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'execution *)
let c = open_out "/blurbel";;
```


Écrire vers un canal

- Fonctions `output_string`, `output_ch r` pour écrire dans un canal de sortie. Le premier argument est le canal.
- Il n'y a pas de `output_int`, en revanche il y a une variante de `printf` (du nom `fprintf`) pour écrire dans un canal.
- La sortie vers un canal est tamponnée.
- Fonctions `flush` et `flush_ ll` pour vider (tous) les tampons de sortie.

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'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;;
```


Entrée par *stdin*

- La fonction `re d_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 `re d_int` et `re d_flo t`.
- Le module `Sc nf` permet de lire des lignes dans un format précis (analogue à Printf).
- Pour faire une lecture plus flexible : voir le cours d'*Analyse Syntaxique* du deuxième semestre.
- Problème dans le cas du mode emacs : attend toujours ";;" pour terminer une entrée.

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
  and co = open_out outfile
  in
    copy_lines ci co;
    close_in ci;
    close_out co
;;
```


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.

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 bytes_this_line = String.length (input_line
    in bytes_this_line + count_bytes ci
  with
      End_of_file -> 0
;;

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