Université Paris Diderot — Programmation Fonctionnelle
Licence — Année 2010-2011

TP n°7 - Correction

Graphismes

Dans ce TP, on introduit le module `Graphics` de la librairie de OCaml. Pour plus de details, voir `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html`.

Voici des extraits de la documentation du module `Graphics` qui illustrent les primitives utilisées dans la solution.

- `open_graph : string -> unit`
  Show the graphics window or switch the screen to graphic mode. The graphics window is cleared and the current point is set to (0, 0). The string argument is used to pass optional information on the desired graphics mode, the graphics window size, and so on. Its interpretation is implementation-dependent. If the empty string is given, a sensible default is selected.
- `moveto : int -> int -> unit`
  Position the current point.
- `lineto : int -> int -> unit`
  Draw a line with endpoints the current point and the given point, and move the current point to the given point.

Pour pouvoir utiliser le mode intéractif, il est nécessaire de charger le module graphics.cma dans le toplevel OCaml. Il est possible de le charger de deux manières différentes :

- Lancer le toplevel OCaml avec comme argument graphics.cma
- Charger le module Graphics dans une session toplevel avec la directive toplevel suivante : `#load "graphics.cma";;`.

Pour pouvoir tester vos programmes, vous pourrez utiliser la fonction `Graphics.read_key` qui attend que l'utilisateur appuie sur une touche.

Exercice 1 [Fonctions trigonométriques] Afficher dans une fenêtre graphique le graphe de la fonction sinus entre 0 et 2π en rouge, et aussi du cosinus en bleu.

Correction :

```
#load "graphics.cma";; (* le dièse est nécessaire *)
open Graphics;;
open_graph " 600x600";; (* l'espace est nécessaire *)

let pi = 4. *. atan 1. ;;
let zoom_x = 50.;;
let zoom_y = 100.;;
let offset_x = 50;;
let offset_y = 200;;


(*
 * fn : fonction à dessiner (cos ou sin)
 * prec : precision
 * deb : debut d'où on commencer à tracer
 * fin : fin ...
 *)
```

```
let dessiner_trigo fn prec deb fin =

  let rec dessiner_trigo_aux fn prec deb x fin =
    if x < fin then
      begin
        let _x = int_of_float (zoom_x *. x) in
        let _y = int_of_float (zoom_y *. (fn x)) in
        let () = plot (_x + offset_x) (_y + offset_y) in
        dessiner_trigo_aux fn prec deb (x +. prec) fin
      end
    else
      ()
  in
  let fn = if fn = "cos" then cos else sin in
  let () =
    if fn = "cos" then
      set_color red
    else
      set_color blue
  in
  dessiner_trigo_aux fn prec deb deb fin
;;

dessiner_trigo "cos" 0.001 0.0 (2.*.pi);;
dessiner_trigo "sin" 0.001 0.0 (2.*.pi);;

read_key ()
```

Exercice 2 [La courbe du dragon]

- La courbe du dragon d'ordre 1 entre les points A et B est le segment $[AB]$;
- Soit C le point formant un triangle isocèle rectangle en C avec A et B, à droite de $\overrightarrow{AB}$. Si A et B sont de coordonnées respectivement (x, y) et (z, t), alors C est de coordonnées (u, v) avec :
  $u = (x + z)/2 + (t - y)/2$
  $v = (y + t)/2 - (z - x)/2$
  La courbe du dragon d'ordre n est :
  - La courbe du dragon d'ordre $n - 1$ entre A et C
  - La courbe du dragon d'ordre $n - 1$ entre B et C

Écrire un programme qui dessine la courbe du dragon.

Correction :

```
#load "graphics.cma";; (* le dièse est nécessaire *)
open Graphics;;
open_graph " 600x600";; (* l'espace est nécessaire *)
let rec dragon n x y z t =
  if n = 1 then begin
    moveto x y;
    lineto z t
  end
  else begin
    let u = (x + z + t - y)/ 2
    and v = (y + t - z + x)/ 2  in
    dragon (n-1) x y u v;
    dragon (n-1) z t u v
```

```
  end;;

dragon 15 200 200 400 400;;
```

1 Ens[illegible] [illegible]nd[illegible] rot

Exercice 3 Pour cette exercice, nous utiliserons le module et `Complex` de la bibliothèque standard OCaml. On rappelle juste qu'à chaque point du plan de coordonnées (x, y), on peut associer un nombre complexe noté $x+iy$. Pour représenter les complexes, la bibliothèque `Complex` définit le type :

```
type t = {
    re : float;
    im : float;
   }
```

ainsi que toutes les opérations de base (addition, multiplication...).

1. Commencez par ouvrir une fenêtre graphique de 500 par 500 et écrivez une fonction `remplit : unit -> unit` qui va remplir point par point cette fenêtre de rouge. On utilisera uniquement des fonctions récursives (pas de boucles). Le but étant de parcourir tout les points, il est interdit d'uliser les fonctions de remplissages du module `Graphics`.
2. Pour chaque point du plan d'affixe c, on définit la suite de Mandelbrot par :

$$\begin{aligned} c_0 &= c \\ c_{n+1} &= c_n^2 + c \end{aligned}$$

   L'ensemble de Mandelbrot est l'ensemble des points pour lesquels la norme des éléments de cette suite ne tend pas vers l'infini.
   On peut montrer que si la norme dépasse 2, alors la suite tend vers l'infini.
   On fera l'approximation suivante : si au bout de 50 itérations la norme est toujours inférieure à 2, alors le point est dans l'ensemble.
   Écrire une fonction
   `diverge : int -> Complex.t -> bool`
   telle que `diverge nb_iter c` renvoie `true` lorsqu'en moins de `nb_iter` itérations, la norme de `c` est plus grande que 2, et `false` sinon.
   Vérifier que la suite diverge en $(1, 1)$ mais pas en $(0, 0)$.
3. Soit `c : Complex.t` le point en bas à gauche du carré que nous voulons afficher et largeur : float sa largeur. La fonction suivante renvoie l'affixe correspondant au point de coordonnées (x, y) sur la fenêtre graphique 500x500 :

   ```
   let calcule_affixe c largeur x y =
     Complex.add c
         {re= (float_of_int x) *. largeur /. 500. ;
          im= (float_of_int y) *. largeur /. 500. }
   ```

   transformer la fonction `remplit` en une fonction
   `mandelbrot : int -> Complex.t -> float -> unit`
   qui prend en paramètres le nombre maximum d'itérations pour chaque point, `c` et `largeur`, et qui dessine l'ensemble de Mandelbrot.
   Calculer `mandelbrot 50 re= -1.5; im= -1. 2.`

4. Modifier le programme pour que la couleur soit choisie en fonction de la vitesse de divergence de la suite (nombre d'itérations pour dépasser 2).
5. L'ensemble de Julia pour une valeur complexe z se calcule comme l'ensemble de Mandelbrot en remplaçant la suite de Mandelbrot par :

$$\begin{aligned} c_0 &= c \\ c_{n+1} &= c_n^2 + z \end{aligned}$$

Dessiner l'ensemble de Julia, par exemple pour $z = 0,75i$

Correction :

```
#load "graphics.cma";;
open Graphics
open Complex

open_graph " 500x500";;

let remplit () =
  let rec dessine y =
    let rec trace_ligne x =
      plot x y;
      if x < 500
      then trace_ligne (x+1)
    in
    trace_ligne 0;
    if y < 500
    then dessine (y+1)
  in dessine 0;;


(*********************************************************************)
(* Première version *)
let diverge nbiter c =
  let rec aux n c_n =
    (n < nbiter) &&
    (Complex.norm c_n > 2.0 || (* Si supérieur à 2, ça diverge forcément *)
    aux (n+1) (Complex.add (Complex.mul c_n c_n) c))
  in aux 0 c;;

let mandelbrot nbiter c largeur =
  let calcule_complexe x y =
    Complex.add
      c
      {re= (float_of_int x) *. largeur /. 500. ;
       im= (float_of_int y) *. largeur /. 500. }
  in
  let rec dessine y =
    let rec trace_ligne x =
      let c = calcule_complexe x y in
      if diverge nbiter c
      then begin
        set_color red;
        plot x y;
      end
      else begin
```

```
        set_color black;
        plot x y
      end;
      if x < 500
      then trace_ligne (x+1)
    in
    trace_ligne 0;
    if y < 500
    then dessine (y+1)
  in dessine 0;;

mandelbrot 50 {re= -1.5; im= -1.} 2.;;

(*****************************************************************)
(* Avec jolies couleurs *)

let diverge nbiter c =
  let rec aux n c_n =
    if (n >= nbiter) ||
    (Complex.norm c_n > 2.0) (* Si supérieur à 2, ça diverge forcément *)
    then n
    else aux (n+1) (Complex.add (Complex.mul c_n c_n) c)
  in aux 0 c;;


let mandelbrot nbiter c largeur =
  let calcule_complexe x y =
    Complex.add
      c
      {re= (float_of_int x) *. largeur /. 500. ;
       im= (float_of_int y) *. largeur /. 500. }
  in
  let calcule_couleur n =
    rgb (256*n/nbiter) 0 0
  in
  let rec dessine y =
    let rec trace_ligne x =
      let c = calcule_complexe x y in
      let n = diverge nbiter c in
      set_color (calcule_couleur n);
      plot x y;
      if x < 500
      then trace_ligne (x+1)
    in
    trace_ligne 0;
    if y < 500
    then dessine (y+1)
  in dessine 0;;

mandelbrot 50 {re= -1.5; im= -1.} 2.;;

(*****************************************************************)
(* Julia *)
let diverge c0 nbiter c =
  let rec aux n c_n =
    if (n >= nbiter) ||
```

```
    (Complex.norm c_n > 2.0) (* Si supérieur à 2, ça diverge forcément *)
    then n
    else aux (n+1) (Complex.add (Complex.mul c_n c_n) c0)
  in aux 0 c;;


let julia nbiter c0 c largeur =
  let calcule_complexe x y =
    Complex.add
      c
      {re= (float_of_int x) *. largeur /. 500. ;
       im= (float_of_int y) *. largeur /. 500. }
  in
  let calcule_couleur n =
    rgb (256*n/nbiter) 0 0
  in
  let rec dessine y =
    let rec trace_ligne x =
      let c = calcule_complexe x y in
      let n = diverge c0 nbiter c in
      set_color (calcule_couleur n);
      plot x y;
      if x < 500
      then trace_ligne (x+1)
    in
    trace_ligne 0;
    if y < 500
    then dessine (y+1)
  in dessine 0;;

julia 60 {re= 0.; im= 0.75} {re= -1.5; im= -1.5} 3.;;
```