TP5 MPSI: Bibliothèque graphique de Caml Light

Alexis Saurin
Alexis.Saurin@ens.fr
http://www.eleves.ens.fr/~saurin/tp_caml

29 avril 2003

1 Introduction

Caml Light dispose de fonctions graphiques qui vont permettre de tracer des dessins :
- `open_graph : string -> unit`,
- `clear_graph : unit -> unit`,
- `close_graph : unit -> unit`,
- `plot : int -> int -> unit`,
- `moveto : int -> int -> unit`,
- `lineto : int -> int -> unit`,
- `draw_char : char -> unit`,
- `draw_string : string -> unit`,
- `set_color : color -> unit`...

On va apprendre à se servir de ces fonctions pour deux applications ici : la première va consister à écrire un programme pour tracer des courbes représentatives de fonctions, comme sur une calculette graphique, la seconde sera la programmation du jeu de la vie, un cas particulier d'automates cellulaires.

Question 1: *Ouverture d'une fenêtre graphique*

Essayez d'ouvrir une fenêtre graphique avec la fonction `open_graph`. Cette fonction prend en argument une chaîne de caractères et ouvre une fenêtre graphique ; la chaîne de caractères peut soit être la chaîne vide, soit une chaîne de la forme `"300x450"` qui spécifie ainsi les dimensions de la fenêtre à ouvrir.

Question 2: *Bibliothèque Graphics*

Ce que vous avez tapé à la question précédente n'a pas fonctionné et a dû vous retourner un message d'erreur du type :

```
#open_graph "";;
Toplevel input:
>open_graph "";;
>^^^^^^^^^^
The value identifier open_graph is unbound.
```

La raison de cette erreur est que les fonctions graphiques ne font pas partie de la bibliothèque standard le Caml Light. Il faut d'abord charger cette bibliothèque de fonctions pour que les fonctions graphiques soient disponibles et connues de l'interpréteur.

Vous pouvez charger cette bibliothèque au moyen de l'expression Caml Light suivante :

```
#open "graphics" ;;
```

Question 3: *Premiers essais*

Faîtes quelques essais : ouvrez des fenêtres de différentes tailles, tracez une droite entre le point à la position `(30, 30)` et le point `(45, 45)`, déplacez le curseur à un autre point et tracer une droite vers `(30, 30)`...

Pour trouver d'autres éléments concernant la bibliothèque graphique, allez voir le manuel de Caml Light.

2 Courbes représentatives de fonctions

On va maintenant écrire un programme qui va nous permettre de tracer des courbes dans un repère, comme le ferait votre calculatrice graphique.

Question 4: *Construction du repère* (∗)

Construire une fonction `repere : int -> int -> unit` qui prend deux entiers correspondant à la taille de la fenêtre, et qui trace un repère orthogonal. Pour tracer la droite des ordonnées, on fera comme suit :

```
let s="ordonnées" in
moveto (longueur/2) 0;
lineto (longueur/2) hauteur;
for i=0 to string_length s -1 do
   moveto  (longueur/2 -20)
           (hauteur -10  - 15*i);
   draw_char s.[i];
done;
```

Pour les abscisses, on pourra utiliser la fonction `draw_string : string -> unit`.

Question 5: *Tracer une courbe représentative* (∗∗)

Définir une fonction `trace_courbe : (float -> float) -> int -> int -> unit` qui trace la courbe correspondant à la fonction passée en argument, les deux entiers servent à définir l'échelle en abscisse et en ordonnée.

Question 6: *Et avec un domaine de définition ?* (∗∗)

Comment faire si on voulait tracer la courbe d'une fonction qui n'est pas définie sur tout son ensemble de départ ? On peut imaginer de mettre une condition sur l'argument de la fonction qui constituera un test avant l'utilisation de la fonction `plot` dans `trace_courbe`, ou bien une liste de valeurs interdites.

En fait, cela dépend des problèmes que l'on considère. On va ici considérer la condition sous la forme d'une fonction de float vers bool. (et on pourra construire une fonction qui transforme une liste de valeurs interdites en fonction de float vers bool).

Question 7: *Avec des couleurs*

Sur un ordinateur avec un écran couleur, on peut tracer les courbes de différentes couleurs pour les rendre plus lisibles.

Pour changer la couleur du trait, on fait par exemple `set_color blue ;;`. Ce qui sera tracé ensuite sera en bleu.

3 Automates cellulaires et le jeu de la vie

Le jeu de la vie se fonde sur l'automate de Conway qui est un cas particulier d'automates cellulaires, sans doute le plus connu de tous. Il a été inventé par John Conway en 1970. Le jeu de la vie a eu une telle popularité à cette époque que l'on a évalué à plusieurs millions de dollars le temps de calcul emprunté par les adeptes de ce jeu aux ordinateurs de leur entreprise pour y jouer...

Le principe des automates cellulaires est en gros d'avoir une ensemble de cellules disposées en damier, qui peuvent prendre un certain nombre (fini) d'états, et dont l'évolution est dictée par une règle fixée une fois pour toute (qui fait donc partie de la définition de l'automate) et qui est fonction de l'état de la cellule ainsi que de ceux de ses voisines.

Si le jeu s'appelle "jeu de la vie", c'est que l'évolution de l'automate de Conway ressemble à celle d'une population de bactérie : les cellules peuvent être dans deux états 0 (mort) ou 1 (vivant) et selon les états des huit cellules les plus proches d'elle, son état est modifié : si une cellule est dans l'état mort et qu'elle est entourée d'exactement trois cellules vivantes, elle passe dans l'état 1, c'est la naissance. Si une cellule est dans l'état 1 et est entourée de deux ou trois cellules vivantes, la cellule reste dans cet état à la génération suivante, c'est la survie. Enfin, si la cellule est vivante et est entourée soit de strictement moins de deux cellules vivantes, soit de strictement plus de trois cellules vivantes, elle passe dans l'état 0, c'est la mort par isolement ou par étouffement.

On va, dans cette partie, programmer le jeu de la vie et utiliser l'interface graphique de Caml Light pour afficher l'évolution du jeu de la vie. Dans un premier temps, on va définir quelques fonctions qui nous serviront pour le jeu, ensuite, on définira la condition de naissance/survie/mort d'une génération à une autre, puis la fonction de transition d'une génération à une autre et enfin la fonction finale, celle du jeu. On finira en regardant le comportement du jeu sur quelques exemples, puis en regardant comment on peut utiliser la structure de notre programme pour définir de nouveaux automates cellulaires.

Question 8: *Représentation informatique du jeu* (∗)

On va représenter une configuration du jeu comme une matrice carré d'entiers, tout simplement. Définissez une fonction `creer_matrice : int -> int -> (int vect vect)` qui crée une matrice remplie de 0, dont les nombres de lignes et colonnes sont indiqués par les arguments de la fonction. Faîtes bien attention à la représentation en mémoire des tableaux pour éviter les surprises...

Pour éviter des problèmes de conditions de bord, on va en fait travailler sur un tore, c'est-à-dire que l'on va travailler modulo la taille de notre matrice (la ligne suivant la dernière ligne est la première...). Définissez une fonction `modulo : int -> int -> int` qui calcule le modulo de son premier argument par rapport au deuxième (la fonction mod de Caml Light calcule en fait le reste de la division euclidienne, ce qui est gênant pour les nombres négatifs...).

Comme on voudra étudier des exemples d'évolution, il nous faudra une fonction pour créer un espace de cellules initiales. Définissez une fonction `initialise : (int*int) list -> int -> int vect vect` qui crée une matrice remplie de 0 partout sauf aux cases indiquées par la liste, celles-ci contiennent des 1, elle prend aussi en paramètres la taille de la matrice.

On aura besoin de faire une copie d'une matrice dans une autre, définissez une fonction `copier : int vect vect -> int vect vect -> unit` qui copie le contenu de son second argument dans le premier.

On affichera un état du jeu dans une fenêtre graphique de Caml Light, chaque case sera représentée par un certain nombre de pixels (sinon on n'y voit rien...). Définissez une fonction `affiche : int vect vect -> int -> unit` qui affiche dans la fenêtre graphique le contenu de la matrice (rien pour 0 et un point pour 1) chaque point étant un carré d'un certain nombre de pixels de large, nombre qui est spécifié en argument de la fonction.

Question 9: *vie et mort* (∗∗)

Implémentez la condition indiquée dans l'introduction de l'exercice : on veut une fonction `vie_mort : int vect vect -> int -> int -> int` qui prenne une matrice M, deux indices i et j et qui retourne soit 0 soit 1 selon qu'à la génération suivante la cellule `m.(i).(j)` vaut 0 ou 1.

Question 10: *transition* (∗∗)

Définissez une fonction `transition : int vect vect -> (int vect vect ->`

`int -> int -> int) -> unit` qui prend une matrice en entrée et une fonction de vie/mort et qui calcule la génération suivante du jeu.

Question 11: *pour finir* (∗)

Définissez une fonction `jeu : (int*int) list -> int -> int -> unit` qui comporte une boucle infinie et qui affiche l'évolution du jeu, étant donné un ensemble de cases noires données par une liste en argument.

On pourrait aussi définir une fonction qui calcule les N premières transitions du jeu de la vie, pour N passé en argument.

Question 12: *des exemples* (∗)

Étudiez le comportement du jeu de la vie avec les structures suivantes :

```
let carre= [(2, 11);
(2, 12); (3, 11); (3, 12)];;

let croix= [(10, 10);
(11, 10); (11, 9); (12, 10)];;

let glisseur= [(3, 15); (4, 15);
(4, 17); (5, 15); (5, 16)];;

let lance_glisseur=
[(2, 11); (2, 12); (3, 11);
(3, 12); (13, 10); (13, 11);
(13, 12); (14, 9); (14, 13);
(15, 8); (15, 14); (16, 9);
(16, 13); (17, 10); (17, 11);
(17, 12); (18, 10); (18, 11);
(18, 12); (23, 12); (23, 13);
(23, 14); (24, 11); (24, 12);
(24, 14); (24, 15); (25, 11);
(25, 12); (25, 14); (25, 15);
(26, 11); (26, 12); (26, 13);
(26, 14); (26, 15); (27, 10);
(27, 11); (27, 15); (27, 16);
(32, 11); (32, 12); (36, 13);
(36, 14); (37, 13); (37, 14)];;

let couple_glisseur=
[(3, 15); (4, 15);
(4, 17); (5, 15); (5, 16);
(7, 15); (7, 14); (8, 14);
(8, 16); (9, 14)];;

let pentomino=
[(3, 15); (4, 13); (4, 14);
(4, 15); (5, 14)];;
```

Question 13: *des améliorations ?* (∗ ∗ ∗)

Le programme tel qu'il est fait ici consiste à mettre à jour toutes les cases de la matrice, et à tout réafficher. Pourtant la matrice comporte de grandes parties vides, en particulier, les cellules susceptibles de devenir noires sont nécessairement des cellules voisines de cellules déjà noires. Comment modifier la fonction de transition pour que l'on tienne compte de cette remarque ? Par ailleurs, on n'est pas forcé de reconsidérer toutes les cases pour l'affichage, on pourrait ne considérer que les cases à modifier : cases blanches qui devenant noires et l'inverse.

Question 14: *un cas plus général d'automate* (∗∗)

Dans la fonction de transition, on a pris en argument une fonction qui était la règle de transition à appliquer dans chaque cas (ie la fonction vie/mort). Définir une nouvelle fonction `automate_cellulaire : (int* int) list -> int -> int -> (int vect vect -> int -> int -> int) -> unit` dérivée de la fonction jeu mais qui prend en argument la règle de transition. L'utiliser pour définir un automate qui consiste à faire un déplacement vers la droite.