▶ **MPSI – Option Informatique**
Année 2001, Neuvième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Labyrinthes

1 Représentation des labyrinthes

Les labyrinthes que nous étudierons aujourd'hui prendront place sur une grille. Chaque case de la grille pourra représenter soit un morceau de mur, soit une partie de chemin.

La première chose à faire est de numéroter les lignes (de haut en bas et de 0 à $n-1$) ainsi que les colonnes (de gauche à droite et de 0 à $p-1$). On peut ensuite représenter le labyrinthe dans une matrice d'entiers *laby* (i.e. un tableau de tableau d'entiers de type *int vect vect*) telle que $laby.(i).(j) = 0$ si la case (i, j) est un mur et 1 si cette case est un chemin.

▶ **Question 1** *Définissez la matrice correspondant au labyrinthe suivant :*

Nous allons maintenant écrire une fonction permettant d'afficher sur la sortie standard de *Caml* un labyrinthe. Les murs seront représentés par exemple par le caractère # tandis que les chemins seront représentés par un caractère d'espacement.

▶ **Question 2** *Écrivez une fonction* print_case *qui prend pour argument un entier décrivant une case (*0 *ou* 1*) et imprime le caractère (*# *ou espace) qui lui correspond.*

value print_case : *int* → *unit*

▶ **Question 3** *Déduisez-en une fonction* print_laby *permettant d'afficher un labyrinthe.*

value print_laby : *int vect vect* → *unit*

2 Trouver la sortie !

Un problème intéressant, lorsqu'on s'intéresse à de tels labyrinthes, est de concevoir un algorithme qui permette à partir d'une position de départ (i_d, j_d), de rejoindre une arrivée (i_a, j_a). Nous ne ferons pas d'hypothèse particulière sur le placement de ces cases qui pourront se trouver à n'importe quel emplacement de la grille.
Le but de cette partie est d'implanter un algorithme de parcours *en profondeur* d'un labyrinthe, sans utiliser beaucoup de ressources. Le principe de l'algorithme est le suivant : lorsqu'on arrive sur une case, on commence par la marquer comme vue (pour être sûr de ne jamais s'y aventurer à nouveau) puis on se rend sur l'une de ses voisines. Si à une étape, tous nos voisins sont des murs ou ont déjà été vus, on revient sur nos pas jusqu'à trouver des cases non encore explorées.

2.1 Gestion d'une pile

Une pile est une suite finie de données de même type munie des deux opérations suivantes : *empiler* (*push* en anglais) qui consiste à ajouter une donnée au sommet de la pile et *dépiler* (*pop* en anglais) qui consiste à lire l'élément situé au sommet de la pile et à le retirer. L'idée importante à propos des piles est que l'on retire les éléments dans l'ordre inverse de leur insertion. Nous représenterons une pile en *Caml* par une référence sur une liste.

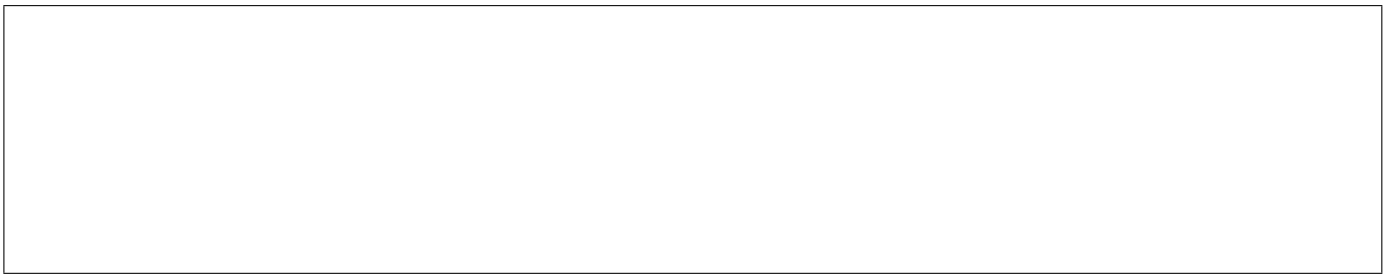

Fig. 1: Exemple de parcours

▶ **Question 4 (Push)** *Écrivez une fonction* push *qui prend pour arguments une pile p et un élément x et qui empile x sur p.*

value push : *'a list ref → 'a → unit*

▶ **Question 5 (Pop)** *Écrivez une fonction* pop *prenant pour argument une pile p et dépile l'élément situé au sommet de p. Votre fonction lèvera une exception si la pile est vide.*

value pop : *'a list ref → 'a*

2.2 Petites fonctions intermédiaires

Dans la suite, on dit que deux cases est voisines si et seulement si elles ont un côté commun. Une case qui n'est pas située sur le bord d'un labyrinthe admet donc exactement quatre voisines.

▶ **Question 6** *Écrivez une fonction* a_un_voisin *telle que* a_un_voisin laby (i , j) *retourne un booléen indiquant si la case (i, j) possède une voisine qui soit un chemin. (On pourra supposer que la case (i, j) n'est pas situé sur un bord du labyrinthe.)*

▶ **Question 7** *Écrivez une fonction* voisin *telle que* voisin laby (i , j) *retourne les coordonnées d'une case voisine de (i, j) (On pourra supposer que la case (i, j) a effectivement un voisin qui n'est pas un mur n'est pas situé sur un bord du labyrinthe.)*

2.3 Parcours en profondeur

Nous allons écrire une fonction parcours qui prend pour arguments la matrice *laby*, les coordonnées des points de départ (i_d, j_d) et d'arrivée (i_a, j_a) et trouve un chemin entre ces deux cases. Pour mener à bien notre parcours, nous aurons besoin de définir localement :

– Une référence *pos* sur un couple d'entiers indiquant la case où l'on se trouve.
– Une pile p – notre fil d'Ariane – contenant le chemin qui nous a mené de (i_d, j_d) à $!pos$.

Lors du parcours, vous « murerez » les cases visitées en modifiant l'entier correspondant dans la matrice (par exemple, en remplaçant 1 par -1). La fonction de parcours pourra alors procéder de la manière suivante : tant que la case courante n'est pas (i_a, j_a), on regarde si elle possède une voisine accessible. Si c'est le cas on y avance, sinon on revient sur ses pas. Un exemple de tel parcours est donné sur la figure 1.

▶ **Question 8** *Écrivez une fonction* parcours *qui étant données deux cases (i_d, j_d) et (i_a, j_a), retourne un chemin joignant ces deux points (sous forme d'une liste). Votre fonction lèvera une exception si un tel chemin n'existe pas.*

▶ **Question 9** *Voyez-vous des simplifications possibles de l'algorithme si on fait l'une des hypothèses suivantes :* « le labyrinthe ne comporte pas de cycle » *ou* « le départ et l'arrivée sont situés sur le bord ».

3 Génération de labyrinthes

Nous pouvons maintenant nous intéresser à un problème plus délicat : celui de la génération « aléatoire » de labyrinthes. Il n'existe pas pour ce problème de solution « canonique ». Voici une possibilité permettant d'engendrer des grilles carrées de taille $2^k + 1$:

– On découpe la grille en quatre zones carrées de même taille.
– Dans trois des quatre murs de séparation (choisis aléatoirement), on creuse un trou à une position impaire.
– On recommence récursivement pour chacun des quatre carrés.

▶ **Question 10** *Écrivez une fonction* make_laby *prenant pour argument un entier k et retournant un labyrinthe carré de taille $2^k + 1$. Vous pourrez écrire une sous-fonction récursive prenant pour arguments trois entiers i_0, j_0 et m et dont la finalité est de remplir la portion carrée de la grille d'angle inférieur gauche (i_0, j_0) et de côté $2^m - 1$ cases.*

▶ **MPSI – Option Informatique**
Année 2001, Neuvième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Labyrinthes

Un corrigé

▶ Question 2

```
let print_case = function
    0 -> print_char '#'
  | 1 | -1 -> print_char ' '
  | 2 -> print_char '*'
  | _ -> invalid_arg "print_case"
;;
```

▶ Question 3

```
let print_laby laby =
  for i = 0 to vect_length laby - 1 do
    for j = 0 to vect_length laby.(i) - 1 do
      print_case laby.(i).(j)
    done;
    print_newline ()
  done;
  ()
;;
```

▶ Question 4

```
let push p x =
  p := x :: !p
;;
```

▶ Question 5

```
let pop p =
  match !p with
    [] -> invalid_arg "pop"
  | hd :: tl ->
    p := tl;
    hd
;;
```

▶ Question 6

```
let a_un_voisin laby (i, j) =
  (laby.(i-1).(j) = 1)
  or (laby.(i+1).(j) = 1)
  or (laby.(i).(j-1) = 1)
  or (laby.(i).(j+1) = 1)
;;
```

▶ Question 7

```
let voisin laby (i, j) =
  if laby.(i-1).(j) = 1 then (i-1, j)
  else if laby.(i+1).(j) = 1 then (i+1, j)
  else if laby.(i).(j-1) = 1 then (i, j-1)
  else (i, j+1)
;;
```

▶ Question 8

```
let parcours laby depart arrivee =
  let pos = ref depart in
  let pile = ref [] in
  laby.(fst !pos).(snd !pos) <- -1;
  while !pos <> arrivee do
    print_pair !pos;
    if a_un_voisin laby !pos then begin
      push pile !pos;
      pos := voisin laby !pos;
      laby.(fst !pos).(snd !pos) <- -1
    end
    else pos := pop pile
  done;
  rev (arrivee :: !pile)
;;
```

Nous définissons également une fonction trace. Cette fonction prend pour arguments un labyrinthe et un chemin, et « trace » le chemin dans la matrice du labyrinthe en mettant un 2 dans chacune des cases traversées.

```
let rec trace laby = function
    [] -> ()
  | (i, j) :: tl ->
    laby.(i).(j) <- 2;
    trace laby tl
;;
```

Enfin, nous écrivons une fonction clean qui redonne à une matrice de labyrinthe sa forme originelle (après application de parcours et/ou trace).

```
let clean laby =
  for i = 0 to vect_length laby - 1 do
    for j = 0 to vect_length laby.(0) - 1 do
      if laby.(i).(j) <> 0 then
        laby.(i).(j) <- 1
    done;
  done
;;
```

▶ Question 10

```
let rec prefix ** x = function
    0 -> 1
  | n -> x * (x ** (n - 1))
;;

let make_laby k =

  let n = 2 ** k + 1 in
  let laby = make_matrix n n 1 in

  let rec def (i0, j0) m=
    if m >= 2 then begin
      let m2 = 2 ** m - 1 in

      (* On trace la croix centrale *)
      for i = i0 to i0 + m2 - 1 do
        laby.(i).(j0 + m2/2) <- 0
      done;
      for j = j0 to j0 + m2 - 1 do
        laby.(i0 + m2/2).(j) <- 0
      done;

      (* On fait trois trous dans cette croix *)
      let a = random__int 4 in
      let r () = (random__int (m2/4 + 1)) * 2 in
      if a <> 0 then laby.(i0 + m2/2).(j0 + r()) <- 1;
      if a <> 1 then laby.(i0 + m2/2).(j0 + m2/2 + 1 + r()) <- 1;
      if a <> 2 then laby.(i0 + r()).(j0 + m2/2) <- 1;
      if a <> 3 then laby.(i0 + m2/2 + 1 + r()).(j0 + m2/2) <- 1;

      (* Appels recursifs dans les sous-carres *)
      def (i0, j0) (m - 1);
      def (i0 + m2/2 + 1, j0) (m - 1);
      def (i0, j0 + m2/2 + 1) (m - 1);
      def (i0 + m2/2 + 1, j0 + m2/2 + 1) (m - 1);
    end
  in

  def (1,1) k;

  (* Trace du bord *)
  for x = 0 to n - 1 do
    laby.(x).(0) <- 0; laby.(x).(n-1) <- 0;
    laby.(0).(x) <- 0; laby.(n-1).(x) <- 0
  done;

  laby
;;
```