Programmation impérative et fonctionnelle avec OCaml

TP 2

Objectifs :
Fonctions récursives :

- parcours de structure circulaire (cf. graphes) en profondeur d'abord ;
- récursion directe ;
- maintien d'état global en place.

Labyrinthe

On souhaite écrire un algorithme pour pouvoir sortir d'un labyrinthe rectangulaire. Le labyrinthe de hauteur h et de largeur w peut être divisé en $h \times w$ cellules carrées de même taille. Chaque cellule est soit **libre** soit occupée par un **mur** ; une cellule libre qui se trouve sur le *bord* du labyrinthe est considérée comme une **sortie**. On démarre à une position donnée, spécifiée par ses coordonnées, et on cherche un chemin permettant de rejoindre une sortie.

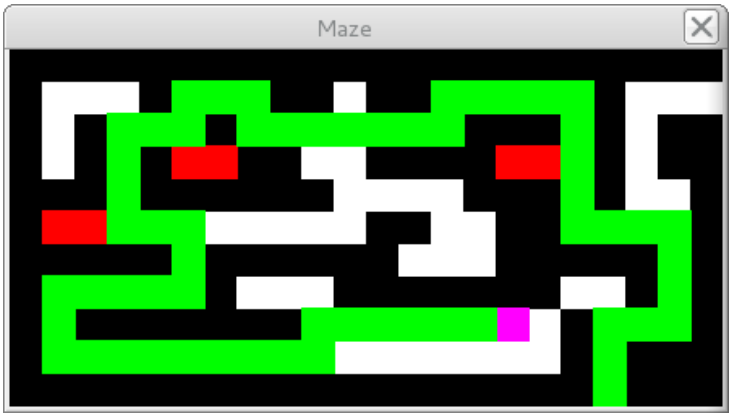

FIGURE 1 – Une solution du labyrinthe spécifié dans `maze1.txt`.

Algorithme On utilisera l'algorithme *récursif* suivant pour résoudre ce problème :

- on essaie d'abord de se déplacer d'une case vers l'**ouest**, puis, si c'est possible, on relancera l'algorithme **récursivement** à partir de la nouvelle position ;
- si on n'a pas trouvé de chemin menant à une sortie à l'étape précédente, on essaiera d'aller à l'**est** (et on répète l'algorithme récursivement) ;
- sinon on essaiera d'aller au **sud** (et...) ;
- enfin, si on n'a trouvé aucun chemin avec les tentatives précédentes, on essaiera au **nord** ;
- si **aucune** des directions possibles ne permet de trouver un chemin, on ne peut pas sortir du labyrinthe à partir de cette position (*dead end*).

Pour **éviter de boucler** indéfiniment si plusieurs chemins mènent à la même cellule, on **marquera** chaque cellule visitée et on veillera à ne pas explorer plusieurs fois une même cellule. De plus on marquera les cellules qui ne mènent à aucune issue pour ne pas les revisiter non plus (mais pouvoir les distinguer des précédentes).

Ainsi, on obtient les **conditions d'arrêt** de l'algorithme récursif :
— si la cellule à considérer est marquée comme :
  — occupée par un mur (`Wall`),
  — déjà explorée (`Marked`),
  — ou sans issue (`Dead`),
  on renverra `false` ;
— si la cellule correspond à une sortie (`Exit`), on marquera la cellule comme faisant partie du chemin solution (`Path`) et on renverra `true`.

Sinon, c'est-à-dire si la cellule est libre (`Free`), on la marque comme étant explorée (`Marked`) et on essaie récursivement les quatre directions possibles. Si l'une de ces tentatives renvoie `true`, on marque la cellule comme faisant partie de la solution (`Path`) et on renvoie `true` ; sinon (i.e. toutes les directions renvoient `false`), on marque la cellule comme sans issue (`Dead`) et on renvoie `false`.

Visualisation La figure 1 montre une fenêtre graphique à la fin de la résolution d'un labyrinthe où les différentes couleurs signifient :
— noir : mur (`Wall`) ;
— blanc : libre (`Free`) ;
— magenta : position de départ ;
— rouge : sans issue (`Dead`) ;
— vert : solution (`Path`).
Le bleu est utilisé durant la résolution pour indiquer les cellules déjà explorées (`Marked`) mais non déterminées.

Données Deux instances de labyrinthe sont décrites dans les fichiers texte `maze1.txt` et `maze2.txt` sous la forme suivante :
— un espace correspond à une cellule libre ;
— un astérisque (`'*'`) correspond à un mur ;
— le caractères `'S'` correspond à la position de départ.

Code fourni Pour écrire cet algorithme, vous devrez consulter le fichier d'interface documenté `maze.mli` et utiliser les fichiers précompilés `maze.cmi`, et `maze.cmo` (bytecode) ou `maze.cmx` (code natif) qui fournissent les fonctions utiles à la lecture des données, la mise à jour des cellules et l'affichage graphique. Si le fichier source de l'algorithme de résolution s'appelle `tp2.ml`, on compilera avec l'une des commandes suivantes :
— bytecode : `ocamlc -o tp2.out unix.cma graphics.cma maze.cmo tp2.ml`
— code natif : `ocamlopt -o tp2.opt unix.cmxa graphics.cmxa maze.cmx tp2.ml`

Résolution

1. Écrire dans un nouveau fichier l'équivalent du `main` (i.e. `let () = ...`) : on prendra le fichier de données en paramètre sur la ligne de commande en utilisant `Sys.argv.(1)`, puis on le lira avec la fonction `Maze.read` et on initialisera l'affichage graphique avec la fonction `Maze.init`.
2. Écrire ensuite la fonction récursive de résolution qui prend le labyrinthe (renvoyé par `Maze.read`) en paramètre et doit renvoyer `true` si on parvient à trouver une issue et `false` sinon. On utilisera les fonctions `mark`, `dead` et `path` du module `Maze` pour mettre

à jour le labyrinthe.
Indications :

- *On utilisera une fonction récursive locale qui prend en paramètres* `x` *et* `y`*, les coordonnées de la cellule à explorer ; on initialisera avec les coordonnées du départ renvoyées par* `Maze.start`*.*
- *On obtiendra le type d'une cellule avec la fonction* `Maze.get`*.*
- *Pour visualiser la résolution, on peut insérer une pause avant chaque étape grâce à la fonction* `Maze.sleep`*.*
- *Pour que la fenêtre graphique ne se ferme pas automatiquement en fin de résolution, on peut attendre qu'une touche soit pressée avec la fonction* `Maze.wait_key`*.*