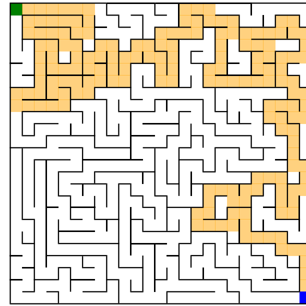
université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

Projet IPF
LDD2 IM

Projet : Labyrinthe



1 Objectifs

Le projet a plusieurs buts :

- Découvrir des algorithmes intéressants, assez subtils, et fondamentaux sur les graphes ;
- Placer la programmation fonctionnelle dans le cadre plus général de la programmation et donc mélanger du code fonctionnel avec du code faisant des effets de bords (en particulier utilisation de tableaux et entrées-sorties dans des fichiers) ou manipulant des tableaux ;
- Acquérir quelques compétences de *génie logiciel* en particulier de programmation modulaire et collaborative dans le cadre d'un projet écrit en OCaml ;

2 Sujet

On souhaite écrire un programme permettant de manipuler des labyrinthes. Le programe `maze.exe` que l'on demande d'écrire doit posséder l'interface suivante (dans les lignes de commandes ci-dessous, un argument `<foo>` est obligatoirement présent alors qu'un argument `[bar]` est optionne) :

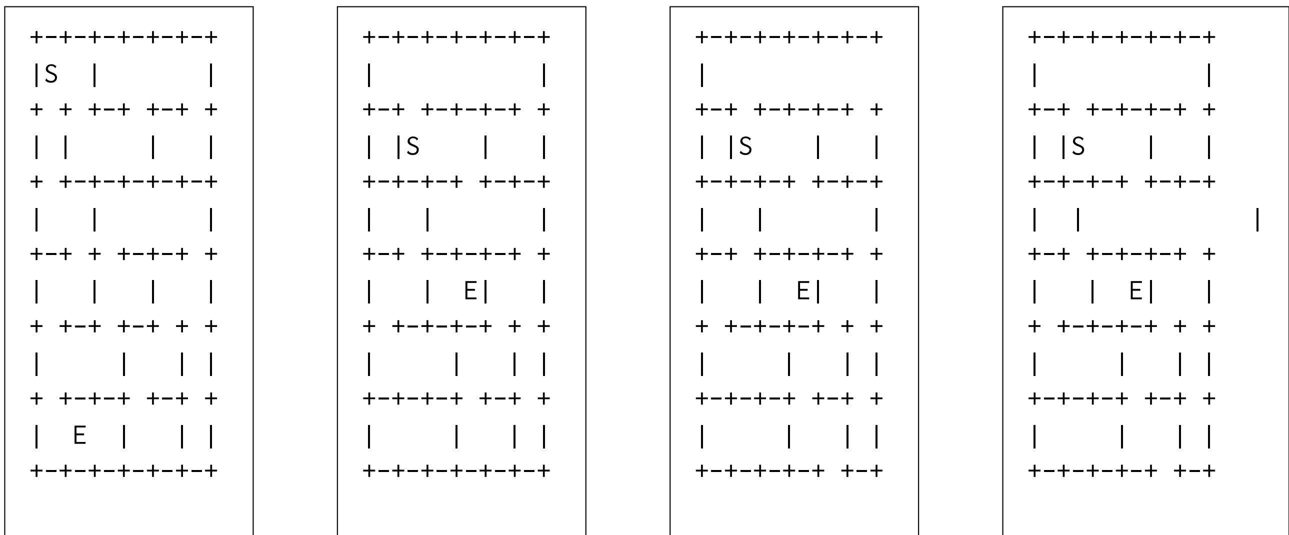
- `maze.exe --help` : affiche un message d'aide sur les différentes options
- `maze.exe print <fichier.laby>` : lit le labyrinthe se trouvant dans le fichier `fichier.laby` et l'affiche dans la console
- `maze.exe solve <fichier.laby>` : lit le labyrinthe se trouvant dans le fichier `fichier.laby` et l'affiche dans la console en mettant en évidence le chemin du départ vers l'arrivée
- `maze.exe random <n> <m> [r]` : génère un labyrinthe aléatoire de hauteur `n` et de largeur `m`, utilisant éventuellement l'entier positif `r` comme graine initiale pour le générateur de nombres aléatoires.

D'autres fonctionnalités vous seront demandées en fin de projet dans un esprit *agile* (ajouter en peu de temps une fonctionnalité sur un projet existant).

2.1 Présentation des labyrinthes

Un labyrinthe est une grille N lignes et M colonnes. Chaque case de la grille représente une pièce. Les pièce peuvent être séparées ou non par des murs. Les murs du contour du labyrinthe sont systématiquement présents. Un labyrinthe possède deux pièces particulières, la pièce de départ et la pièce d'arrivée, distinctes.

Les labyrinthes sont représentées de façon textuelles par des fichiers en ASCII-art. Des exemples de tels fichiers sont données à la figure 1. Le fichier 1.(a) respecte toutes les contraintes, de même que le fichier 1.(b). Le fichier 1.(c) est invalide car des murs sont manquant sur les bords par endroits. Le fichier 1.(d) est invalide car ne représente pas une grille (la troisième ligne est plus longue). Parmi les autres erreurs de syntaxe, on peut citer :



(a) Fichier valide (b) Fichier valide (c) Fichier invalide (d) Fichier invalide

FIGURE 1 – Exemples de fichiers valides et invalides contenant des labyrinthes

— la présence de caractères autres que `|`, `-`, `+`, `S`, `E`, l'espace et le retour charriot
— l'absence de point de départ ou de point d'arrivée
— le mauvais alignement des caractères autorisés (par exemple `|` à la place d'une pièce).

Votre programme devra s'arrêter avec un message d'erreur s'il détecte qu'un fichier d'entrée est invalide.

On peut par contre remarquer que le fichier 1.(b) est bien valide même s'il est particulier : l'arrivée (`E`) étant entourée de murs, il n'existe pas de chemin depuis la case de départ (`S`).

2.2 Compléments sur le langage OCaml

2.2.1 Tableaux

L'utilisation de listes (le seul type de collection vu dans le cours d'IPF) pour certains aspects de ce projet peut être problématique. On autorise donc l'utilisation de tableaux OCaml. Ces derniers sont semblables aux tableaux du C

— de taille fixe
— dont toutes les cases contiennent des valeurs du même type
— modifiables en place

La fonction **`Array.make n v`** permet de créer un tableau de **`n`** cases contenant toutes la valeur **`v`**. On peut accéder à la **`i`**ème case d'un tableau **`tab`** avec la notation **`tab.(i)`** (les indices commencent à 0). Si on souhaite mettre à jour la **`i`**ème case, on pourra écrire **`tab.(i) <-v`** Enfin, le module **`Array`** est similaire au module **`List`** et contient toutes les fonctions utilitaires sur les tableaux (par exemple **`Array.length`** ou **`Array.fold_left`**).

2.2.2 Chaînes de caractères

On peut accéder au **`i`**ème caractère d'une chaîne **`str`** avec la notation **`str.[i]`**. Le module **`String`** contient des fonctions utilitaires sur les chaînes. Attention, les chaînes de caractères ne sont pas modifiables, on **ne peut pas écrire** « **`str.[i] <-c`** ». Les éléments d'une chaîne sont des caractères, de type **`char`**. Les caractères littéraux sont délimités par des guillemets simples « **`'`** ». Par exemple, pour tester que le premier caractère d'une est + on peut écrire

```
if String.length str > 0 && str.[0] = '+'
then ...
else ...
```

2.3 Lecture de fichiers

Pour lire un fichier, il faut en premier lieu obtenir un descripteur de fichier en lecture (type `in_channel`), ce qui peut être fait au moyen de la fonction `open_in : string -> in_channel` qui prend en argument une chaîne de caractères représentant un chemin et renvoie le descripteur associé. En cas d'erreur (fichier inexistant, permission incorrecte, ...) la fonction lève l'exception `Sys_error msg` où `msg` est une chaîne de caractères décrivant l'erreur.

Il est ensuite possible de lire les lignes d'un fichier en appelant la fonction

```
input_line : in_channel -> string
```

Cette dernière renvoie la prochaine ligne d'un fichier (et avance le descripteur sur la ligne suivante) ou lève l'exception `End_of_file` si la fin du fichier est atteinte. Enfin, la fonction

```
close_in : in_channel-> unit
```

referme le fichier.

Il n'est pas nécessaire pour ce projet de pouvoir ouvrir des fichiers en écriture. Pour sauvegarder les labyrinthes générés, il suffit d'utiliser `Printf.printf` dans son code OCaml pour afficher dans la console, puis une redirection du shell :

```
$ ./maze.exe random 10 10 > test.laby
```

2.3.1 Modules et compilation séparées

On a déjà mentionnés les modules `List`, `String` et `Array`. Leur documentation (comme le reste de la bibliothèque standard) est disponible en ligne :

https://www.ocaml.org/releases/4.14/htmlman/stdlib.html

Vous êtes invités à lire la documentation de ces trois modules, certaines fonctionalités existent déjà, ce qui vous fera gagner du temps.

Il est aussi possible de créer soit même ses modules. Pour cela, il suffit de mettre son code dans un fichier séparé avec l'extension `.ml`. On peut ajouter à ce fichier un fichier d'interface, portant l'extension `.mli`. Ce dernier permet masquer certains détails d'implémentation. Dans le cadre de ce projet, on suggère vivement d'utiliser un fichier séparer pour définir le type d'une grille ainsi que les opérations associées. Cela donnera par exemple un fichier `grid.ml`

```
1   type t = int list list (* c'est juste pour l'exemple, ce type
2                             n'est pas forcément le bon *)
3
4   let load file =
5       let c_in = open_in file in
6       ....
7
8   let print_aux x = ...
9
10  let print grid =
11    Printf.printf "...."
12
13  let get grid row col = ...
```

Une fois ce fichier écrit, on peut écrire un fichier d'interface `grid.mli` contenant uniquement des définitions de types et les signatures des fonctions.

```
type t    (* le type t est laissé abstrait,
             on ne peut pas y accéder comme une liste *)

val load : string -> t (* charge une grille depuis un fichier *)
val print : t -> unit (* afficher une grille *)
val get : t -> int -> int -> int (* renvoie  la case i j *)

(* la fonction print_aux n'est pas listée donc
   inaccessible depuis l'extérieur *)
```

Ainsi, depuis un autre fichier (par exemple `maze.ml`, le fichier principal) on peut utiliser le module `Grid` :

```
let g = Grid.load "fichier.laby"

let () = Grid.print g

(* ci-dessous du code qui ne compile pas: *)
(*
  let x = Grix.print_aux 42
  la fonction print_aux n'est pas exportée

  let l = match g with
          [[]] -> true
          | _ -> false

  la définition du type Grid.t n'est pas exportée,
  on peut pas l'utiliser comme une liste.
*)
```

Ce découpage en fichiers est très important pour pouvoir travailler à deux en se répartissant les tâches. Les types abstraits permettent aussi une certaine sécurité : on peut implémenter, par exemple, les grilles avec un type naïf comme ci-dessus. Le reste du code n'utilisant que les fonctions exportées, on peut ensuite changer la représentation du type sans toucher au reste du code.

3 Fichiers fournis

L'archive proposée sur le site du cours est composée comme suit :

— Un fichier `maze.ml` qui contiendra le programme principal.

— Un répertoire `test` contenant des fichiers de labyrinthes valides de taille diverses (la taille est dans le nom de fichier).

— Des fichiers `dune`, `dune-project` et `dune-workspace` permettant de construire votre projet avec la commande `dune`. Il ne faut pas modifier ces fichiers.

Vous pouvez ajouter autant de fichier OCaml que nécessaire, l'utilitaire `dune` les compilera automatiquement (s'ils sont utilisés par votre programme principal). Vous pouvez construire le projet avec `dune build`, qui produit un exécutable `maze.exe` dans le répertoire courant.

4 Modalités

Le projet est à faire en **binôme**. Les rendus se feront obligatoirement via `git`. Il est demandé d'utiliser exclusivement le `gitlab` de l'université Paris-Saclay :

`https://gitlab.dsi.universite-paris-saclay.fr/`

Il vous est demandé d'ajouter à vos projets le responsable de l'UE :

- `kim.nguyen@universite-paris-saclay.fr`

Le barème précis sera donné prochainement, mais les points pris en compte seront :

- la qualité du code
- la qualité des commits et de l'organisation
- le rapport
- les tests

Le rendu final est attendu **17 janvier 2024, 20h00**. À cette date, nous ferons un clone de votre dépôt `gitlab` et corrigerons ce qui s'y trouve.