Boggle

Projet MPIL 2018
Deuxième partie

Thibault Suzanne
thi.suzanne@gmail.com

« Oui, vous tous, comprenez que les mots sont des choses.
Ils roulent pêle-mêle au gouffre obscur des proses,
Ou font gronder le vers, orageuse forêt.
Du sphinx Esprit Humain le mot sait le secret. »
Victor Hugo, *Les Contemplations (1856)*

1 Introduction

Dans la première partie de ce projet, vous avez implémenté un moteur pour le jeu de Boggle, ainsi qu'un solveur de grille. Cette deuxième partie vous propose d'implémenter une interface graphique pour pouvoir y jouer par vous-mêmes.

Cette partie est optionnelle. Si vous souhaitez la rendre, vous pouvez le faire jusqu'au lundi 20 mai, 23h59. Un bonus sur la notation du projet sera accordé aux étudiants ayant fait ce travail. Les autres ne seront pas pénalisés.

2 Énoncé

Votre travail consiste à réaliser à l'aide de `js_of_ocaml` une interface pour le jeu de Boggle, similaire à celle disponible à l'adresse `https://www.di.ens.fr/~suzanne/misc/boggle/boggle.html`. Pour cela, vous complèterez le code de l'archive fournie à l'adresse `https://www.di.ens.fr/~suzanne/misc/boggle.tar.gz`. Il s'agit d'un corrigé de la première partie, auquel on a rajouté quelques fonctions supplémentaires qui n'étaient pas demandées. Vous pouvez également utiliser votre propre travail en rajoutant ces fonctions si vous le souhaitez.

Une grosse partie de l'interface vous est déjà fournie. Vous devez simplement implémenter le module `Action`, avec la signature demandée.

2.1 Architecture du projet

Le code de l'interface est situé dans le répertoire `web/`. Il est composé de trois modules différents : `State`, `Html` et `Action`. Il contient également un fichier `main.ml` qui en est le point d'entrée, ainsi qu'un fichier `boggle.html` contenant la structure de la page de jeu et un fichier `boggle.css` fournissant un style (rudimentaire) pour les éléments de cette page. Enfin, un module `Prelude` fournit quelques fonctions utilisables dans les différents modules.

2.1.1 `State`

Ce module vous est déjà fourni.

Le module `State` contient l'état global du jeu. À chaque action de l'utilisateur correspondra une mise à jour de cet état global.

Il est réparti en trois sous-modules différents : `Board`, `Path` et `Words`.

Le module `Board` gère la grille elle-même, c'est-à-dire le tableau des lettres. Il contient une fonction permettant d'accéder à ce tableau ainsi qu'une fonction permettant de le réinitialiser (en créant un nouveau tableau aléatoire).

Le module `Words` gère la liste des mots formés. Il contient une fonction `found` renvoyant l'ensemble des mots déjà trouvés par l'utilisateur, une fonction `check` permettant de vérifier qu'un mot existe (dans un dictionnaire chargé à la création du module) et, le cas échéant, de l'ajouter aux mots trouvés, et d'une fonction `reset` remettant à zéro (c'est-à-dire au lexique vide) l'ensemble des mots trouvés. Il fournit également une fonction `solution` renvoyant un itérateur sur l'ensemble des mots valides qu'on peut former sur la grille.

Le module `Path` gère le chemin, c'est-à-dire la suite de cases de la grille parcourues par l'utilisateur pour former un mot. Il fournit des fonctions pour ajouter des cases à ce chemin, pour le remettre à zéro, pour y accéder et pour accéder à la dernière case (si elle existe).

2.1.2 `Html`

Ce module vous est déjà fourni.

Le module `Html`, également réparti en plusieurs sous-modules, fournit différentes fonctions permettant de mettre à jour l'affichage du jeu. Ces fonctions devront donc être appelées quand l'état global du jeu change, afin de refléter ce changement sur la page.

Le sous-module `Html.Board` fournit également une fonction `cell_tag` qui renvoie l'élément HTML correspondant à une case donnée de la grille. Cette fonction est appelée dans le fichier `main.ml`, vous n'aurez pas besoin de vous en servir vous-mêmes.

Le module `Html` dépend du module `Board` : les fonctions d'affichage accèdent à l'état global afin de récupérer les données à afficher.

Remarque 2.1. *Bien que le moteur implémenté dans la Partie 1 travaille avec des minuscules, l'affichage de la grille et des mots formés est en majuscules. C'est le fichier style de la page qui permet cette configuration : vous ne devez pas vous en préoccuper (et ne devriez pas avoir à le faire si vous utiliser les fonctions fournies).*

2.1.3 `Action`

Votre travail consiste à implémenter ce module.

Le module `Action` regroupe dans le type `t` les comportements du programme en réaction à des actions de l'utilisateur.

Dans l'interface proposée, il existe quatre actions possibles, que vous devrez implémenter :

- Réinitialiser l'ensemble du jeu, quand on clique sur le bouton correspondant. Tout l'affichage (chemin et liste de mots compris) doit être mis à jour.
- Afficher les solutions, quand on clique sur le bouton correspondant. Cette action doit également désactiver le bouton permettant de valider un mot.

- Valider le mot formé par le chemin en cours, quand on clique sur le bouton correspondant. Cette action doit également remettre à zéro le chemin, et mettre à jour l'affichage de ce chemin et de la liste des mots trouvés. Si le mot n'existe pas, un message sera affiché en haut à gauche de la page. Vous utiliserez la fonction `State.Words.check` pour implémenter cette action : elle affiche elle-même le message, vous n'avez pas à vous en préoccuper.
- Ajouter une case au chemin correspondant, lorsqu'on clique sur la case en question. Cette fonction doit également mettre à jour l'affichage du chemin. Si on clique sur une case qu'il n'est pas possible d'ajouter (ce n'est pas un voisin de la dernière case, ou elle fait déjà partie du chemin), cette action ne fait rien.

Le type `Action.t` est implémenté (dans le fichier `action.ml`) par des fonctions de type `unit -> unit`. Vous devez donc implémenter ces différentes fonctions (qui se contentent dans l'archive fournie d'afficher un message `"Not implemented"`), en utilisant pour cela les fonctions du module `State` pour mettre à jour l'état global et celles du module `Html` pour mettre à jour l'affichage correspondant.

Vous devrez également implémenter les opérateurs `( => )` et `( ==> )`, qui permettent d'attacher des actions à des éléments de la page Html. **Rappel** : utilisez l'attribut `##.onclick` des éléments et la fonction `Dom_html.handler`.

`( ==> )` permet d'attacher une action à une valeur représentant un élément Html. `( => )` permet d'attacher une action en donnant simplement l'identifiant de l'élément : elle se charge alors de récupérer cet élément à l'aide de `Dom_html.getElementById` avant de lui attacher l'action.

2.2 Compilation et exécution

Pour compiler votre projet, exécutez depuis la racine la commande `make web`. Vous pourrez alors jouer au Boggle en visitant la page `_build/default/web/boggle.html`.

2.2.1 Mise en place de l'environnement

À la PPTI, la mise en place de l'environnement permettant de compiler le projet est exactement la même que pour la première partie.

Sur vos machines personnelles, en plus des commandes décrites dans le sujet de la première partie pour installer `opam`, vous aurez besoin d'installer `js_of_ocaml` :

```
$ opam install js_of_ocaml-ppx
```

3 Rendu

Les consignes de rendu sont similaires à celle de la première partie.

Pour rendre votre projet, vous l'archiverez dans un `.zip`. Cette archive devra s'appeler `boggle_WEB_<votrenom>.zip`. Vous enverrez cette archive `.zip` à l'équipe enseignante avant le **lundi 20 mai, 23h59**. Vous pouvez faire ce travail en binôme.

Annexe A Fichiers utiles

On rappelle ici l'interface des différents modules du projet, le point d'entrée, ainsi que la structure HTML de la page de jeu.

A.1 `state.mli`

```
open Boggle

(** A module for managing the global state of the game *)

module Board : sig
  val get : unit -> Board.t
  (** Get the current board (initialised to a random one). *)

  val reset : unit -> unit
  (** Create a new random board*)
end

module Words : sig
  val found : unit -> Lexicon.t
  (**  The set of valid words found by the player *)

  val check : string -> unit
  (** If the word is valid, add it to the found words *)

  val reset : unit -> unit
  (** Sets the found words to an empty set *)

  val solutions : unit -> string Iter.t
  (** The set of all valid words on the grid *)
end

module Path : sig
  val get : unit -> Path.t
  (** Get the current path *)

  val add_tile : int * int -> unit
  (** Try to add a given tile to the current path. If this is not
     allowed, do nothing. *)

  val reset : unit -> unit
  (** Sets the current path to the empty path. *)

  val last_tile : unit -> (int * int) option
  (** Gets the coordinates of the last tile of the path, if it
     exists. *)
end
```

A.2 `html.mli`

```
(** Display and manage the html rendering the state of the game. The
    [display] functions are meant to be explicitely called when this
    state changes. *)

module Board : sig
  val cell_tag : int * int -> Dom_html.element Js.t
  (** Get the element corresponding to a cell of the board. *)

  val display : unit -> unit
  (** Display the current board on the pagex. *)
end

module Words : sig
  val display_solutions : unit -> unit
  (** Display the solutions of the current board. The solutions that
      were not found by the player are typesetted differently. *)

  val display_found : unit -> unit
  (** Display the words found by the player. *)
end

module Path : sig
  val display : unit -> unit
  (** Display the word corresponding to the current path and display
      the current path. The path is displayed by colouring its tiles,
      and colouring differently the last tile. *)
end
```

A.3 `action.mli`

```
type t

val reset_board : t
val display_solutions : t
val check_word : t
val path_add_tile : int * int -> t

val ( => ) : string -> t -> unit
(** Attach an action to an element given by its id *)

val ( ==> ) : Dom_html.element Js.t -> t -> unit
(** Attach an action to an element *)
```

A.4 `main.ml=`

```
open Boggle
open Prelude
```

```
let () =
  Html.Board.display ();
  Action.(
    "new_grid" => reset_board;
    "solve_grid" => display_solutions;
    "check_word" => check_word;
  );

  for i = 0 to dim - 1 do
    for j = 0 to dim - 1 do
      let cell_tag = Html.Board.cell_tag (i, j) in
      Action.(cell_tag ==> path_add_tile (i, j))
    done
  done
```

A.5 boggle.html

```
<!doctype html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Boggle, le jeu</title>
    <link rel="stylesheet" href="boggle.css">
  </head>
  <body>
    <p id="message">
    </p>

    <table id="board">
      <tr>
        <td> ? </td> <td> ? </td> <td> ? </td> <td> ? </td>
      </tr>
      <tr>
        <td> ? </td> <td> ? </td> <td> ? </td> <td> ? </td>
      </tr>
      <tr>
        <td> ? </td> <td> ? </td> <td> ? </td> <td> ? </td>
      </tr>
      <tr>
        <td> ? </td> <td> ? </td> <td> ? </td> <td> ? </td>
      </tr>
    </table>

    <br />

    <p id="current_word">
      -
    </p>

    <div id="interface">
```

```
      <button type="button" id="check_word">Valider le mot</button>
      <button type="button" id="new_grid">Nouvelle grille</button>
      <button type="button" id="solve_grid">Résoudre la grille</button>
    </div>

    <p id="all_words"> </p>

    <script src="main.bc.js"></script>
  </body>
</html>
```