

Boggle

Projet MPIL 2018
Première partie

Thibault Suzanne
thi.suzanne@gmail.com

« Car le mot, qu'on le sache, est un être vivant.
La main du songeur vibre et tremble en l'écrivant;
La plume, qui d'une aile allongea l'envergure,
Frémit sur le papier quand sort cette figure »
Victor Hugo, *Les Contemplations* (1856)

1 Introduction

*Boggle*¹ est un jeu de lettres où les joueurs doivent, en temps limité, former des mots à l'aide de lettres disposées aléatoirement sur un plateau carré. Deux lettres consécutives dans un mot doivent être adjacentes sur le plateau (elles doivent « se toucher », éventuellement par la diagonale), et une case-lettre du plateau ne peut être utilisée qu'une seule fois par mot. Les pluriels, féminins et verbes conjugués sont autorisés. La Figure 1 présente une grille de Boggle où l'on peut notamment former les mots *engoncer*² (cf. Figure 1b), *songe* et *secs*.

Ce projet en deux parties consiste à implémenter le jeu de Boggle, pour un joueur. Pour la première partie, détaillée dans cet énoncé, vous implémenterez les bases du moteur de jeu, ainsi qu'un générateur aléatoire de grilles et un solveur trouvant tous les mots du plateau. La deuxième partie ajoutera une interface graphique permettant de jouer en solitaire.

La Section 2 détaille l'énoncé et précise le travail à réaliser. La Section 3 donne les consignes à suivre pour réaliser et rendre ce projet.

En Annexe, la Section A détaille l'utilisation de *jbuilder*, un système de compilation pour OCaml utilisé par défaut dans le projet. La Section B indique les interfaces des différents modules à implémenter (cf. Section 2). Vous n'êtes pas obligés de lire ces annexes pour réaliser le projet, mais elles peuvent le cas échéant vous apporter des informations utiles.

Vous avez jusqu'au 4 avril, 23h59 pour rendre votre travail. Les projets rendus en retard seront pénalisés.

1. <http://www.boggle.fr/>

2. **ENGONCER** v. tr. (se conjugue comme *Avancer*). XVII^e siècle. Dérivé de *gons*, ancien pluriel de *gond*, par comparaison plaisante. En parlant d'un vêtement, habiller d'une façon disgracieuse, qui fait paraître le cou enfoncé dans les épaules. *Cet habit vous engonce*. Par ext. Fig. *Avoir l'air engoncé*, guindé, contraint. Source : *Dictionnaire de l'Académie Française*, 9^e édition.



(a) La grille de jeu



(b) Un chemin formant le mot *engoncer*

FIGURE 1 – Un plateau de Boggle

2 Énoncé détaillé

2.1 Travail demandé

La suite de cet énoncé présente différents *modules*, regroupés dans une *bibliothèque*, et un *programme principal*. Votre tâche consiste à écrire ces différents modules (cf. Section 2.4), dans les fichiers spécifiés, puis à écrire le programme principal qui respecte les spécifications demandées (cf. Section 2.5).

Vous allez télécharger une archive qui contient le squelette d'une implémentation OCaml du projet. Vous pouvez si vous y tenez implémenter ce projet dans un autre langage du cours MPIL (Swift ou F#), mais il est alors **impératif** que vous suiviez rigoureusement l'architecture du projet telle qu'elle est définie dans cet énoncé (les différents répertoires comme l'organisation et l'interface des modules).

2.2 Rappel des règles du jeu

Un tableau de Boggle de dimension N est une grille carrée de N lignes et N colonnes contenant N^2 cases, dans chacune desquelles se trouve une lettre. Dans le cadre de ce projet, les lettres seront en minuscules et non-accentuées.

Le but du jeu est de former des mots à l'aide de suites de lettres adjacentes (par un côté ou par un coin) de la grille. Une lettre de la grille ne peut être présente qu'au plus une fois dans un mot formé.

Toutes les formes des mots sont autorisées : verbe conjugué, pluriel, féminin... Les lettres étant non-accentuées, on retire aussi les accents sur les mots (par exemple, le mot *hypothese* sera considéré comme un mot français valide). On fixe une longueur minimale aux mots à former : dans le Boggle classique, elle est de 3. D'autres versions proposent 2. Ces questions seront résolues en spécifiant explicitement dans un fichier tous les mots autorisés.

Le jeu classique propose à plusieurs utilisateurs de chercher simultanément des mots en un temps limité. On compte ensuite les points de chaque utilisateur à partir des mots qu'il a été le seul à trouver et de leur longueur. D'autres versions proposent des variantes, comme des lettres accordant des points en fonction de leur rareté (comme au Scrabble, il devient alors intéressant de former des mots avec le *w* s'il est présent sur la grille – mais c'est aussi plus difficile!), des cases *lettre compte double* ou *mot compte triple*... Ce projet se limitera à un seul joueur, et on ne fera pas de décompte des points dans la première partie.

2.3 Architecture du projet

Cette section présente l'organisation technique du projet. Vous devez **impérativement** respecter cette organisation dans votre rendu.

2.3.1 Les différents dossiers

Les différents dossiers présents à la racine du projet sont :

bin/ : contiendra le code de votre programme (exécutable) principal.

dict/ : contient des fichiers *dictionnaire* avec les listes de mots valides.

lib/ : contient les différents modules du moteur de jeu, utilisables sous la forme d'une bibliothèque Boggle.

sujet/ : contient l'énoncé du projet.

Certains fichiers sont déjà présents dans ces dossiers, comme `boggle.opam` ou `jbuild`. **Vous ne devez pas les supprimer** : ils sont nécessaires pour la compilation du projet avec les outils choisis. En cas de suppression accidentelle, vous pouvez simplement les récupérer dans l'archive initiale et les recopier.

2.4 Les différents modules

Le répertoire `lib/` contient les différents modules qu'on vous demande de programmer. Dans le cadre de ce projet, on ne vous demande d'écrire que les implémentations des modules. En outre, les différents types vous sont fournis : vous n'avez que les différentes fonctions à programmer.

Les modules sont présentés ici dans l'ordre qu'on vous conseille d'adopter pour les programmer, mais vous êtes libres de le faire dans un autre ordre.

Chaque module est fourni avec un module d'implémentation factice qui appelle `failwith "Unimplemented"` pour chaque fonction demandée. On a ainsi un projet qui compile même lorsque les fonctions demandées ne sont pas encore écrites (ce qui pourra vous être utile pour tester votre code). Votre travail consiste à remplacer les lignes `failwith "Unimplemented"` par une implémentation correcte de chaque fonction.

Les fonctions à compléter impérativement sont celles qui apparaissent dans les fichiers `nomDuModule.mli` correspondant aux modules considérés. Elles sont accompagnées d'une courte documentation expliquant le fonctionnement attendu. Certains fichiers contiennent d'autres fonctions, suggérées pour guider votre travail : vous pouvez choisir de ne pas les implémenter si vous avez en tête une solution différente pour les fonctions obligatoires.

2.4.1 RandomLetter

Ce module est déjà implémenté : vous pouvez l'utiliser directement.

`RandomLetter` est un module permettant de choisir une lettre aléatoirement, en fonction d'une certaine *distribution*, c'est-à-dire de la probabilité d'apparition de chaque lettre dans un texte choisi au hasard.

Il fonctionne à l'aide d'une *fonction d'ordre supérieur* qui prend en paramètre une distribution et renvoie une fonction, de type `unit -> char`, qui renvoie elle-même un caractère aléatoire à chaque appel. Lorsque vous souhaitez tirer des lettres aléatoirement, le style conseillé est donc de commencer par déclarer une fonction `picker` qui fera le tirage, puis d'appeler cette fonction. La documentation de `RandomLetter` présente un exemple.

Le sous-module `Distribution` du module `RandomLetter` contient deux distributions que vous pouvez utiliser : la distribution uniforme `uniform`, qui donnera à chaque lettre une probabilité identique d'apparition, et la distribution `fr` qui donnera à chaque lettre une probabilité d'apparition égale à sa fréquence dans la langue française.

Pour utiliser ces distributions en dehors du module `RandomLetter`, il suffit d'écrire par exemple `RandomLetter.Distribution.fr`.

2.4.2 Iter

Ce module est déjà implémenté : vous pouvez l'utiliser directement.

`Iter` est un module implémentant des *itérateurs* sur des valeurs d'un certain type. Ces itérateurs correspondent à des *séquences* d'éléments de ce type, c'est-à-dire d'un certain nombre, éventuellement infini, d'éléments qui se suivent. Les itérateurs seront utilisés dans plusieurs modules du projet.

L'interface du module précise toutes les fonctions qu'il exporte. Toutes les fonctions qui ne forcent pas l'itérateur (voir la documentation dans `Iter.mli`) s'exécutent en temps constant. Les opérations qui forcent l'itérateur ont une complexité linéaire en la taille de la séquence sous-jacente (elles ne terminent donc pas si cette séquence est infinie).

Les itérateurs sont semblables aux *streams* que vous avez déjà vus, mais ils sont purement fonctionnels : parcourir un itérateur en le forçant ne le consomme pas, il reste utilisable. Ainsi on peut obtenir le comportement suivant :

```
# let x = Iter.range 0 9;;
val x : int Iter.t = <fun>
# Iter.iter print_int x;;
0123456789- : unit = ()
# Iter.iter print_int x;;
0123456789- : unit = ()
```

2.4.3 Board

L'interface de ce module vous est fournie : à vous de compléter son implémentation.

Le module `Board` vous servira d'échauffement avant de vous attaquer au reste du projet : il s'agit d'implémenter diverses fonctions simples pour gérer les tableaux de Boggle.

Un tableau de Boggle sera simplement représenté par un tableau de tableaux de caractères : `char array array`.

2.4.4 Lexicon

L'interface de ce module vous est fournie : à vous de compléter son implémentation.

Le module `Lexicon` implémente des *lexiques*, c'est-à-dire des ensembles de mots. Ces ensembles sont implémentés sous la forme d'*arbres préfixes*, ou *tries*.

Dans un arbre préfixe, les nœuds ne stockent pas la chaîne à laquelle ils correspondent : c'est leur position dans l'arbre qui la définit. En effet, les *arêtes* de l'arbre sont étiquetées par des caractères : pour connaître le mot associé à un nœud, il faut descendre vers ce nœud depuis la racine, et la suite de lettres rencontrées sur les arêtes forme le mot recherché. Les nœuds de l'arbre sont, eux, étiquetés par un booléen qui indique s'ils correspondent à un mot appartenant au langage représenté.

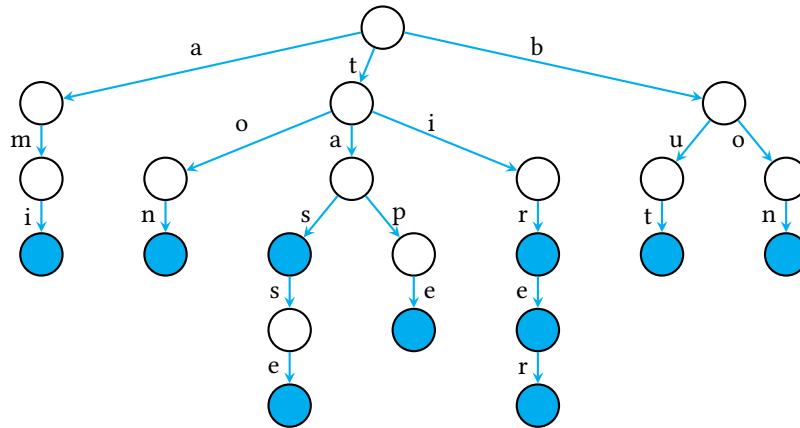


FIGURE 2 – Un exemple d'arbre préfixe

La Figure 2 montre un exemple d'arbre préfixe. Les nœuds pleins indiquent qu'il s'agit d'une fin de mot du langage. Ainsi, l'ensemble de mots représenté contient par exemple les mots *ami*, *tas*, *tasse* et *tire*, mais pas le mot *tass*, qui n'arrive pas sur un nœud « fin de mot ».

Lorsqu'on veut représenter des ensembles *de mots*, cette structure a plusieurs avantages par rapport aux représentations génériques des ensembles (comme celle du module `set` d'OCaml que vous avez peut-être utilisé) :

- Son empreinte mémoire est réduite, une même lettre étant utilisée pour plusieurs mots de l'ensemble.
- Elle fournit la plupart des opérations standards avec une meilleure complexité : par exemple, l'ajout d'un mot de longueur l dans un lexique de taille N se fait en $O(l)$ plutôt qu'en $O(N)$, ce qui est dans les cas d'utilisations typiques bien meilleur.
- Elle permet d'implémenter très rapidement l'opération, pour une lettre α , qui consiste à *renvoyer l'ensemble des mots commençant par α , auxquels on a retiré ce α* (on l'appellera *suffixes de α*). En effet, il suffit de renvoyer le sous-arbre qu'on atteint par l'arête étiquetée par α . Par exemple, sur le lexique de la Figure 2, les *suffixes de b* sont les mots *ut* et *on*. Cette opération sera particulièrement utile pour résoudre les grilles de Boggle.

Le module `Lexicon` utilise un sous-module `M` qui implémente des tables d'association (clef \mapsto valeur) dans le type `'a M.t` : une clef est un caractère (de type `char`), la valeur associée est de type `'a`. On utilise à cette fin le module `Map` de la bibliothèque standard, dont la documentation est disponible à l'adresse <https://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.Make.html>.

Le module `M` fourni implémente également une fonction `to_iter` qui renvoie un itérateur sur toutes les paires (clef, valeur) présentes dans la table. Vous pouvez vous en servir pour écrire les autres fonctions de `Lexicon`.

Comme pour `RandomLetter`, toutes les fonctions du module `M` (celles provenant de `Map` ainsi que `to_iter`) sont accessibles dans le module `Lexicon` en écrivant `M.nom_de_la_fonction`.

Le type des lexiques est alors défini comme indiqué en Figure 3. Pour comprendre cette définition, on peut la voir comme la représentation d'un lexique par son nœud racine. Le champ `eof` indique le cas échéant que ce nœud correspond à une fin de mot (le lexique ayant cette racine contient donc le mot vide `""`), et le

```

type t = {
  eow : bool;
  words : t M.t;
}

```

FIGURE 3 – Le type OCaml des lexiques

champ `words` représente les arêtes reliant le nœud à ses fils : à chaque lettre, on associe (s’il existe) le nœud atteint en suivant l’arête étiquetée par cette lettre. Le cas de base de ce type récursif est réalisé par un nœud dont le champ `word` est une table d’association vide.

La fonction `load_file` vous est déjà fournie. Le fichier `dict/dico_fr.txt` contient une liste de tous les mots valides en français (pluriels et verbes conjugués compris), un mot par ligne. Le répertoire `dict` contient également un fichier `dico_fr_petit.txt` qui liste les 100 premiers mots français. Vous pouvez si vous le souhaitez vous en servir pour faire des tests. **Vous ne devez pas modifier `dico_fr.txt`** : il servira de référence pour le programme final.

2.4.5 Path

L’interface de ce module vous est fournie : à vous de compléter son implémentation.

Le module `Path` vous servira à manipuler des *chemins* sur une grille de Boggle. Un chemin est une séquence de cases de la grille qui respecte les deux invariants suivants :

- Deux cases consécutives dans le chemin sont voisines sur la grille.
- Toute case de la grille est présente au plus une fois dans le chemin.

Remarque 2.1. *Comme les règles le précisent, deux cases voisines sont deux cases « qui se touchent » par un côté ou par un angle. On remarquera que toutes les cases n’ont pas 8 voisins : par exemple, les cases tout à droite n’ont pas de voisin droit.*

Ces chemins correspondent donc aux « trajets » que l’on peut suivre sur la grille pour former des mots. Les fonctions du module `Path` doivent garantir qu’on ne peut former que des chemins valides (qui respectent les deux invariants).

Un chemin est simplement représenté par une liste de coordonnées (numéro de ligne, numéro de colonne) de cases de la grille. N’oubliez pas que le module `Lexicon` vous permet de construire des ensembles de mots (où chaque mot n’est par définition présent qu’une seule fois).

2.4.6 Solver

L’interface de ce module vous est fournie : à vous de compléter son implémentation.

Le module `Solver` fournit une méthode de résolution de grilles de Boggle, qui trouve tous les mots formables pour une grille donnée.

On utilisera à cette fin la technique du *retour sur trace*, aussi appelée *backtracking*³. Le retour sur trace est une technique de programmation, ainsi qu’une famille d’algorithmes qui la mettent en œuvre, consistant à prendre des décisions pour faire avancer la résolution d’un problème et à revenir en arrière en cas de blocage (c’est-à-dire lorsqu’il s’avère que ces décisions sont incohérentes).

3. Bien qu’anglais, ce mot est quasi-systématiquement utilisé, y compris en français.

Si vous avez déjà résolu des Sudoku de difficulté élevée, vous avez sans doute déjà utilisé cette technique sans le savoir : on essaye de placer un chiffre dans une case, on continue la résolution de la grille avec ce chiffre, et s'il s'avère qu'elle est impossible, on efface le chiffre et on en essaye un autre.

L'application du backtracking au Boggle consiste schématiquement à ajouter des cases à un chemin jusqu'à ce qu'on ne puisse plus former aucun mot avec le chemin en cours (en calculant successivement les suffixes de chaque lettre qu'on ajoute). À chaque fois qu'on arrive sur une fin de mot possible, on ajoute le mot aux solutions.

L'Algorithme 1 formalise cette explication. La fonction `BACKTRACK` est le cœur de la résolution : elle visite une case en l'ajoutant à un chemin déjà parcouru et avance dans les mots du lexique en sélectionnant les suffixes de la lettre inscrite sur la case visitée. Si on arrive sur une fin de mot, le chemin parcouru (auquel on a ajouté la case en cours) est un chemin valide. `BACKTRACK` s'appelle récursivement pour calculer les chemins valides en passant par les voisins de la case visitée, et renvoie le total des chemins obtenus. `FINDALLPATHS` appelle simplement `BACKTRACK` sur chaque case de la grille avec un lexique initial et un chemin initialement vide.

La présentation de l'Algorithme 1 utilise des notations ensemblistes pour les itérateurs. Voici un bref rappel de leurs significations : \emptyset est l'itérateur vide, $\{x\}$ est l'itérateur sur un seul élément x , et si $iter$ est un itérateur sur des valeurs x_0, x_1, x_2, \dots de type τ et f une fonction de type $\tau \rightarrow \nu$, alors $\bigcup_{x \in iter} f(x)$ est un itérateur sur des valeurs de type ν qui itère successivement sur tous les éléments des $f(x_i)$. À quelle fonction du module `Iter` cette notation vous fait-elle penser ?

Votre travail sur le module `Solver` consiste à implémenter ces deux fonctions. On vous donne l'algorithme détaillé, il ne vous reste donc plus qu'à l'adapter en OCaml à l'aide des modules que vous avez déjà implémentés.


2.5 Le programme final

C'est à vous d'implémenter ce programme.

Une fois tous les différents modules de la bibliothèque implémentés et testés, il ne vous reste plus qu'à écrire le programme final. Le fichier correspondant est `bin/main.ml`. Il est compilé avec la bibliothèque par la commande `$ make` (depuis le répertoire racine du projet) et peut être exécuté avec la commande `$./run.sh`. Vérifiez que ces commandes fonctionnent : si vous n'avez pas modifié le fichier `bin/main.ml`, le programme doit simplement afficher un message vous invitant au travail.

On prendra comme mots valides la liste des mots du fichier `dict/dico_fr.txt` de 3 lettres ou plus. Attention : vous n'avez pas le droit de modifier ce fichier ! Vous n'avez pas non plus le droit d'en créer un autre qui contienne uniquement les mots de plus de 3 lettres. Utilisez les fonctions que vous avez écrites pour construire cette liste.

Exécuté avec la commande `$./run.sh`, le programme doit :

1. Afficher une grille de Boggle aléatoire, avec la fréquence des lettres adaptée au français.
2. Afficher un message invitant l'utilisateur à appuyer sur , et attendre cet évènement.⁴
3. Afficher toutes les solutions de la grille (appel : d'au moins 3 lettres), classées par ordre décroissant de longueur puis par ordre alphabétique croissant.⁵ Chaque mot ne doit être affiché qu'une seule fois.

4. Conseil : utilisez `read_line`.

5. Conseil : passez par les listes et utilisez `List.sort`.

```

1 function BACKTRACK(board, lexicon, path, (i, j))
    Input :
        board : une grille de Boggle
        lexicon : les suffixes du chemin déjà parcouru parmi les mots autorisés
        path : le chemin déjà parcouru
        (i, j) : les coordonnées d'une case à essayer d'ajouter au chemin déjà parcouru
    Output :
        Un itérateur sur tous les chemins commençant par path formant un mot valide
2 if On peut ajouter (i, j) à path then
3     path'  $\leftarrow$  path ++ (i, j)
4      $\alpha \leftarrow$  lettre de la case (i, j)
5     lexicon'  $\leftarrow$  suffixes de  $\alpha$  dans lexicon
6     if lexicon' est vide then
7         return  $\emptyset$ 
8     else
9         if lexicon' contient le mot vide then
10            solution_chemin_courant  $\leftarrow$  {path'}
11        else
12            solution_chemin_courant  $\leftarrow$   $\emptyset$ 
13        end
14        solutions_via_voisins  $\leftarrow$   $\bigcup_{(i', j') \in \text{voisins de } (i, j)}$  BACKTRACK(board, lexicon', path', (i', j'))
15        return solution_chemin_courant  $\cup$  solutions_via_voisins
16    end
17 else
18     return  $\emptyset$ 
19 end

20 function FINDALLPATHS(board, lexicon)
    Input :
        board : une grille de Boggle
        lexicon : un lexique de mots autorisés
    Output :
        Un itérateur sur tous les chemins sur la grille formant un mot autorisé
21 return  $\bigcup_{(i, j) \in \text{cases de la grille}}$  BACKTRACK(board, lexicon, empty path, (i, j))

```

Algorithme 1 : Résolution d'une grille de Boggle par backtracking

Ce programme peut également prendre un paramètre dans la ligne de commande, qui lui sera alors passé de la façon suivante : `./run.sh unparametre`. S'il est présent, ce paramètre sera considéré comme une grille à résoudre, donnée par ses lettres (en minuscule) dans l'ordre usuel de lecture (de gauche à droite, puis de haut en bas). Le programme devra alors simplement afficher toutes les solutions de la grille, avec les mêmes contraintes que précédemment : elles doivent être classées par ordre décroissant de longueur puis par ordre alphabétique croissant, et chaque mot ne doit être affiché qu'une seule fois. **Note** : on pourra supposer que tous les caractères de `unparametre` sont des lettres minuscules sans le vérifier.

Les modules de la bibliothèque que vous avez écrits sont regroupés dans un super-module `Boggle` (par exemple, vous pouvez appeler la fonction `Boggle.Lexicon.has_empty_word`.)

Certaines opérations du programme peuvent échouer, la fonction correspondante renvoyant alors `None`. Le programme devra alors terminer prématurément en affichant un message décrivant l'erreur rencontrée. Pour terminer le programme OCaml, vous pourrez écrire `exit 1`.

3 Déroutement du projet et consignes générales

3.1 Téléchargement du squelette

Le squelette du projet est disponible à l'adresse <https://www.di.ens.fr/~suzanne/misc/boggle.tar.gz>. Téléchargez l'archive et extrayez la (`$ tar xzvf boggle.tar.gz` sous Linux). Elle contient un dossier `boggle` contenant lui-même le squelette du projet (ainsi qu'une copie PDF de cet énoncé).

3.2 Mise en place de l'environnement

3.2.1 À la PPTI

Le projet est prévu pour compiler dans l'environnement de la PPTI, auquel on a ajouté et configuré *opam*, le gestionnaire de paquets moderne pour OCaml. Vous n'avez pas besoin de manipuler *opam* par vous-mêmes, mais vous avez besoin de configurer votre shell afin d'intégrer son environnement. Pour cela, ajoutez les lignes suivantes à votre fichier `~/.bashrc` (ou l'équivalent si vous utilisez un autre shell) :

```
# OPAM configuration
PATH=$PATH:/users/Enseignants/suzanne/bin; export PATH;
eval $(opam config env --root="/users/Enseignants/suzanne/.opam")
```

Si vous avez déjà des lignes similaires, remplacez les par celles-ci. En cas de doute, envoyez un mail à thi.suzanne@gmail.com. Relancez ensuite `bash` pour recharger cette configuration.

3.2.2 Sur vos machines personnelles

Si vous souhaitez travailler sur vos machines personnelles, vous aurez besoin d'un environnement OCaml comprenant au moins *jbuilder*, ainsi qu'*odoc* si vous souhaitez générer la documentation HTML (cf. Section A). Il est également conseillé d'utiliser la même version d'OCaml qu'à la PPTI pour être sûr d'écrire un programme qui compilera sur l'environnement d'évaluation.

Pour mettre en place cet environnement, vous pouvez utiliser les commandes suivantes :

```
# Pour installer opam la première fois :
$ apt install opam # Ou l'équivalent pour votre système d'exploitation/distribution
$ opam init --comp 4.06.0
```

```
# opam vous proposera alors d'éditer le fichier de configuration de
# votre shell pour charger automatiquement les variables
# d'environnement nécessaires à son fonctionnement. Acceptez.
```

```
# Pour mettre en place l'environnement :
```

```
$ opam switch 4.06.0
$ eval $(opam config env)
$ opam install jbuilder odoc
```

Vous pourrez ensuite compiler et exécuter le squelette du projet. En cas de question ou de problème, envoyez un mail à thi.suzanne@gmail.com.

3.3 Environnement, compilation et exécution

Le projet est fourni avec un Makefile et un script `run.sh`. La commande `$ make` compile le projet, et la commande `$./run.sh` l'exécute. Le squelette OCaml contient une version préécrite de ces deux commandes qui fonctionne dans l'environnement OCaml de la PPTI (configuré comme indiqué en Section 3.2.1).

Si vous utilisez un autre langage, vous devez écrire un Makefile et un script `run.sh` qui respectivement compilent et exécutent le projet dans l'environnement par défaut de la PPTI (c'est-à-dire l'environnement de base accessible à un nouvel utilisateur, sans configuration supplémentaire).

L'environnement de compilation et d'exécution en OCaml proposé dans le squelette utilise `jbuilder`. La Section A en annexe développe son utilisation, ainsi que des commandes supplémentaires qui peuvent vous servir.

Quel que soit le langage choisi, vous devez **impérativement** rendre un projet qui **compile sans erreur** avec la commande `$ make` et qui s'exécute avec la commande `$./run.sh`.

3.4 Rendu

Rappel : Vous avez jusqu'au 4 avril, 23h59 pour rendre votre travail.

Vous garderez l'architecture des différents dossiers telle qu'elle vous a été distribuée. Vous renommerez le dossier principal (initialement `boggle`) en `boggle_<votrenom>` (évidemment, vous remplacerez `<votrenom>` par votre nom de famille).

Vous placerez ensuite ce dossier dans une archive `.zip`. Pour cela, sous Linux, placez-vous **dans le dossier contenant le dossier `boggle_<votrenom>`** et exécutez la commande suivante :

```
$ zip -r boggle_<votrenom> boggle_<votrenom>
```

Vérifiez que l'archive ainsi générée (qui doit s'appeler `boggle_<votrenom>.zip`) contient bien un dossier `boggle_<votrenom>` qui contient lui-même votre code. Vous pouvez par exemple exécuter la commande suivante dans l'environnement de la PPTI (dans lequel elle doit fonctionner) :

```
$ unzip boggle_<votrenom>.zip
```

Si vous faites ce projet en binôme, remplacez dans les instructions précédentes `<votrenom>` par `<nom1>_<nom2>`, où `<nom1>` et `<nom2>` sont vos deux noms de famille.

Vous enverrez ensuite l'archive `.zip` obtenue par mail à l'équipe pédagogique (voir le lien sur la page web du cours, à l'adresse <http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2017/ue/3I008-2018fev/>). Vous indiquerez comme sujet du mail [Rendu MPIL 1] `<nom1> <nom2>`.

La notation tiendra très fortement compte du respect de ces consignes!

Annexe A Utilisation de jbuilder

Le projet utilise par défaut jbuilder (qui sera bientôt renommé dune), un système de compilation moderne pour OCaml. Les fichiers nécessaires à son bon fonctionnement vous sont fournis, ainsi qu'un Makefile et un script run.sh qui encapsulent son utilisation.

Vous pouvez également utiliser les commandes suivantes (à la racine du projet) :

jbuilder build lib/.merlin : génère le fichier lib/.merlin. Vous pouvez également générer bin/.merlin, pour utiliser merlin lors de l'édition des différents fichiers.

jbuilder build @doc : génère la documentation en format web de l'interface des différents modules de la bibliothèque Boggle. Vous pouvez ensuite consulter cette documentation en ouvrant avec votre navigateur le fichier _build/default/_doc/boggle/Boggle/index.html. Il s'agit d'une version plus agréable à lire (et avec des liens cliquables entre les différents modules) de la documentation présente dans les fichiers .mli.

jbuilder build : compile la librairie Boggle et l'exécutable boggle (qu'on vous demande de programmer).

jbuilder exec boggle [args] : lance l'exécutable boggle avec les paramètres optionnels args (voir la Section 2.5).

jbuilder utop : lance utop, un terminal OCaml, en chargeant automatiquement la librairie Boggle. **Cette commande est très utile pour tester votre implémentation des différents modules.**

jbuilder clean : supprime les fichiers générés par jbuilder pour repartir à neuf. Attention, cette commande supprime également la documentation et les fichiers .merlin, il faudra les générer à nouveau.

Annexe B Interfaces

Cette fonction rappelle les interfaces des différents modules, tels qu'ils sont fournis avec le squelette de base. On rappelle que chaque fichier dont le contenu est rappelé ici est situé dans le répertoire `lib/` du projet.

B.1 `iter.mli`

```
(** Itérateurs sur des séquences de valeurs d'un certain
type. Certaines fonctions {e forcent} l'itérateur : elles
consomment tous ses éléments pour produire le résultat. Ces
fonctions ne terminent pas si l'itérateur est infini. Les autres
fonctions sont paresseuses : elle produisent un itérateur mais
n'évaluent pas ses éléments lorsqu'on les appelle. Elles terminent
même sur des itérateurs infinis. *)

type 'a t = ('a -> unit) -> unit
(** Un itérateur de valeurs de type ['a].

    Ce type n'est pas abstrait. Vous pourrez donc l'utiliser
    directement dans les autres modules, bien que ce ne soit ni
    nécessaire ni conseillé dans le cadre de ce projet, les fonctions
    exportées par ce module étant suffisantes.
*)

val empty: 'a t
(** L'itérateur sur 0 élément. *)

val singleton : 'a -> 'a t
(** Un itérateur sur une séquence d'un élément. *)

val cons : 'a -> 'a t -> 'a t
(** [cons x s] itère sur [x] puis sur les éléments de [s]. *)

val append : 'a t -> 'a t -> 'a t
(** [append s1 s2] itère sur les éléments de [s1] puis ceux de [s2] *)

val length : 'a t -> int
(** [length s] est le nombre d'éléments de [s]. Force l'itérateur. *)

val range : int -> int -> int t
(** [range from to_] est itère sur les entiers de [from] (inclus)
    à [to_] (inclus). Si [from > to_], [range from to_] est l'itérateur
    vide. *)

val flatten : 'a t t -> 'a t
(** Un itérateur sur les éléments concaténés de plusieurs
    itérateurs. *)

val map : ('a -> 'b) -> 'a t -> 'b t
```

```

(** [map f s] est un itérateur sur [f x1, f x2, ...], où [x1, x2, ...]
    sont les éléments de [s]. *)

val flat_map : ('a -> 'b t) -> 'a t -> 'b t
(** [flat_map f s] applique paresseusement [f] à tous les éléments de
    [s], puis itère successivement sur les éléments des itérateurs
    ainsi obtenus. Fonctionnellement, [flat_map f s = flatten (map
    f s)]. *)

val filter : ('a -> bool) -> 'a t -> 'a t
(** [filter predicate s] itère sur les éléments [x] de [s] pour
    lesquels [predicate x] vaut [true]. *)

val fold : ('acc -> 'a -> 'acc) -> 'acc -> 'a t -> 'acc
(** Si [s] est l'itérateur sur [x1, x2, ..., xn], [fold f init s] est
    [f (... (f (f acc x1) x2) ... ) xn]. Force l'itérateur. *)

val exists : ('a -> bool) -> 'a t -> bool
(** Renvoie [true] si et seulement si il existe un élément d'un
    itérateur satisfaisant un prédicat donné. Si aucun élément ne
    satisfait le prédicat, renvoie [false] si l'itérateur est fini, et
    ne termine pas sinon.

    {b Note :} cette fonction force « partiellement » l'itérateur,
    c'est-à-dire jusqu'à trouver un élément satisfaisant le prédicat. *)

val iter : ('a -> unit) -> 'a t -> unit
(** Applique une fonction successivement à tous les éléments d'un
    itérateur. Force l'itérateur. *)

val product : 'a t -> 'b t -> ('a * 'b) t
(** [product a b] itère sur tous les couples formés d'un élément de
    [a] et d'un élément de [b]. *)

val to_rev_list : 'a t -> 'a list
(** Renvoie la liste composée de tous les éléments d'un itérateur, en
    ordre inverse. Force l'itérateur. *)

```

B.2 randomLetter.mli

```

(** Un module pour sélectionner des lettres aléatoirement selon une
    certaine distribution. *)

module Distribution : sig
  (** Un module regroupant des distributions de lettres. *)

  type t
  (** Le type des distributions de lettres. *)

  val uniform : t

```

```

(** La distribution uniforme des 26 lettres de l'alphabet (en
    minuscule). Chaque lettre a la même fréquence. *)

val fr : t
(** La distribution des 26 lettres minuscules correspondant à la
    langue française. Les accents et autres signes diacritiques ont
    été retirés.
    *)
end

val picker : Distribution.t -> (unit -> char)
(** [picker distribution] renvoie une fonction [random_letter] qui
    permet de tirer aléatoirement une lettre selon la distribution
    donnée.

    Voici un exemple d'utilisation de {!picker} dans le terminal (une
    fois le module {!RandomLetter} ouvert) :

    {[
# let random_letter_fr = picker Distribution.fr;;
val random_letter_fr : unit -> char = <fun>
# random_letter_fr ();;
- : char = 's'
# random_letter_fr ();;
- : char = 'a'
# random_letter_fr ();;
- : char = 'e'
    ]}

    [random_letter_fr] renverra une lettre avec une probabilité égale
    à sa fréquence dans la langue française. On notera par exemple
    qu'ici, cette fonction a renvoyé 3 lettres parmi les 4 plus
    fréquentes (c'est un hasard, mais c'était un résultat plus probable
    que ['k'], ['w'] et ['z']).
    *)

```

B.3 board.mli

```

(** Module pour définir et travailler sur des grilles de Boggle. On ne
    considère que des grilles carrées. *)

type t
(** Le type d'une grille. *)

val get_letter : t -> int -> int -> char
(** [get_letter board i j] renvoie le caractère présent à la ligne [i]
    et à la colonne [j] sur la grille [board]. *)

val dim : t -> int
(** La dimension d'une grille, c'est à dire le nombre de lignes (qui

```

```

    est égal au nombre de colonnes). *)

val all_positions : t -> (int * int) Iter.t
(** Un itérateur sur toutes les positions (ligne, colonne) d'une
    grille. *)

val are_neighbours : t -> int * int -> int * int -> bool
(** Est-ce que deux cases données par leurs positions sont voisines ?
    Deux cases sont voisines si elles se "touchent" par un côté ou en
    diagonale. On considérera que les cases données sont des cases
    valides sur la grille. *)

val neighbours : t -> int * int -> (int * int) Iter.t
(** Un itérateur sur les cases voisines d'une case donnée. *)

val make : int -> (unit -> char) -> t
(** [make dim make_char] crée une grille de dimension [dim] et appelle
    [make_char ()] pour remplir chaque case. [make_char] est une
    fonction renvoyant un caractère à chaque appel. Voir le module
    {!RandomLetter}. *)

val from_string : string -> t option
(** [from_string s] crée une grille [grid] à partir d'une chaîne de
    caractères [s] comprenant tous les caractères de la grille dans
    l'ordre usuel de lecture (de gauche à droite, puis de haut en
    bas). [s] doit avoir un nombre carré de caractères, c'est-à-dire
    que la longueur de [s] doit être le carré d'un entier [n] qui sera
    donc la dimension de la grille. Si [s] n'a pas un nombre carré de
    caractères, [from_string s] renvoie [None]. *)

val print : t -> unit
(** Affiche une grille. Deux caractères consécutifs sur une même ligne
    sont séparés par une espace. Deux lignes consécutives sont
    affichées consécutivement. On affichera un caractère saut de ligne
    après la dernière ligne. On pourra, pour simplifier le code,
    afficher une espace après le caractère de chaque ligne.

    Voici un exemple d'affichage d'une grille :
    {[a t r s
    e u l c
    n m t e
    h t s c]}
    *)

```

B.4 path.mli=

```

(** Chemins sur une grille. Un chemin est une séquence de cases
    valides de la grille qui respecte les deux invariants suivants :

    - Deux cases consécutives dans le chemin sont voisines sur la grille.

```

```

- Toute case de la grille est présente au plus une fois dans le chemin.

Tout chemin construit à l'aide des fonctions fournit dans ce
module garantit de respecter ces deux invariants.
*)

type t
(** Le type des chemins. *)

val empty : t
(** Le chemin vide (qui ne contient aucune case). *)

val add_tile : Board.t -> t -> (int * int) -> t option
(** Ajoute une case de la grille, donnée par ses coordonnées (numéro
de ligne, numéro de colonne), à un chemin. Renvoie [None] si le
chemin ainsi constitué ne serait pas valide (c'est-à-dire si un des
invariants des chemins n'était pas respecté).
*)

val to_string : Board.t -> t -> string
(** Renvoie le mot décrit lorsqu'on parcourt un chemin sur une grille,
c'est-à-dire le mot constitué des lettres correspondant à chaque
case du chemin dans l'ordre.
*)

val iter_to_words : Board.t -> t Iter.t -> string Iter.t
(** Étant donné une grille et un itérateur sur des chemins, renvoie
un itérateur sur les mots décrits par ces chemins. Si deux chemins
décrivent le même mot, ce mot n'est présent qu'une fois dans
l'itérateur renvoyé. Aucun ordre n'est spécifié sur les mots de
l'itérateur.
*)

```

B.5 lexicon.mli

```

(** Un module pour gérer des Lexiques, c'est-à-dire des ensembles de
mots. *)

type t
(** Le type des lexiques. *)

val empty : t
(** Le lexique vide, qui ne contient aucun mot *)

val is_empty : t -> bool
(** Est-ce qu'un lexique est vide ? *)

val add : string -> t -> t
(** [add word lexicon] ajoute le mot [word] au lexique [lexicon]. *)

```



```

val to_iter : t -> string Iter.t
(** Un itérateur sur les mots d'un lexique. *)

val letter_suffixes : t -> char -> t
(** [letter_suffixes lexicon alpha] est le lexique contenant tous les
    mots présents dans [lexicon] et commençant par la lettre [alpha],
    desquels on a retiré cette première lettre [alpha].

    Par exemple, si [lexicon] contient les mots ["abstraction"],
    ["a,"] et ["chameau"], [letter_suffixes lexicon 'a'] contiendra les
    mots ["bstraction"] et [""] (le mot vide).
*)

val has_empty_word : t -> bool
(** Renvoie [true] si le lexique contient le mot vide, [false]
    sinon.
*)

val filter_min_length : int -> t -> t
(** [filter_min_length len lexicon] renvoie le lexique composés des
    mots de [lexicon] qui ont [len] caractères ou plus.
*)

val load_file : string -> t option
(** Charge un fichier de mots dans un dictionnaire. [load_file
    filename] ouvrira le fichier [filename] et ajoutera chaque ligne de
    ce fichier comme un mot du lexique [lexicon], puis renverra [Some
    lexicon]. Si l'ouverture du fichier échoue, renvoie [None].

    Aucune vérification n'est faite sur le contenu du fichier : chaque
    ligne est considérée comme un mot valide. Les caractères "saut de
    ligne" ne sont pas considérés comme faisant partie des mots.
*)

```

B.6 solver.mli

```

(** Module pour résoudre les grilles de Boggle. *)

val find_all_paths : Board.t -> Lexicon.t -> Path.t Iter.t
(** Étant donné une grille et un lexique, renvoie un itérateur sur
    tous les chemins possibles dans la grille formant un mot présent
    dans le lexique. *)

```