2 Gestion du plateau

1) Il faut d'abord créer un plateau vide à l'aide de la fonction `nouveau_plateau` et utiliser la fonction `affiche_plateau` pour afficher ce plateau. Il faut ensuite afficher le plateau déjà construit dans `plateau_figure`.

```
let () =
  print_endline "Plateau vide :";
  let plateau_initial = nouveau_plateau () in
  affiche_plateau plateau_initial;
  print_newline ();
  print_endline "Plateau de la figure 1c :";
  affiche_plateau plateau_figure
```

2) Pour chaque caractère c du mot, on regarde la position $(i, j + k)$:

- si c'est une case vide alors on ajoute c à la liste ;
- si c'est une lettre identique alors on continue ;
- si c'est une lette différente, il s'agit d'une erreur logique qu'on ne traite pas dans cette question.

```
let lettres_necessaires_horizontal plateau mot (i, j) =
  let rec aux k acc =
    if k = String.length mot then
      List.rev acc
    else
      let case =
        try plateau.(i).(j + k)
        with Invalid_argument _->failwith "Position hors plateau"
      in
      match case with
      | Lettre(c) ->
          if c <> mot.[k] then
            failwith "Conflit avec une lettre déjà posée"
          else
            aux (k + 1) acc
      | Vide | LettreDouble | LettreTriple
      | MotDouble | MotTriple | Centrale ->
         aux (k + 1) (mot.[k] :: acc)
  in
  aux 0 []
```

3) On peut factoriser la logique commune entre la fonction `lettres_necessaires_horizontal` et une future `lettres_necessaires_vertical` en généralisant l'indice de déplacement. Plutôt que de dupliquer du code, on utilise le paramètre `est_vertical` pour déterminer dynamiquement si on incrémente la ligne ou la colonne.

```
let lettres_necessaires plateau mot (i, j) est_vertical =
  let rec aux k acc =
    if k = String.length mot then
      List.rev acc
    else
      let i', j' = if est_vertical then (i + k, j) else (i, j + k)
     in
      let case =
        try plateau.(i').(j')
        with Invalid_argument _->failwith "Position hors plateau"
      in
      match case with
      | Lettre(c) ->
          if c <> mot.[k] then
            failwith "Conflit avec une lettre déjà posée"
          else
            aux (k + 1) acc
      | Vide | LettreDouble | LettreTriple | MotDouble | MotTriple
     | Centrale ->
          aux (k + 1) (mot.[k] :: acc)
  in
  aux 0 []
```

4) 💻 On utilise la fonction `points` fournie dans le module `Alphabet`.

```
let somme_lettres mot =
  let rec aux k acc =
    if k = String.length mot then acc
    else aux (k + 1) (acc + points mot.[k])
  in
  aux 0 0
```

5) 💻 On prend en compte uniquement la valeur du mot même.

```
let valeur_mot_seul plateau mot (i, j) est_vertical =
  let rec aux k acc mot_mul =
    if k = String.length mot then acc * mot_mul
    else
      let i', j' = if est_vertical then (i + k, j) else (i, j + k)
     in
      let case = plateau.(i').(j') in
      match case with
      | Lettre(c) ->
          let v = points c in
          aux (k + 1) (acc + v) mot_mul
      | Vide | Centrale ->
          let v = points mot.[k] in
          let mot_mul' =
          if case = Centrale then max mot_mul 2 else mot_mul in
          aux (k + 1) (acc + v) mot_mul'
      | LettreDouble ->
          let v = 2 * points mot.[k] in
          aux (k + 1) (acc + v) mot_mul
```

```
      | LettreTriple ->
          let v = 3 * points mot.[k] in
          aux (k + 1) (acc + v) mot_mul
      | MotDouble ->
          let v = points mot.[k] in
          aux (k + 1) (acc + v) (max mot_mul 2)
      | MotTriple ->
          let v = points mot.[k] in
          aux (k + 1) (acc + v) (max mot_mul 3)
  in
  aux 0 0 1
```

6) ✎ Le mot `NUCAL` rapporte 14 points.

```
Printf.printf "Plateau 1b\n";
  let plateau = nouveau_plateau () in
  List.iter (fun (coord, est_vertical, mot) ->
      place_mot plateau coord est_vertical mot)
    [
      ((7,3), false, "PARADERAS");
      ((5,9), true, "MORIO");
      ((3,11), true, "HIAIS");
      ((6,8), false, "JOCISTE")
    ];
  affiche_plateau plateau;
  print_newline ();
  let score = valeur_mot_seul plateau "NUCAL" (4, 10) true in
  Printf.printf "Score de NUCAL seul : %d\n" score
```

7) 👤 Quand un mot est ajouté sur le plateau, il peut croiser des lettres déjà posées ou en poser de nouvelles à côté d'autres lettres, formant ainsi des mots perpendiculaires.

Pour chaque lettre ajoutée cette fois-ci, on regarde si elle crée un mot perpendiculaire en s'attachant à d'autres lettres déjà présentes. Si oui, on calcule la valeur de ce mot, en tenant compte uniquement des bonus de la lettre qu'on pose, et on l'ajoute au score total.

On ne compte pas le mot principal, et on ignore les lettres déjà posées dans ce mot.

Les étapes de l'algorithme sont :

- parcourir le mot principal lettre par lettre, à chaque position (i, j) dans le mot posé,
  - si la case contient déjà une lettre (donc ce n'est pas une lettre posée cette fois-ci), on ignore cette position,
  - sinon, c'est une nouvelle lettre posée, on vérifie s'il existe un mot perpendiculaire ;
- explorer dans la direction perpendiculaire,
  - si le mot est posé horizontalement, on explore vers le haut et le bas pour trouver des lettres adjacentes verticalement,

  - si le mot est posé verticalement, on explore vers la gauche et la droite pour trouver des lettres adjacentes horizontalement,
  - on reconstitue ainsi un mot perpendiculaire centré autour de la lettre qu'on vient de poser ;
- si un mot perpendiculaire est trouvé, on calcule son score et on l'ajoute au total ;
- renvoyer la somme des scores de tous les mots perpendiculaires formés.

8) 💻

```
let valeur_mot plateau mot (i, j) est_vertical =
  let score_principal = valeur_mot_seul plateau mot (i, j)
    est_vertical in
  let score_perpendiculaires =
    let rec collect k acc =
      if k = String.length mot then acc
      else
        let (i', j') =
          if est_vertical then (i+k, j) else (i, j+k) in
        match plateau.(i').(j') with
        | Lettre(_) -> collect (k + 1) acc (* Lettre déjà posée *)
        | _ ->
          let rec remonte (i, j) =
            let (i', j') =
              if est_vertical then (i, j-1) else (i-1, j) in
            match plateau.(i').(j') with
            | Lettre(_) -> remonte (i', j')
            | _ -> (i, j)
          in
          let rec descend (i, j) acc =
            if (i, j) = (i', j') then (* On va poser une lettre *)
              let next =
                if est_vertical then (i, j+1) else (i+1, j) in
              descend next (acc ^ String.make 1 mot.[k])
            else
              match plateau.(i).(j) with
              | Lettre(c) ->
                let next =
                  if est_vertical then (i, j+1) else (i+1, j) in
                descend next (acc ^ String.make 1 c)
              | _ -> acc
          in
          let start = remonte (i', j') in
          let mot_perp = descend start "" in
          if String.length mot_perp > 1 then
            let score = valeur_mot_seul plateau mot_perp start (
    not est_vertical) in
            collect (k + 1) (acc + score)
          else
            collect (k + 1) acc
    in
    collect 0 0
  in score_principal + score_perpendiculaires
```

9) 💻 La fonction renvoie toutes les positions (i, j) du plateau qui sont des ancres, c'est-à-dire :

- la case centrale, si elle est encore libre ;
- toutes les cases vides qui sont adjacentes (haut, bas, gauche, droite) à au moins une lettre déjà posée.

```
let ancres_plateau plateau =
  let n = Array.length plateau in
  let est_lettre i j =
    match plateau.(i).(j) with
    | Lettre(_) -> true
    | _ -> false
  in
  let est_ancre i j =
    match plateau.(i).(j) with
    | Lettre(_) -> false
    | _ ->
      plateau.(i).(j) = Centrale ||
      let voisins = [ (i-1, j); (i+1, j); (i, j-1); (i, j+1) ] in
      List.exists (fun (x, y) ->
        x >= 0 && x < n && y >= 0 && y < n && est_lettre x y
      ) voisins
  in
  let rec parcours i j acc =
    if i = n then acc
    else if j = n then parcours (i + 1) 0 acc
    else
      let acc' = if est_ancre i j then (i, j) :: acc else acc in
      parcours i (j + 1) acc'
  in
  List.rev (parcours 0 0 [])
```

10) ✎ Le plateau donné en figure 1c possède 73 ancres.

```
Printf.printf "ancres_plateau\n";
let ancres = ancres_plateau plateau_figure in
Printf.printf "Ancres détectées :\n";
List.iter (fun (i,j) -> Printf.printf "(%d,%d) " i j) ancres;
print_newline ();
Printf.printf "\nPlateau avec %d ancres :\n" (List.length ancres);
```

3 Manipulation du lexique

11) ✎ L'ensemble des mots du lexique du Scrabble est un langage fini, c'est-à-dire un ensemble fini de chaînes de caractères sur un alphabet donné (ici, les lettres majuscules de A à Z). Or, tout langage fini est régulier donc l'ensemble des mots du lexique forme un langage régulier

12) 💻

```
let construit_gaddag nom_fichier =
  let g = vide () in
  let mot_suivant = lire_lexique nom_fichier in
  try
    while true do
      let mot = mot_suivant () in
      if String.length mot > 0 then
        ajoute_conjugues g mot
    done;
    g  (* jamais atteint *)
  with Lexique_fini -> g
```

4 Trouver un coup valide

13) 💻 Voir le fichier `scrabble.ml` pour le code de la fonction `trouver_coup`.
Il faut retravailler sur les coins du plateau.

La fonction `trouver_coup` cherche tous les mots valides pouvant être placés à partir d'une position d'ancre donnée, en utilisant un tirage et le GADDAG. Elle fonctionne en trois étapes principales. (1) Point de départ : pour chaque lettre du tirage, on tente de la placer sur l'ancre, uniquement si cela forme un mot valide dans la direction perpendiculaire. Cette validation est effectuée par la fonction `verifie_lettre`, qui reconstruit les lettres déjà présentes autour de la case (dans l'axe perpendiculaire au mot en cours de construction) et vérifie dans le GADDAG si le mot ainsi formé est valide. (2) `recule_depuis` : on construit récursivement le préfixe du mot (en allant vers la gauche si horizontal ou vers le haut si vertical), en posant des lettres du tirage sur les cases libres et en suivant les lettres déjà présentes sur le plateau. À chaque étape, si une transition `!` existe dans le GADDAG, on bascule vers la phase d'extension à droite. (3) `avance_depuis` : on complète le suffixe du mot (à droite ou en bas), à partir de l'état du GADDAG obtenu après avoir rencontré `!`. Cette phase respecte également les lettres déjà posées sur le plateau et les lettres encore disponibles du tirage. La validité des lettres posées est systématiquement vérifiée par `verifie_lettre`, assurant que chaque lettre placée respecte aussi les mots perpendiculaires. La fonction renvoie ainsi la liste complète des mots valides pouvant être posés depuis cette ancre, en prenant en compte les contraintes du tirage, des lettres déjà posées, du lexique (*via* le GADDAG), et des mots perpendiculaires.

14) 👤 Il faut toutes les ancres disponibles sur le plateau. Pour chaque ancre, et pour chaque sens de placement (horizontal et vertical) :

- appeler une fonction de génération de mots valides pour produire tous les mots possibles à partir de cette ancre, du tirage, et du GADDAG ;
- pour chaque mot trouvé, calculer son score ;
- parcourir tous les coups valides et garder celui avec le score maximal.

15) 💻

```
let meilleur_coup plateau tirage gaddag : ((int * int) * string *
    bool * int) =
  let ancres_total = ancres_plateau plateau in
  let rec explore (meilleur_pos, meilleur_mot, meilleur_dir,
    meilleur_score) ancres =
    match ancres with
    | [] -> (meilleur_pos, meilleur_mot, meilleur_dir,
    meilleur_score)
    | ancre :: rest ->
      let try_direction acc dir =
        let mots = trouver_coup plateau tirage gaddag ancre dir in
        List.fold_left (fun acc mot ->
            let score = valeur_mot plateau mot ancre dir in
            let (_, _, _, best_score) = acc in
            if score > best_score then
              (ancre, mot, dir, score)
            else
              acc
          ) acc mots
      in
        let best_after_vertical = try_direction (meilleur_pos,
    meilleur_mot, meilleur_dir, meilleur_score) false in
        let best_after_horizontal = try_direction
    best_after_vertical false in
        explore best_after_horizontal rest
  in
  explore ((-1, -1), "", false, -1) ancres_total
```

6 Statistiques des tournois

16) ✎ Pour reconstituer la liste des identifiants des joueurs à partir des données disponibles dans la table games, on peut extraire tous les identifiants de joueurs qui apparaissent comme gagnants ou perdants dans au moins une partie.

```
SELECT DISTINCT playerid
FROM (
  SELECT winnerid AS playerid FROM games
  UNION
  SELECT loserid AS playerid FROM games
);
```

17) 💻 On reconstitue la liste de tous les joueurs. On fait ensuite un `LEFT JOIN` entre joueurs et parties pour récupérer les victoires. Ceci permet de garder les joueurs qui n'ont jamais gagné. Enfin on agrège les résultats par joueur.

```
SELECT joueurs.playerid, COUNT(g.gameid) AS nb_victoires
FROM (
  SELECT winnerid AS playerid FROM games
  UNION
  SELECT loserid AS playerid FROM games
```

```
) AS joueurs
LEFT JOIN games g ON g.winnerid = joueurs.playerid
GROUP BY joueurs.playerid;
```

18) ✎ C'est le joueur d'ID 4001 qui a eu le plus grand nombre de voctoires 44.

```
SELECT joueurs.playerid, COUNT(g.gameid) AS nb_victoires
FROM (
  SELECT winnerid AS playerid FROM games
  UNION
  SELECT loserid AS playerid FROM games
) AS joueurs
LEFT JOIN games g ON g.winnerid = joueurs.playerid
GROUP BY joueurs.playerid
ORDER BY nb_victoires DESC
LIMIT 1;
```

19) ✎ Pour chaque joueur, il faut identifier la dernière partie qu'il a jouée (en tant que gagnant ou perdant). Puisque la colonne `date` ne contient pas l'heure, il est possible que plusieurs parties aient lieu le même jour. Pour obtenir la dernière partie exacte jouée par chaque joueur, on peut s'appuyer sur l'identifiant des parties, qui est croissant dans le temps.

```
SELECT playerid, newrating
FROM (
  SELECT gameid, winnerid AS playerid, winnernewrating AS
    newrating FROM games
  UNION ALL
  SELECT gameid, loserid AS playerid, losernewrating AS newrating
    FROM games
) AS participations
WHERE (playerid, gameid) IN (
  SELECT playerid, MAX(gameid)
  FROM (
    SELECT winnerid AS playerid, gameid FROM games
    UNION ALL
    SELECT loserid AS playerid, gameid FROM games
  )
  GROUP BY playerid
);
```

20) ✎ On écrit la requête SQL pour l'identifiant `playerid = 1234`.

```
WITH player_games AS (
  SELECT gameid, date, 1 AS is_win
  FROM games
  WHERE winnerid = 1234
  UNION ALL
  SELECT gameid, date, 0 AS is_win
  FROM games
  WHERE loserid = 1234
),
```

```sql
grouped AS (
  SELECT *,
         SUM(1 - is_win)
         OVER (ORDER BY gameid ROWS UNBOUNDED PRECEDING) AS
    defeat_block
  FROM player_games
),
streaks AS (
  SELECT defeat_block, COUNT(*) AS streak_length
  FROM grouped
  WHERE is_win = 1
  GROUP BY defeat_block
)
SELECT MAX(streak_length) AS max_consecutive_wins
FROM streaks;
```

7 Jeu en mode classique

21) Scrabble est un jeu d'accessibilité.

- Il y deux joueurs qui jouent en alternance sans hasard pendant le coup.
- On peut modéliser un état par le plateau accompagné des tirages en cours, les scores et lettres restantes.
- L'état initial est le plateau vide, le sac complet et le joueur 1 qui a 7 lettres à sa disposition.
- Les transitions sont réalisées par le joueur courant qui pose un mot valide et donc modifie l'état. Le prochain état dépend uniquement de l'état courant et l'action du joueur courant.
- L'objectif est de maximiser le score. Les deux joueurs tentent chacun d'accéder à un état favorable.

Le hasard est présent au moment du tirage (choix des lettres), mais pas dans la mécanique du tour de jeu lui-même, ce qui est conforme à de nombreux modèles de jeux avec hasard contrôlé, comme les jeux de stratégie tour par tour.

22) Pour rechercher une stratégie gagnante, on peut utiliser un graphe d'états et construire un arbre de parties possibles. Chaque nœud correspond donc à un état et chaque arête à un coupe valide. Chaque état est annoté avec l'état du plateau, le sac de lettres, les tirages des deux joueurs, les scores des deux joueurs et enfin l'information du joueur courant.

Le jeu se termine après un nombre fini de cours : quand le sac et les tirages sont vides. On peut utiliser la rétropropagation minimax avec un parcours exhaustif de tous les états possibles jusqu'à une profondeur donnée. Pour chaque feuille de l'arbre, on peut utiliser la fonction d'évaluation qui consiste à calculer la différence de scores (score du joueur 1 - score du joueur 2). Le joueur courant maximise son évaluation et l'adversaire la minimise. Si l'évaluation d'un nœud est favorable quel que soit le choix adversaire, on dit que le joueur a une stratégie gagnante depuis cet état. On

peut optimiser cette méthode avec l'élagage alpha-bêta pour éviter d'explorer des branches inutiles.

23) 💻 On peut donc modéliser le jeu avec les types suivants.

```
type joueur = J1 | J2

type etat = {
  plateau : Scrabble.plateau;
  sac : Alphabet.lettre list;
  tirage_j1 : Alphabet.lettre list;
  tirage_j2 : Alphabet.lettre list;
  score_j1 : int;
  score_j2 : int;
  joueur_courant : joueur;
}

(* ancre, mot, vertical/horizontal *)
type coup = (int * int) * string * bool
```

L'algorithme minimax peut être implémenté avec la fonction suivante où il faut définir les différentes fonctions qui manquent.

```
let rec minimax etat gaddag profondeur =
  if profondeur = 0 || (etat.sac = [] && etat.tirage_j1 = [] &&
    etat.tirage_j2 = []) then
    evaluer_etat etat
  else
    let coups = generer_coups_valides etat gaddag in
    let scores = List.map (fun coup ->
      let nouvel_etat = appliquer_coup etat coup in
      let prochain_joueur = if etat.joueur_courant = J1 then J2
    else J1 in
      let nouvel_etat = { nouvel_etat with joueur_courant =
    prochain_joueur } in
      minimax nouvel_etat gaddag (profondeur - 1)
    ) coups in
    match etat.joueur_courant with
    | J1 -> List.fold_left max min_int scores
    | J2 -> List.fold_left min max_int scores
```

Enfin il faut renvoyer une variable de type `coup` qui correspond au meilleur score calculé avec la fonction `minimax`.