Sorbonne Université – Licence informatique 2ème année
Fonctions et procédures de calcul – 2I008

Projet : Boulder Dash

Le but de ce projet est d'implanter en OCaml une version simplifiée du jeu *Boulder Dash*. Dans ce jeu, vous incarnez un personnage dont le but est d'explorer une cave tout en collectant des diamants sans se faire écraser par des rochers. Vous pouvez trouver une version jouable en ligne de ce jeu : `https://codeincomplete.com/games/boulderdash/play/`.

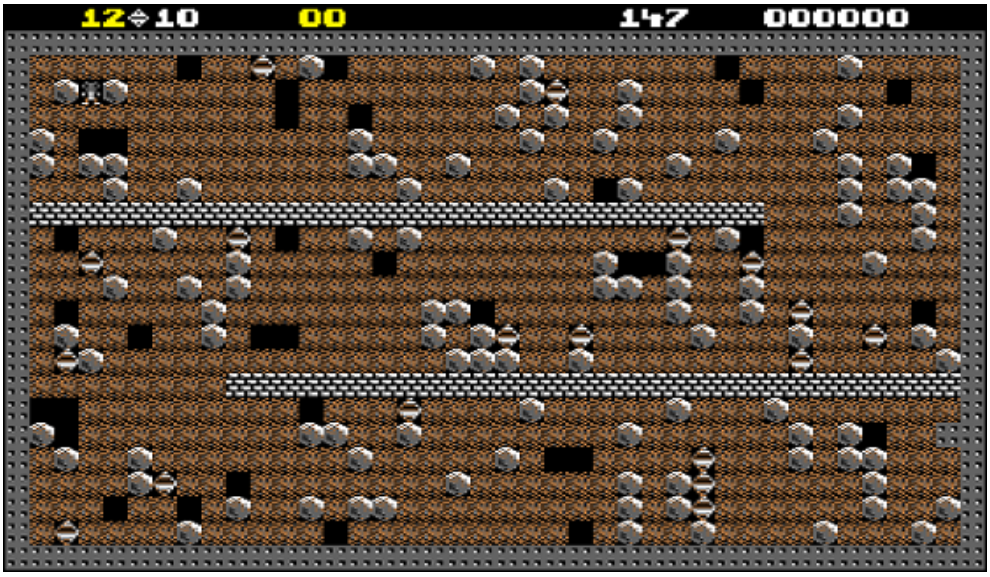

FIGURE 1 – Capture d'écran de la version en ligne de Boulder Dash

Structure du squelette Le code à compléter vous est fourni dans `/Vrac/2I008-1819/projet`. Des tests vous sont fournis pour vérifier votre implantation des structures demandées dans les deux prochaines sections. Ils sont localisés dans le dossier `test`, et vous pouvez les lancer avec la commande `make test`. Une fois que les types du jeu sont définis, vous pouvez compiler la totalité du project avec `make`, et l'exécuter avec `make run`.

Consignes de rendu

- Le projet est à faire seul ou en binôme. Il est fortement conseillé de faire le projet seul, notamment pour se préparer à l'examen et parce que les correcteurs seront moins exigeants.
- Nous vous rappelons que toute tentative de triche sera sévèrement sanctionnée.

Pour le groupe du mardi le projet est à rendre avant le mardi 23 avril à 22h00 (heure de Paris), par mail à raphael.monat@lip6.fr, avec comme sujet "[2I008] Projet". N'oubliez pas de nous préciser votre nom, prénom et numéro d'étudiant.

Pour le groupe du mercredi le projet est à rendre avant le mercredi 24 avril à 22h00 (heure de Paris), par mail à martin.pepin@lip6.fr, avec comme sujet "[2I008] Projet". N'oubliez pas de nous préciser votre nom, prénom et numéro d'étudiant.

Séance 1 : structure de données, types et affichage

But de la structure de données On cherche à représenter des lignes du plateau de notre jeu, qui peuvent être majoritairement vide. Au lieu d'implanter le plateau sous la forme d'une structure pleine telle qu'un tableau de tableau, nous allons donc chercher à définir une structure ayant une valeur par défaut pour tous les indices des cases, ces valeurs pouvant être changées localement sur certaines cases en utilisant une liste d'association.

Liste d'association sur les entiers avec valeur par défaut

Dans cette sous-section nous allons compléter le fichier `assoc.ml`. L'objectif est de représenter l'ensemble des fonctions f de $\mathbb{Z}$ dans un ensemble quelconque S, telle que f est constante sur $\mathbb{Z}$ (tout les entiers ont la même image par f, cette image est appelée `default`), sauf sur un sous-ensemble fini d'entier. Pour représenter une telle fonction, il suffit : (1) de donner la valeur `default` et (2) de donner l'image de tout les entiers n'ayant pas `default` comme image.

Le ficher `assoc.ml` contient donc la définition de type : `type 'a t = {default: 'a; assoc: (int * 'a) list}`. Le champ `default` indique la valeur par défaut de la fonction représenté, le champ `assoc` est une liste d'association *triée* indiquant les images de certains entiers.

Q1 – Définir la fonction `remove_assoc: int -> (int * 'a) list -> (int * 'a) list`, telle que `remove_assoc i l` prend une liste `l` triée et calcule une nouvelle liste d'association qui ne contient plus la clé `i` (et qui reste triée).

— Si `l = [(0, "a"); (1, "b")]` alors `remove_assoc 0 l = [(1, "b")]` et `remove_assoc (-1) l = l`

Q2 – Définir la fonction `constant : 'a -> 'a t`, telle que `constant x` calcule une structure représentant la fonction constante sur $\mathbb{Z}$, dont l'image de chaque entier est `x`.

Q3 – Définir la fonction `find : int -> 'a t -> 'a` telle que `find i m`, renvoie l'image de `i` par la fonction représentée par la structure `m`.

— Si `m = {assoc = [(1, "a"); (3, "b")]; default = "_"}` alors `find 1 m = "a"` et `find 2 m = "_"`.

Q4 – Définir la fonction `set : int -> 'a -> 'a t -> 'a t` telle que `set i v m` calcule une nouvelle structure qui associe `v` à `i` et est égale à `m` sinon.

— Si `m = {assoc = [(1, "a"); (3, "b")]; default = "_"}` alors `set 2 "c" m = {assoc = [(1, "a"); (2, "c"); (3, "b")]; default = "_"}`

— Si `m = {assoc = [(1, "a"); (3, "b")]; default = "_"}` alors `set 3 "_" m = {assoc = [(1, "a")]; default = "_"}`.

Q5 – Définir la fonction `fold : (int -> 'a -> 'b -> 'b) -> int -> int -> 'a t -> 'b -> 'b` telle que `fold f a b m init` calcule `f b m(b) (... (f (a+1) m(a+1) (f a m(a) init)) ...)`, où `m(i)` est l'image par la structure `m` de `i`. Il est demandé d'écrire une fonction n'utilisant pas de références.

— Si `m = {assoc = [(1, "a"); (3, "b")]; default = "_"}` alors : `fold (fun i x acc -> acc ^ (string_of_int i) ^ x) 0 3 "" = "0_1a2_3b"`.

Carte du jeu

On va maintenant utiliser les fonctions définies dans `assoc.ml` pour définir un type similaire aux tableaux usuels de `Ocaml`. Cette section va donc remplir le fichier `matrix.ml`. Un tableau contenant des éléments de type `'a` sera une structure contenant : une largeur (champ `larg`), une hauteur (champ `haut`), une association des entiers vers une association des entiers vers des éléments de type `'a`. Chaque ligne est représentée par une association d'entiers vers des éléments de type `'a` et la matrice est une association d'entiers vers des ligne.

On a donc le type :

```
type 'a t = {larg: int; haut: int; map: ('a Assoc.t) Assoc.t}
```

Q6 – Définir une exception `Out_of_bounds` qui sera levée dès qu'un accès en dehors de l'intervalle $[0, \texttt{larg} - 1] \times [0, \texttt{haut} - 1]$ a lieu.

Q7 – Définir une fonction `make: int -> int -> 'a -> 'a t` telle que `make larg haut default` calcule la matrice de largeur `larg`, de hauteur `haut`, contenant uniquement la valeur `default`.

Q8 – Définir une fonction `read: int -> int -> 'a t -> 'a` telle que `read i j m` calcule l'élément à la `i`ème ligne, et à la `j`ème colonne de `m`.

Q9 – Définir une fonction `set: int -> int –> 'a -> 'a t -> 'a` telle que `set i j v m` calcule une nouvelle matrice obtenue en mettant `v` à la `i`ème ligne, et à la `j`ème colonne de `m`.

Q10 – Définir une fonction `fold (int -> int -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b` telle que `fold f m acc` calcule `f haut larg m(haut)(larg) (... (f 0 1 m(0)(1) (f 0 0 m(0)(0) acc)) ... )`, où `m(i)(j)` est la valeur contenue à la `i`ème ligne, `j`ème colonne de `m`. On itérera donc sur les lignes, et pour chaque ligne on itérera sur toutes les colonnes. Il est demandé d'écrire une fonction n'utilisant pas de références.

Q11 – En utilisant la fonction `fold` définie précédemment, définir la fonction `iter : (int -> int -> 'a -> unit) -> 'a t -> unit` telle que `iter f m` appelle f sur chaque case de la matrice `m`.

Types

Maintenant que nous avons un type `'a Matrix.t` nous sommes en mesure de définir les types de base du jeu. Dans le fichier `type.ml` :

Q12 – Définir le type `cell`, qui décrit ce qui peut être contenu dans une case de la matrice : soit une pierre qui ne peut être bougée (stone), soit un rocher circulaire (boulder), soit de la terre (dirt), soit un diamant (diamonds), soit une case vide (empty).

Q13 – Définir la fonction `string_of_cell : cell -> str` nécessaire pour le débuguage du jeu.

Q14 – Le type de la carte `map`, est une matrice contenant des cellules. L'état complet du jeu `game` est décrit par un type enregistrement à deux champs : la carte `map`, et la position du joueur dans cette carte `player`. Définir ce type.

Q15 – Définir le type `dir` décrivant les quatre directions possibles de déplacement du joueur.

Q16 – Définir la fonction `string_of_dir : dir -> str` nécessaire pour le débuguage du jeu.

Affichage

On va définir les fonctions permettant l'affichage graphique de l'état du jeu. On modifie donc dans cette sous-section le fichier `drawing.ml`. Afin de faciliter le travail, un type `scale = (int * int) -> (int -> int)` est défini. Un élément de type `scale` est une fonction qui étant donné une coordonnée dans la matrice du jeu vous donne les coordonnées en pixel du coin en bas à gauche de la case sur le rendu graphique. La documentation du module Graphics est disponible ici : `https://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html`.

Q17 – Définir une fonction `draw_rect_cell: Graphics.color -> (int * int) -> scale -> unit`, telle que `draw_rect_cell c (i, j) scaler` dessine un carré de couleur `c`, ce carré représente les bords de la case `(i, j)`. Son coin bas gauche a donc pour coordonnée dans la fenêtre graphique `scale (i, j)` et son coin haut droite : `scale (i+1, j+1)`.

Q18 – De même, définir la fonction `fill_rect_cell` dessinant un carré plein.

Q19 – De même, définir la fonction `fill_diamond_cell` dessinant un losange plein.

Q20 – Enfin, définir la fonction `fill_circle_cell` dessinant un cercle plein.

Q21 – A l'aide des fonctions précédentes définir la fonction `draw_cell: cell -> (int * int) -> scaler -> unit` dessinant une cellule :

- un mur sera représenté par une case remplie de noir ;
- un rocher sera représenté par un cercle gris ;
- de la terre sera représentée par une case remplie de marron ;
- un diamant sera représentée par un losange de couleur bleutée ;
- une case vide sera... vide.

On prendra garde à dessiner le contour de chaque case.

Q22 – Définir une fonction `draw_map: map -> scaler -> unit` dessinant la carte complète du jeu.

Q23 – Définir une fonction `draw_player: (int * int) -> scaler -> unit` dessinant la position du joueur, il sera repéré par un cercle rouge.

Q24 – Définir une fonction `draw_game: game -> scaler -> unit` qui dessine la carte, la position du joueur et qui synchronise l'affichage.

Q25 – Définir les fonctions `init_graphics: unit -> unit`, qui ouvre une fenêtre et enlève la synchronisation automatique et `reinit_graphics` qui supprime le contenu de la fenêtre graphique.

Séance 2 : dynamique du jeu

Durant cette séance, nous allons nous intéresser aux fonctions qui régissent le *moteur* du jeu. Nous allons en particulier compléter `game.ml`.

Q26 – Si vous ne l'avez pas déjà fait, jouez au jeu en ligne (et observez le mouvement des rochers).

Un tour de jeu consiste en l'application successive d'un déplacement du personnage, puis de la chute possible des rochers. **Dans la suite, on suppose que le bord du plateau de jeu est toujours rempli de pierres qui ne peuvent pas bouger.**

Déplacement du personnage

Q27 – Définissez la fonction `position_after_move : int * int -> dir -> int * int` qui étant donné des coordonnées et une direction renvoie les coordonnées du personnage après déplacement dans la direction indiquée.

On rappelle que la première coordonnée correspond à la hauteur du joueur dans la carte

Q28 – Définissez la fonction `player_turn : game -> dir -> game` qui calcule la nouvelle position du joueur et si cette nouvelle position est valide :

- Si la case est vide, déplace le joueur et renvoie le nouveau plateau de jeu
- Si la case est de la terre ou un diamant, déplace le joueur et met à jour le plateau de jeu pour mettre une case vide à la place de l'ancienne case
- Si la case est une pierre qui ne peut pas être bougée, la fonction n'a pas d'effet
- Si la case est une pierre qui peut être poussée dans une case vide, il faut mettre à jour les contenus des cases pour bouger la pierre et renvoyer la nouvelle position du personnage. Attention : les pierres ne peuvent **pas** être soulevées par le joueur, juste poussées à gauche ou à droite.

Chute des rochers

Nous allons maintenant chercher à simuler la chute des rochers : un rocher situé au-dessus d'une case vide va tomber dans cette case. Un rocher situé au-dessus d'un autre rocher roulera à gauche ou à droite de ce rocher si les cases à côté et en diagonale par rapport à ce dernier sont vides.

La figure 2 représente ce comportement. N'hésitez pas à le vérifier dans le jeu en ligne !

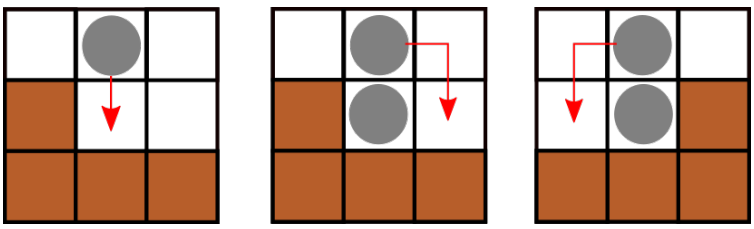

FIGURE 2 – Chute des rochers

Q29 – Définissez la fonction `is_empty : int * int -> game -> bool` qui renvoie vrai si et seulement si la case en question est vide. On fera attention à bien considérer la position du joueur en plus du contenu de la map.

Q30 – Définissez la fonction `position_after_fall : game -> (int*int) -> (int*int)` telle que `position_after_fall g (i,j)` retourne la position du rocher situé en `(i,j)` dans `g` après application *d'une étape* des règles énoncées ci-dessus. Si le rocher ne peut ni tomber ni rouler, la fonction retournera les coordonnées `(i,j)` inchangées.

Q31 – Définissez la fonction `move_boulder_step : game -> (int * int) -> game` qui, à l'aide de la fonction précédente, fait tomber (ou rouler) un rocher.

Après déplacement du rocher, la fonction devra vérifier que le personnage ne se trouve pas juste sous la nouvelle position de ce rocher. Si c'est le cas, alors le personnage meurt (car il est écrasé), et la fonction retourne l'exception `Dead`.

Q32 – Définissez la fonction `find_movable_boulder : game -> (int * int) option` qui parcourt la carte du jeu, et retourne *un* des rochers qui doivent bouger (i.e. tomber ou rouler) pendant ce tour de jeu. Cette fonction devra faire usage d'une des fonctions de parcours du module `Matrix`.

Il est par ailleurs demandé de prendre en compte le fait qu'il est coûteux et inutile de continuer à parcourir la carte dès lors qu'on a trouvé un rocher "déplaçable".

Q33 – Définissez la fonction `world_turn : game -> game` qui déplace tous les rochers déplaçables jusqu'à ce qu'il n'y en ait plus, et renvoie alors le nouveau plateau de jeu. Pour voir l'évolution des déplacements des rochers, on réaffichera le monde entre chaque déplacement. Vous utiliserez la fonction `Unix.sleepf : float -> unit` pour faire une courte pause entre tous ces affichages.

Diamants et calcul du score

Dans notre version, le but du jeu est de récolter tous les diamants présents dans un monde.

Q34 – Ajoutez un champ `diamonds` au type `game` représentant le nombre de diamants qui restent à récolter.

Q35 – Dans `game.ml`, ajoutez une fonction `win : game -> bool` qui indique si la partie est gagnée.

Q36 – Définissez la fonction `count_diamonds : cell Matrix.t -> int` qui compte, à l'aide de la fonction `Matrix.fold` le nombre de diamants dans la carte.

Q37 – Dans le fichier `parse.ml`, mettre à jour la fonction `parse_file` afin d'initialiser le jeu avec le nombre total de diamants.

Q38 – Mettre à jour la fonction `draw_game` afin d'afficher le nombre de diamants restant à récolter.

Q39 – Mettre à jour la fonction `player_turn` afin de décrémenter cette valeur lorsque le joueur récupère un diamant. Si le joueur a gagné, la fonction devra lever l'exception `Win`.

Noix

Dans *Boulder Dash*, quand un rocher tombe sur une noix, elle est transformée en diamant.

Q40 – Ajouter un constructeur noix (walnut) au type des cellules, et mettre à jour la fonction `string_of_cell`.

Q41 – Mettre à jour `draw_cell` afin d'afficher les noix.

Q42 – Mettre à jour la fonction `count_diamonds` pour prendre aussi en compte les diamants cachés dans des noix.

Q43 – Mettre à jour la fonction `move_boulder_step` pour remplacer une noix sur laquelle un rocher tombe par un diamant.

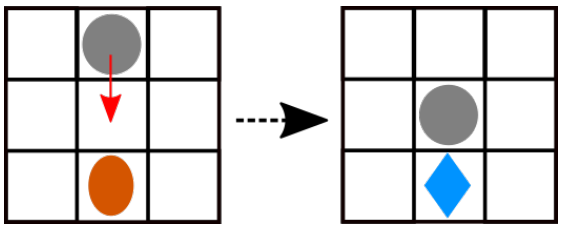

FIGURE 3 – Ouverture d'une noix

Fonction principale

Q44 – Modifiez la fonction principale du jeu (dans `main.ml`) pour que soit affiché à l'écran "Gagné" (resp. "Perdu") quand le joueur ramasse tous les diamants (resp. quand il meurt).

Q45 – Modifier la fonction principale du jeu pour que l'appui sur la touche `X` du clavier quitte le jeu.

Bonus : autres fonctionnalités

Vous êtes libres d'implanter d'autres fonctionnalités au projet, qui seront comptabilisées (en fonction de la difficulté de leur implémentation) comme des questions bonus. Vous pouvez par exemple ajouter une porte de sortie, limiter le temps qu'a le joueur, créer différent type de pierres précieuses, ajouter de la dynamite, ajouter un éditeur de niveaux, ajouter le moyen d'annuler des coups, ... Dans ce cas, documentez les fonctionnalités que vous avez ajoutées dans un fichier `README.md`, en précisant les règles que vous utilisez et quelles parties du code vous avez modifiées.