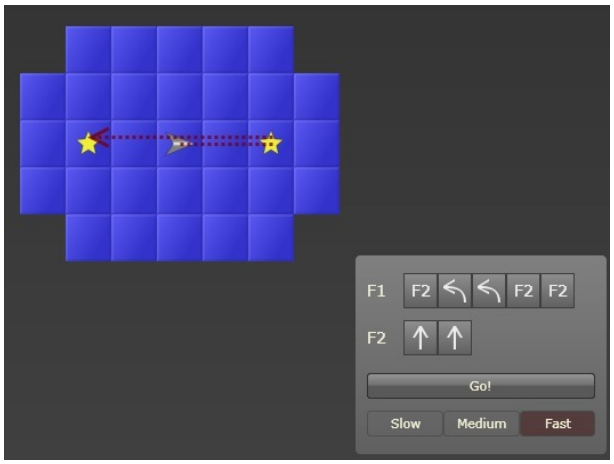
Feuille d'exercices n°9

Projet - Robozzle

Robozzle est un jeu de programmation dans lequel vous dirigez un petit robot chargé de ramasser des étoiles disséminées sur une grille colorée. Mais, pour cela, pas de souris, clavier ou autres formes de contrôles interactifs : vous devez programmer le comportement de votre robot.



Nous allons, au cours du projet, implanter l'affichage des niveaux et l'interprétation des programmes du robot.

Les niveaux Les niveaux de Robozzle définissent le **terrain** sur lequel le robot peut circuler, le nombre de fonctions que le robot pourra utiliser et leurs tailles (en nombre d'instructions). Sont aussi définies la position initiale du robot ainsi que sa direction et enfin la position des étoiles sur le terrain. Un niveau sera représenté par un fichier `.map`. Le fichier `niv0.map` donne une description textuelle d'un niveau très simple.

Les programmes Un programme sera composé de fonctions qui seront elles-mêmes composées de plusieurs instructions. On dispose de :

- Trois instructions de **mouvement** : avancer d'une case, faire un quart de tour vers la gauche et faire un quart de tour vers la droite.
- Une instruction permettant de **colorier** la grille.

- Une instruction permettant d'**appeler une des fonctions** qui composent votre programme.

Chaque instruction i peut être associé à une couleur c. Cette couleur fonctionnera comme une condition d'exécution de i: le robot n'effectuera l'instruction i que si la case sur laquelle il est placé est de couleur c. Un programme sera représenté par un fichier `.prog`.

Architecture du code Le répertoire `src` contiendra le code du projet. Notre projet se découpe en 3 modules principaux :

- `Niveau` (fichier `niveau.ml`) qui définit la représentation des niveaux.
- `Programme` (fichier `programme.ml`) qui définit les programmes et leur évaluation
- `Vue` (fichier `vue.ml`) qui gère l'affichage.

Nous aurons, en plus, un module `Main` qui sera le point d'entrée du programme et un module `Adj` que l'on définira dans cette première séance. Vous êtes libres de définir vos propres modules mais cela nécessitera de modifier le `Makefile` pour qu'il les prenne en compte. On vous fournira, au cours du projet, des exemples de niveaux (`.map`) et de programmes de robot (`.prog`). Ces fichiers se trouveront dans le répertoire `exemples`. Vous êtes libres d'ajouter vos propres exemples dans ce répertoire.

Sera également fourni un module `Parsing` (fichier `parsing.ml`) permettant, à partir de fichiers `.map` et `.prog`, de construire la représentation OCaml correspondante.

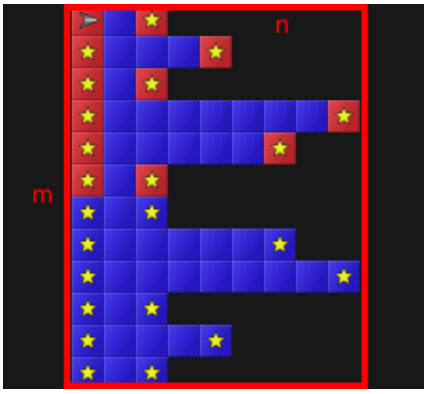
Vous pouvez compiler les sources en tapant `make` dans un terminal dans le répertoire `src`. Cela produira un exécutable `robozzle` que vous pourrez lancer sur un `.map` et un `.prog` comme dans l'exemple suivant : `./robozzle ../exemples/niv0.map ../exemples/niv0.prog`

Première séance

Durant la première séance, nous allons nous intéresser à la construction des niveaux et à leur affichage. Nous allons pour cela commencer par implanter une structure de liste d'adjacence.

Exercice I : Listes d'adjacences ordonnées

Le premier exercice consiste à définir une structure de donnée efficace pour représenter les cases des niveaux.



Nous avons ici un exemple de niveau. Une première idée serait de le représenter avec une matrice de taille $m \times n$. Néanmoins, avec cette méthode, on est obligé de réserver $m \times n$ espaces mémoire de taille fixe. Comme ce niveau est "creux" (comme la majorité des niveaux de Robozzle), il y aurait de nombreuses cases sans contenu, ce qui n'est pas efficace d'un point de vue mémoire.
L'idée est alors de ne stocker uniquement les coordonnées auxquelles il y a véritablement des cases. On choisit donc une représentation de liste d'adjacence.

Pour cela, nous allons implanter notre propre module de liste d'adjacence que nous maintiendrons triée pour améliorer la complexité des opérations.

Dans le fichier `adj.ml`, on définit une structure de liste d'association dont les clés seront des entiers. Ces clés seront ordonnées dans l'ordre croissant et représenteront les coordonnées des cases.

```
type 'a t = (int * 'a) list
```

Notre niveau est en 2 dimensions. Il nous faut donc des matrices d'adjacence ordonnées. Il s'agit d'une liste d'adjacence qui à chaque abscisses associera la liste d'adjacence des ordonnées. Un élément de cette matrice est donc un couple (i, l_o). où i représente le numéro de ligne et l_o est une liste d'association représentant les cases de la i-ième ligne.

```
type 'a matrix = ('a t) t
```

Par exemple, sur les fichiers `niv0.map` et `niv1.map`, les listes obtenues devront être :

- `niv0.map`

```
[(0,[(0,None); (1,None); (2,None); (3,None); (4,None); (5,None)]
```

- `niv1.map`

```
[(0 , [(6,Bleu); (7,Rouge)]);
 (1 , [(5,Bleu); (6,Bleu)]);
 (2 , [(4,Bleu); (5,Bleu)]);
 (3 , [(3,Bleu); (4,Bleu)]);
 (4 , [(2,Bleu); (3,Bleu)]);
```

```
(5 , [(1,Bleu); (2,Bleu)]);
(6 , [(0,Bleu); (1,Bleu)]);
(7 , [(0,Vert)])]
```

On commence par implanter des opérations sur les **listes d'adjacence**. Ouvrez le fichier `adj.ml`.

Q1 –Définissez la fonction `mem: int -> 'a t -> bool` qui vérifie si il y a un élément à l'indice spécifié dans la liste passée en paramètre. On remarquera que l'on a pas besoin de parcourir toute la liste si on rencontre un indice plus grand que celui passé en paramètre.

Q2 –Définissez la fonction `get: int -> 'a t -> 'a` qui récupère l'élément à l'indice spécifié dans la liste passée en paramètre. La fonction lèvera l'exception `Not_found` si il n'y a rien à cet indice.

Q3 –Définissez la fonction `set: int -> 'a -> 'a t -> 'a t` qui retourne la liste passée en paramètre à laquelle on a ajouté un couple (clé,valeur). Si la clé est déja présente dans la liste, on replace la valeur associée par celle passée en paramètre. On fera attention à préserver l'ordre lors de l'insertion.

Nous allons maintenant définir ces opérations sur les **matrices d'adjacence**. Vous devez bien sûr utiliser les fonctions définies aux questions précédentes.

Q4 –Définissez la fonction `mem_matrix: (int * int) -> 'a matrix -> bool` qui vérifie si il y a un élément à la position spécifiée dans la matrice passée en paramètre.

Q5 –Définissez la fonction `get_matrix: (int * int) -> 'a matrix -> 'a` qui récupère l'élément à la position spécifiée dans la matrice passée en paramètre. La fonction lèvera l'exception `Not_found` si il n'y a rien à cet indice.

Q6 –Définissez la fonction `set_matrix: (int * int)-> 'a -> 'a matrix -> 'a matrix` qui retourne la matrice passée en paramètre à laquelle on a ajouté l'élément donné à la position spécifiée. On fera attention à préserver l'ordre lors de l'insertion.

Q7 –Enfin, définissez une fonction `bornes: 'a matrix -> (int*int)*(int*int)`, qui retourne les dimensions maximales (abcisse_min,abcisse_max),(ordonée_min,ordonée_max) d'une matrice.

Les niveaux

Pour définir un niveau (voir le fichier `niveau.ml`), nous allons définir les types suivants:
Le type `couleur` qui représente la couleur d'une case. Une case est soit colorée soit transparente (cette couleur sera alors représentée par `None`).

```
type couleur = Vert | Rouge | Bleu | Jaune | None
```

On définit ensuite un type `orientation`, représentant la direction dans laquelle le robot "regarde".

```
type orientation = Haut | Bas | Gauche | Droite
```

Les coordonnées des cases sont représentées par un couple d'entier.

```
type coordonnee = int * int
```

On peut maintenant définir l'état d'un robot (sa position et sa direction).

```
type etat_robot = {
  pos : coordonnee;
  dir : orientation;
}
```

Enfin, on définit un niveau comme étant composée d'une grille (une matrice d'adjacence de `couleur`), l'état du robot, la liste des fonctions et la liste des positions des étoiles à récupérer.

```
type t = {
grille : couleur Adj.matrix;
robot : etat_robot;
fonctions : (string * int) list;
etoiles : (int*int) list;
}
```

Exercice II : L'affichage des niveaux

Ouvrez le fichier `vue.ml`. Celui-ci définit plusieurs références qui seront utiles pour l'affichage. Pour vous faciliter la tâche, nous vous fournissons une fonction `map_coord_to_graphics_coord` qui convertira des coordonnées du terrain en position de pixel dans la fenêtre graphique.

Q1 –Complétez le code des fonctions `dessine_case: (int*int) -> couleur -> unit`, `dessine_etoile: (int*int) -> unit` et `dessine_robot: etat_robot -> unit` du module `Vue`.

Q2 –Complétez la fonction `dessine_niveau: Niveau.t -> unit` qui, à l'aide des fonctions précédentes dessine entièrement un niveau.

Q3 –Définissez la fonction `init: Niveau.t -> (int*int) -> unit` qui crée une fenêtre graphique aux dimensions égales aux valeurs passées en paramètre. Elle mettra aussi à jour les valeurs des références de la façon suivante :

- `min_x, max_x, min_y max_y` reçoivent les dimensions de la grille (Il vous faudra utiliser la fonction `Adj.bornes`)
- `largeur` et `hauteur` reçoivent les dimensions de la fenêtre
- `hauteur_haut` et `hauteur_bas` définissent l'espace de la fenêtre alloué à la pile (partie haute) et au terrain (partie basse). Vous pouvez les assigner respectivement à 25% et 75% de la hauteur totale.

Exercice III : Manipulation des niveaux

On va maintenant implanter les opérations sur les niveaux.

Q1 –Pour commencer, implanter les fonctions `rot_gauche: orientation -> orientation` (resp. `rot_droite: orientation -> orientation`), qui à partir d'une orientation calculent l'orientation obtenue en effectuant un quart de tour vers la gauche (resp. droite). Vous définirez ensuite les fonctions correspondantes `set_robot_direction_gauche: t -> t` et `set_robot_direction_doite: t -> t` qui, étant donnée un niveau, change la direction du robot de ce niveau.

Q2 –Définissez la fonction `avancer : etat_robot -> etat_robot` qui, étant donnée un robot, fait avancer celui-ci d'une case dans la direction dans laquelle il "regarde". Définissez également la fonction `robot_avancer : niveau -> niveau` qui, étant un niveau, change la position du robot de ce niveau.

Q3 –Définissez la fonction `test_couleur: couleur -> niveau -> bool` qui teste si le robot du niveau est sur une case de la couleur passée en paramètre. Si la couleur passée en paramètre est `None`, cette fonction renvoie vrai.

Q4 –Définissez la fonction `get_couleur: (int*int) -> niveau -> couleur` qui renvoie la couleur de la case passée en paramètre. Définissez également la fonction `set_couleur: (int*int) -> couleur ->(int*int) t -> t` qui change la couleur de la case passée en paramètre.

Q5 –Pour finir, définissez la fonction `case_valide : niveau -> bool` qui vérifie que la position courante du robot existe dans le niveau (i.e. qu'elle ne fait pas partie de la partie "creuse" du niveau).

Exercice IV : L'évaluation des programmes

Important : Les exercices précédents ont défini des fonctions qu'il convient de réutiliser dans la suite.

Nous allons maintenant modifier le fichier `programme.ml`.

Q1 –Lire et comprendre les types en entête de fichier

Lors de l'exécution du programme le robot maintient une pile de commande qu'il doit effectuer. Cette pile de commande est encodé par un élément de type `sequence`.

Q2 –Implanter la fonction `pile_initiale : programme -> sequence`. La première commande à mettre dans la pile est un appel à la fonction se trouvant en premier dans le programme.

Q3 –Implanter la fonction `trouve_fonction : string -> programme -> sequence`. Cette fonction doit renvoyer la liste des commandes contenues dans une fonction

Q4 –Implanter la fonction `est_fini : niveau -> bool` testant si le joueur a ramassé toutes les étoiles

Q5 – Implanter la fonction `une_etape : programme -> niveau -> sequence -> (niveau * sequence)`, telle que `une_etape : prog etat pile` effectue la prochaine instruction sur la pile `pile`

et renvoie le nouvel état et la nouvelle pile.

Cette fonction doit :

- Lever les bonnes exceptions quand il y a lieu : le robot atteint une position invalide ou la pile est vide.
- Enlever les étoiles si le robot se trouve sur une étoile
- Comparer la couleur liée à l'instruction à effectuer avec la couleur sous la position du robot.

  si le test de couleur réussi, on exécute l'instruction

  sinon on ne modifie pas l'état (l'instruction n'est pas exécutée)

  De plus si l'instruction est un appel de fonction, on empile les instructions contenues dans la fonction appelée à la pile, dans tout les autres cas on retire de la pile l'instruction en haut de pile.

Q6 –Implanter la fonction `verifie : programe -> niveau -> unit`, cette fonction a pour but de stopper l'exécution de votre programme si le niveau et le programme ne correspondent pas. Cette fonction fera les tests suivants :

- Le robot ne se trouve pas dans une position invalide
- Les étoiles ne sont pas dans une position invalide
- Les fonctions contenues dans le programme n'ont pas une taille plus grande que ce qui est autorisé par le niveau.
- Les fonctions contenues dans le programme n'ont pas un nom non autorisé par le niveau.

Si un des test est faux, on lèvera une exception "informative". On rappelle que `failwith "erreur"` permet de lever une exception, avec le message d'erreur `"erreur"`.

Exercice V : L'affichage de la pile

Q1 –Finissez votre interface graphique en complétant les fonctions de dessin de la pile. Pour vous faciliter la tâche on limitera le nombre maximum de cases de pile à afficher (la valeur `pile_max` définira cette limite)

Exercice VI : Le main

Q1 –Complétez le code de la fonction `loop` qui sera votre boucle d'interaction : vous y gérerez :

- l'affichage du jeu,
- le calcul de l'état suivant lorsqu'on appuie sur une touche,

- la vérification que la partie n'est pas terminée.

Le tout en faisant attention aux exceptions qui peuvent survenir.