BORDEAUX INP Enseirb-Matmeca

Robot Programming

Soulié Alexandre

Paul Maël

Ladagnous Louis-Victor

Lohézic Victor

Département Informatique

Table des matières

1 Introduction

1.1 Présentation du sujet et objectif

Dans ce projet, nous allons devoir concevoir un langage de programmation permettant de faire évoluer un robot dans un monde avec des objectifs à remplir. Ce langage sera fourni avec un évaluateur et une interface graphique montrant le robot en action.

1.2 Cadre de travail

On peut séparer le travail en deux parties différentes : M.Lohézic et M.Ladagnous s'occuperont de la partie langage tandis que M.Paul et M.Soulié se chargeront de l'évaluateur.

2 Analyse des problèmes rencontrés

2.1 Conception du langage

Le langage doit concevoir toutes les options que l'on souhaite ajouter au jeu. Il s'agit de définir les types de déplacements du robot dont le plateau affecte ces déplacements. La présence d'obstacles interfère avec la façon dont le robot se déplace et il a donc fallu réfléchir aux interactions du robot avec les composants du plateau. Il s'agit aussi de considérer les objectifs à atteindre tels que la récupération d'étoiles sur le plateau et la façon dont ces différents objectifs peuvent impacter le robot et son environnement.

2.2 Conception de l'évaluateur

L'évaluateur devra marcher via une structure de pile d'appel. Cet élément est celui qui provoquera le plus de disruptions. Le principal problème a été de comprendre la structure de pile d'appel. Après discussions, nous nous sommes rendu compte que notre implémentation de pile d'appel ne correspondait pas aux attentes.

Il semblerait que la méthode précédemment utilisée relevait plus de l'extension in-line, et rendait donc notre code moins fonctionnel. L'extension in-line est une technique de programmation consistant à remplacer la fonction par le corps de cette dernière[1]. Cette technique était celle qui nous était venue le plus rapidement, mais elle s'avère hautement non fonctionnelle. Il aura donc fallu repenser la pile d'appel pour correspondre aux attentes du sujet.

Un autre défi était de proposer un code le plus pur possible. Il aura donc fallu réfléchir aux solutions pour éviter les effets de bords.

Enfin, le choix des structures de données pour représenter un programme et ses fonctions a été un choix délicat, notamment lié aux changements de structure de pile d'appel.

2.3 Génération de Puzzle

Il s'agit de trouver comment générer un puzzle qui est résoluble. La génération aléatoire d'un monde ne permet pas de garantir la possibilité de résoudre le monde. Une stratégie possible est d'utiliser l'évaluateur en lui donnant un programme. Néanmoins, des questions restent en suspens : quel programme générer pour l'évaluateur ? et comment ? Quelles sont les conditions qui permettent de stopper l'évaluation du programme ? De plus, les générations du monde et du programme doivent être réalisées de la manière la plus pure possible.

2.4 Conception de l'afficheur

Il a tout d'abord fallu réfléchir à la forme que l'on voulait qu'ait notre afficheur. Nous avons dû aussi penser aux différentes interactions entre l'utilisateur et notre application. Celle-ci devait être intuitive et montrer aux mieux les différentes parties sur lesquelles nous avons travaillé comme le fonctionnement de l'évaluateur. Ensuite, il s'agit aussi d'utiliser les différents modules utilisés et de mettre à l'épreuve la modularité de notre projet sous la contrainte de la programmation fonctionnelle.

2.5 Conception du solveur

En ce qui concerne la conception du solveur, il a fallu réfléchir à une solution simple de représenter un programme et à des méthodes de sélection des programmes. En effet, on ne peut pas tester l'ensemble des possibilités pour résoudre un puzzle car cela résulte en une explosion combinatoire.

3 Conception des solutions algorithmiques

3.1 Langage de programmation

Pour implémenter les déplacements du robot, il faut tout d'abord décider de l'implémentation d'un puzzle en JavaScript. Nous avons choisi d'implémenter un puzzle sous la forme d'un plateau, la position initiale et la direction

initiale du robot sur le plateau.

$$\textbf{Puzzle} \begin{cases} \textbf{board :} & \text{plateau} \\ \textbf{robotRowInit :} & \text{indice} \\ \textbf{robotColInit :} & \text{indice} \\ \textbf{robotDirInit :} & \text{direction} \end{cases}$$

Nous avons fait le choix de considérer la direction du robot comme un entier partant de 0 et allant jusqu'à 3. La figure 1 ci-dessous représente la convention choisie pour représenter la direction du robot.

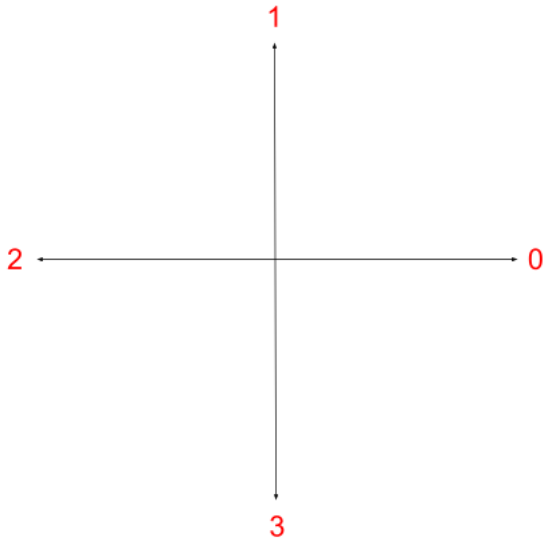


FIGURE 1 – Convention pour les directions du robot

Ensuite, nous avons choisi de représenter notre plateau comme une matrice de dictionnaire, chaque case de la matrice représentant une case du plateau. Chaque case du plateau correspond à un dictionnaire possédant les champs *color*, *achievement* et *obstacle*. Le champ *color* représente la couleur présente sur cette case tandis que les champs *achievement* et *obstacle* représentent l'objectif et l'obstacle présent ou non sur la case. Ce choix permet de manipuler et de visualiser le plateau de manière simple. En effet, un dictionnaire représentant chaque case nous permet de librement indiquer les informations que l'on souhaite sur une case, de manière très explicite.

Ainsi, nous avons pu implémenter les fonctions de base correspondant au mouvement du robot d'une case en avant, les rotations horaires et anti-horaires. Dans un second temps, nous avons aussi ajouter une fonction *teleport* permettant au robot de se téléporter plusieurs cases vers l'avant.

Le robot est implémenté sous la forme d'un dictionnaire avec les champs *row*, *col* et *dir* correspondant à la position du robot sur le plateau et à la direction dans laquelle il est orienté. Les fonctions *move* et *teleport* fonctionnent de manière quasi-identique, on remarque qu'un mouvement d'un case en avant correspond en fait à une téléportation d'une case en avant. Les deux fonctions se traitent donc de la même manière. Il s'agit de vérifier que le mouvement et la couleur passés en paramètre sont valides par rapport au robot. Les fonctions $rotateLeft$ et $rotateRight$ nécessitent

$$nouvelleDirection = (ancienneDirection + 3)\%4 \tag{1}$$

$$nouvelleDirection = (ancienneDirection + 1)\%4 \tag{2}$$

seulement la vérification de la couleur passé en paramètre. La formule 1 calcule la nouvelle direction du robot en fonction de l'ancienne lors d'une rotation horaire, la formule 2 correspond au cas d'une rotation anti-horaire.

La programmation de langage étant pure, toutes les fonctions du langage renvoient un nouveau dictionnaire représentant la nouvelle instance du robot.

Finalement, afin de mener à bien nos tests sur tous les fichiers, une fonction *cons* a été rajoutée au langage. Celle-ci permet de générer un dictionnaire à partir de trois paramètres, *color*, *achievement* et *obstacle*. Ainsi, il est simple de générer des plateaux pour rigoureusement tester nos fichiers.

3.2 Type abstrait de données : la liste et la pile

Nous avons tout d'abord implémenté une liste.

Nom	Liste	
Utilise	élément, liste, booléen, entier, chaîne de caractères	
Opérations		
cons	élément × élément	⟶ liste
head	liste	⟶ élément
tail	liste	⟶ liste
isEmpty	liste	⟶ booléen
listDisp	liste	⟶ chaîne de caractères

Une liste est un type de données inductif.

$$\textbf{Liste} \begin{cases} \textbf{car :} & \text{element} \\ \textbf{cdr :} & \text{liste} \end{cases}$$

Il est implémenté en JavaScript sous la forme d'un dictionnaire avec les champs car et cdr. Par conséquent, la liste est une paire pointée. Le champ

cdr contient un dictionnaire qui peut être soit un dictionnaire représentant une liste, soit un dictionnaire vide qui représente la liste vide. Toutes les fonctions sont pures car sans effet de bord. L'accès aux champs d'un dictionnaire sont réalisés en temps constant. La fonction cons est un constructeur de liste. On comprend alors pourquoi la liste est définie de manière inductive, la liste vide est une liste, si e est une valeur et l une liste alors cons(e, l) est une liste. La fonction head renvoie l'élément en tête de la liste tandis que tail renvoie la liste sans l'élément en tête. La fonction isEmpty vérifie que la liste soit bien vide. Par conséquent, toutes les opérations ont une complexité en temps constante sauf l'opération listDisp qui parcourt toute la liste grâce à une fonction récursive pour afficher la liste.

La pile est un type abstrait qui fonctionne de manière analogue à une pile de livre. Pour retirer, le livre tout en bas, on est obligé de dépiler tous les livres précédents. Lorsqu'on souhaite ajouter un livre au-dessus de la pile, on est obligé de l'ajouter au-dessus de cette dernière.

Nom	Pile	
Utilise	élément, liste, booléen, entier	
Opérations		
stackCreateEmpty		$\longrightarrow$ liste
stackIsEmpty	pile	$\longrightarrow$ booléen
stackPush	élément $\times$ pile	$\longrightarrow$ liste
stackPop	pile	$\longrightarrow$ pile
stackPeek	pile	$\longrightarrow$ élément
stackDisplay	pile	$\longrightarrow$ item
stackAppend	pile $\times$ pile	$\longrightarrow$ pile
stackReverse	pile	$\longrightarrow$ pile
stackLength	pile	$\longrightarrow$ entier

Notre implémentation de la pile repose sur la liste que nous avons implémentée précédemment. La fonction stackCreateEmpty s'effectue en temps constant puisqu'elle renvoie une liste vide. La fonction stackIsEmpty a la même complexité puisqu'elle vérifie si la liste est vide. La fonction stackPush ajoute un élément en haut de la pile en temps constant grâce au constructeur de la liste. La fonction stackPop retire le dernier élément de la pile en temps constant en appelant la fonction tail de la liste. De manière analogue, la fonction stackPeek renvoie l'élément en haut de la pile en temps constant. La fonction qui permet d'afficher la pile se réalise en temps constant. La fonction stackAppend est une fonction récursive qui permet d'ajouter une pile au-dessus d'une seconde pile. Si la pile qu'on souhaite placer au-dessus

de la seconde possède N éléments alors la complexité en temps est $O(N)$. Notre fonction stackReverse est une fonction récursive terminale. Elle encapsule une fonction récursive reverse prenant en paramètres notre pile initiale et une autre pile qui sera inversée. Ainsi si la pile initiale est vide, on renvoie la pile inversée, sinon on appelle de nouveau notre fonction récursive mais on dépile la pile initiale et on ajoute l'élément dépilé à la seconde pile passée en paramètres. Dans la fonction stackReverse, il y a un appel à la fonction récursive avec pour premier paramètre la pile à inverser et second paramètre une pile vide. En enfermant la fonction récursive dans stackReverse, on masque les paramètres inutiles au client. De plus, on remarque que la fonction récursive est récursive terminale, soit elle renvoie une valeur finale, soit elle fait un appel récursif en réécrivant ses paramètres. L'avantage d'une telle écriture, c'est que toute fonction récursive terminale peut être réécrite de manière à ne pas faire exploser la pile d'appel. Cependant, les machines actuelles Javascript ne réalisent pas ces optimisations. On bénéficie tout de même des propriété de la pureté. La complexité en temps de cette fonction est linéaire en le nombre d'éléments de la pile. La fonction stackLength est une fonction récursive qui renvoie la taille de la pile avec une complexité en temps linéaire en nombre d'éléments de cette pile.

Les structures inductives ont une influence sur les algorithmes. La liste étant une structure inductive, les algorithmes ont aussi une structure inductive. Notre pile reposant sur cette liste est aussi soumise à cette influence. Cela explique que les algorithmes des opérations ont tendance à se ressembler, notamment avec leur forme récursive. Aucune des opérations de la pile ou de la liste ne réalise des effets de bords. La liste et la pile sont donc implémentées de manière pure et sont donc immutables. Ce sont donc des structures de données fonctionnelles. Elles sont donc forcément persistantes, c'est-à-dire ces structures de données persistent en mémoire à ses traitements. Le principal avantage est que les données sont indépendantes et par conséquent persistent. L'indépendance nous permet de les réutiliser, c'est ce que l'on fait pour stackAppend. L'inconvénient est le coût mémoire car les instances ont tendance à s'accumuler au fur et à mesure.

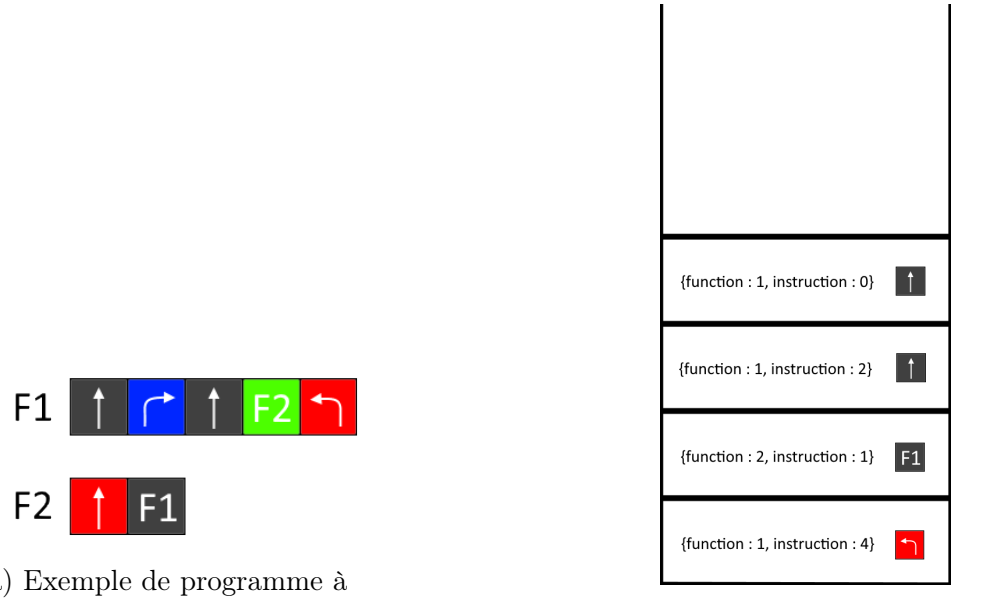
3.3 Évaluateur du langage

L'évaluateur se trouve sous la forme d'une unique fonction, prenant en paramètres le programme à exécuter et le puzzle. Un programme est un tableau de tableaux d'instructions, chacun de ces tableaux les plus internes représentant $F1$, $F2$, etc... Les instructions sont conservées sous forme congelée dans ces tableaux afin de les réactiver au moment de l'exécution. La fonction

d'évaluation contient un grand nombre de fonctions en son sein, privées car utilisées uniquement dans l'évaluateur. Une de ces fonctions est majeure, car elle définit le caractère récursif terminal de notre fonction d'évaluation. Cette fonction accepte une multitude de paramètres :

— Les coordonnées du robot
— Un compteur d'appels d'instructions (au sens de notre langage)
— Un compteur d'appels de fonctions (au sens de notre langage)
— La pile d'appel
— Un dictionnaire répertoriant les *achievements* obtenus
— Une pile d'appel dans une version plus visualisable pour aider au débuggage

Prenons ce moment pour nous arrêter sur la pile d'appel. Après être passés au travers des problèmes de compréhension (voir 2.2), nous avons opté pour une pile d'appel grâce à un module de pile qui elle-même marche via un module de listes chaînées génériques. Dans notre pile d'appel se trouve des dictionnaires contenant le numéro de la fonction ainsi que l'indice de l'instruction actuelle. Le but était de recréer des pointeurs de fonction, pour s'approcher au maximum d'une pile d'appel classique. On peut rapprocher le numéro de la fonction comme l'adresse d'une fonction et l'indice de l'instruction comme le numéro de la ligne à exécuter dans cette fonction, avec parfois des appels à d'autres fonctions.



(a) Exemple de programme à deux fonctions inter-dépendantes

(b) Exemple de pile d'appel

FIGURE 2 – Illustration des structures de données utilisées

Tout notre module de pile possède sa propre interface, avec des fonctions pures. Cela implique que l'entièreté du code de l'évaluateur est pure, donc par définition sans effet de bord. Cette propriété a été fondatrice pour l'élaboration de ce code afin de profiter pleinement du cadre fonctionnel du projet.

Cet évaluateur peut renvoyer des erreurs de tous types, ou le booléen *True* lorsque le programme donné en entrée résout le puzzle.

C'est aussi dans l'évaluateur que l'on implémente les mécanismes de jeu, notamment l'étoile à ramasser. On implémente le fait de terminer le puzzle lorsque le robot récolte l'étoile en passant sur la case correspondante, mais on se permet d'ajouter des obstacles. Chaque case de la carte est décomposée en 3 catégories : *color*, *achievement* et *obstacle* (c.f. 3.1). Chaque obstacle est référencé sous la forme d'une chaîne de caractère dans la clé *obstacle*. Voilà les principales idées qui ont été implémentées ou non :

- `Jumppad` : Quand le robot passe sur une case de type `jumppadX`, le robot saute de 3 cases vers la direction X
- `Spikes` : Un tour sur deux, les pics du `spike` se lèvent, si le robot s'y trouve dessus, il échoue alors
- `Bouton Laser` : Les case `laser` empêchent le robot de passer. Lorsque le robot passe sur une case `button`, il désactive ainsi les lasers jusqu'a la fin de l'évaluation (**non implémenté**)
- `Armure` : Lorsque le robot passe sur une case `armor`, il s'équipe d'une armure et est immunisé aux dégâts d'autres obstacles, une seule fois dans l'éxecution (**non implémenté**)

Les `jumppad` et `spike` implémentés ne posent pas de grands soucis de programmation et ne demande que des changements assez mineurs. Par exemple, les `spike` fonctionnent de pair avec un argument de notre évaluateur, en regardant la parité du nombre d'instructions effectuées. La paire `bouton/laser` et l'`armure` n'aurait aussi pas posé de grands soucis de programmations et d'algorithmique. Ces facilités sont dues grâce à la présence d'un dictionnaire qui sert de base pour tracer si oui ou non la clé à été récupérée à n'importe quel moment de l'exécution. Il suffit de rajouter des champs à ce dictionnaire et de le mettre à jour correctement à chaque appel récursif pour garder une trace de nos obstacles et des collectionnables.

3.4 Génération de Puzzle

La stratégie mise en place est la coloration d'un monde à l'aide de l'évaluateur qui exploite un programme généré aléatoirement.

Tout d'abord, nous nous sommes intéressés à la génération d'un monde vide. La fonction réalisant cette tâche est pure. En effet, aucun effet de bord n'est réalisé. Le monde est représenté sous la forme d'un tableau. On utilise la méthode from qui s'applique sur l'objet "Array". D'après sa documentation, elle crée une nouvelle instance de tableau à partir d'un objet itérable. L'objet itérable choisi est un tableau de 12 cases. Ces cases constituent en réalité les lignes de notre monde. from prend en paramètre une fonction qui s'applique sur chaque élément. Par conséquent, on passe en paramètre la fonction from qu'on applique sur un tableau de taille 16, ce qui nous permet de créer les colonnes et d'obtenir un tableau de dimension 12×16.

On compose avec une dernière fonction d'ordre supérieur qui se nomme map. Elle permet d'appliquer une transformation à chaque élément du tableau. On compose alors avec une deuxième fonction map. Ainsi, nous itérerons sur les lignes, puis les colonnes de chaque ligne. On utilise alors un constructeur de la section 3.1 qui génère une case vide de notre monde.

Cette composition de fonctions permet d'obtenir un code concis sur une ligne et pur donc sans dépendance. En revanche, il nécessite deux parcours du tableau au lieu d'un seul en programmation impérative. Une fonction génère un puzzle à partir de ce monde en positionnant le robot initialement au centre du monde. En effet, expérimentalement, on se rend compte que les puzzles générés sont plus complexes puisque le robot a moins de chance de quitter le monde rapidement.

L'évaluateur utilise des fonctions qui sont congelées. Par conséquent, une fois que la fonction est décongelée, nous ne pouvions pas accéder aux paramètres de la fonction congelée. Or nous avons besoin de la couleur pour pouvoir changer la couleur de la case du monde. Nous avons donc réalisé une fonction qui permet de générer un tableau de couleur de dimension 5×5. De manière analogue, à l'initialisation du monde, nous avons utilisé la méthode from pour créer un tableau de dimension 5×5.

On compose avec map pour les tableaux de taille 5 représentant les lignes. Cela nous permet de créer une nouvelle ligne avec des couleurs générées de manière aléatoire prises dans un tableau de couleurs fourni en paramètre de la fonction. La fonction map appelle une fonction qui génère un nombre aléatoire qui a 80% de chance de produire le dernier indice du tableau couleur. Cette probabilité permet d'avoir la couleur bleue majoritairement présente ce qui est le cas d'une grande partie des puzzles de Robozzle. Néanmoins, il est possible d'ajouter plus de couleurs si nécessaire, en modifiant les paramètres de la fonction qui génère l'indice du tableau.

La génération de programmes fonctionne de manière analogue avec, à la

place d'un tableau de couleurs, un tableau avec les instructions congelées et des dictionnaires pour les fonctions.

Ensuite, on a repris la structure de l'évaluateur. On a ajouté deux fonctions encapsulées qui se nomment updateBoard et addLastAchievement. Le plateau est modifié à chaque appel de fonction ou évaluation d'instructions grâce à updateBoard. Cette fonction parcourt le monde en composant deux fonctions map pour pouvoir itérer sur toutes les cases du monde. La fonction updateBoard prend aussi en paramètre les coordonnées de la case à modifier. Ainsi, lorsque les coordonnées de la case itérée ne correspondent pas, nous renvoyons la case actuelle, sinon on construit une nouvelle case de la couleur associée à l'instruction ou fonction. Il est possible de passer en paramètre à cette fonction des obstacles ou des conditions de succès comme une étoile. L'avantage est que le monde n'est pas modifié puisque map renvoie un nouveau tableau sans aucun effet de bord. Cependant, on est obligé d'itérer sur tout le tableau pour la simple modification d'une valeur dans un tableau. Une copie profonde du tableau consitue une autre solution possible. Cela permet ensuite d'accéder à l'élément avec les bonnes coordonnées en temps constant. Mais l'opération de copie est lourde. De plus, la fonction n'est plus pure mais elle référentiellement transparente. La fonction addLastAchievement agit de manière analogue mais ajoute une étoile sur une case. Dans l'évaluateur, sa fonction récursive principale evalRec est modifiée pour prendre le monde en paramètre. A chaque appel récursif, on passe en paramètre le monde que l'on met à jour avec la fonction updateBoard. Et lorsque l'on se trouve dans les cas limites de l'évaluateur, on renvoie un dictionnaire correspondant à un puzzle avec le monde qui est mis à jour avec la fonction qui permet d'ajouter une dernière condition de succès (lastAchievement).

3.5 Afficheur

La première étape était d'analyser les différents objectifs du projet et d'identifier les fonctions et contraintes de l'afficheur. Il doit être en mesure de montrer l'exécution d'un programme ainsi que la pile d'appel liée à cette exécution. Il nous a semblé nécessaire d'ajouter un bouton pour lancer l'exécution. Une partie de notre projet est consacrée à la génération de puzzle. Un autre bouton est alors nécessaire pour cette génération. On peut alors imaginer qu'en appuyant de nouveau sur le bouton start, notre algorithme permettra de résoudre des puzzles et qu'il exécute sa solution. Nous avons alors réalisé une maquette de ce que nous voulions 3 :

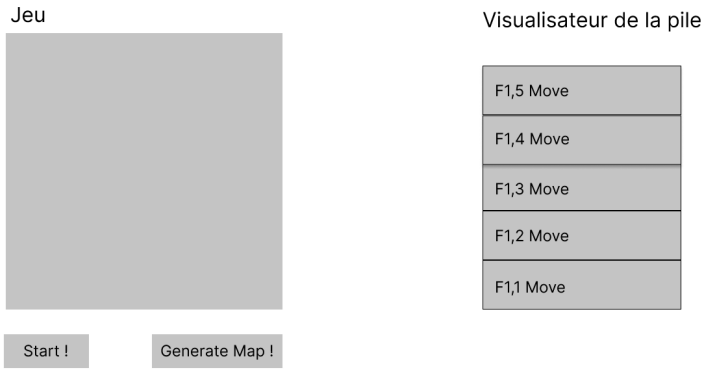


FIGURE 3 – Maquette de l'interface graphique

Ensuite, les parties HTML et CSS du site ont ainsi été développées. Dans la partie html, chaque élément possède un identifiant de la forme elt-i. L'élément en bas de la pile possède un identifiant avec $i = 0$. Chaque fois que l'on monte dans la pile, on incrémente i. Il restait à développer la partie en JavaScript en s'inspirant du modèle proposé par le sujet.

Notre première idée fut d'importer l'évaluateur présenté à la section 3.3 du fichier eval.js. L'évaluatueur est donc composé d'une fonction récursive possédant dans sa signature un dictionnaire contenant la position du robot et la pile d'appel actuelle. L'inconvénient est que notre module masque cette partie de code. L'encapsulation de la fonction récursive empêche d'avoir accès aux positions intermédiaires du robot et aux différents états de la pile d'appel. En effet, la durée de vie de la position du robot ou de la pile d'appel correspond au temps entre l'appel de la fonction récursive de l'évaluateur et de son prochain appel récursif. Par conséquent, l'évaluateur ne peut obtenir que la position finale et la dernière pile d'appel qui ne nous intéressent pas. Par conséquent, nous avons pensé à ajouter un paramètre à l'évaluateur une pile de piles d'appel. Cette pile contient les différents états de la pile d'appel. Dans cette pile d'appel nous avons ajouté la position du robot afin que l'afficheur puisse l'actualiser. Lors de l'implémentation de cette idée, on a rencontré des problèmes. En effet, le fonctionnement de l'évaluateur était tel que lorsqu'il actualisait la pile d'appel pour un appel de fonctions, la position du robot n'était pas valide. En effet, l'évaluateur ne pouvait pas évaluer les fonctions qu'il ajoutait dans la pile alors qu'il y avait d'autres instructions à exécu-

ter auparavant. De plus, la complexité temporelle de cette solution ne nous satisfaisait pas. En effet, tout d'abord on évalue le programme. Supposons que cela se fait avec $N \in \mathbb{N}$ appels de la fonction récursive de l'évaluateur. Ensuite, l'afficheur parcourt avec une autre fonction récursive, la pile des états de la pile d'appel. De plus, une fonction récursive parcourt chaque état de la pile d'appel pour l'affichage. C'est regrettable d'avoir une complexité $O(2N + \prod_{i=1}^{N} \text{taille de la pile à l'état } i)$.

Nous avons donc décidé de perdre en qualité modulaire. Nous avons copié l'évaluateur dans le fichier JavaScript de l'afficheur. Le problème d'encapsulation disparaît alors. Cela nous permet d'accéder aux différentes valeurs des positions des robots et aux états de la pile d'appel. Désormais la position du robot est synchronisée avec la pile d'appel. Dans la fonction récursive de l'évaluateur sont ajoutées deux fonctions qui permettent d'actualiser la position du robot et d'afficher la pile d'appel. La fonction permettant d'afficher la pile encapsule une fonction récursive permettant d'afficher chaque élément de la pile. Cette fonction est appelée avec une pile d'appel qui est inversée. Ensuite, on regarde l'élément au-dessus de la pile. On modifie le CSS de l'élément $i = 0$ pour afficher la bonne instruction ou fonction. On rappelle notre fonction avec i incrémentée et la pile dépilée. Cependant, l'exécution de l'affichage sans délai entre les appels de la fonction récursive de l'évaluateur ne permet pas de distinguer visuellement la position du robot et la pile.

Nous avons alors utilisé une promesse. Une promesse Ecmascript est la réalisation d'un calcul asynchrone. L'utilisation de ce concept avec evalRec permet d'obtenir un délai avant l'appel de la prochaine fonction. Pour cela, on ajoute le mot clé async devant la signature de la fonction récursive de l'évaluateur. Cela signifie que notre fonction renvoie toujours une promesse. De plus, nous avons codé une fonction sleep qui prend en paramètre un temps en ms et qui renvoie une promesse. Cette promesse renvoyée par sleep enveloppe la méthode setTimeout (elle exécute une fonction au bout d'un certain temps). On précède l'appel de la fonction sleep dans la fonction récursive par le mot clé await. Il permet d'attendre que la promesse renvoyée par sleep soit réalisée.

D'autres fonctions ont été ajoutées pour pouvoir représenter les puzzles générés et afficher si le programme permet bien de résoudre le puzzle.

Afin de réaliser, une démonstration du projet, nous nous sommes rendu compte qu'il était nécessaire d'ajouter des boutons pour pouvoir choisir les

puzzles de démonstration. Nous avons finalement obtenu, l'interface graphique suivante 4 :

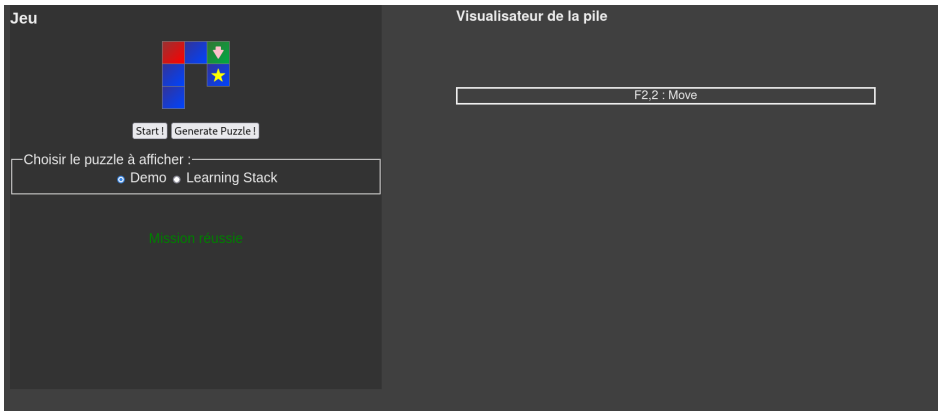


FIGURE 4 – Interface graphique obtenue

La gestion des interactions de l'utilisateur requiert l'utilisation de la programmation événementielle. C'est un style de programmation manipulant le flot de contrôle autour des événements. Ici, notre page HTML est décrit sous la forme d'un état global. L'exécution d'un programme est associé à l'événement : "Cliquer sur le bouton start !". La génération de puzzle correspond à l'événement : "Cliquer sur le bouton Generate Puzzle". Les fonctions permettant d'exécuter un programme ou de produire un puzzle sont des Callback. Ainsi, chaque traitement est rattaché à un événement, et opère une transformation de l'état. De plus, le comportement de la fonction qui exécute le programme a un comportement étendu par l'état de la page. En effet, si l'utilisateur clique sur le bouton "Learning Stack" ou "Demo", le programme et le puzzle exécutés sont différents.

Toutes les fonctions de l'afficheur ne sont pas pures, ni réferentiellement transparentes. En effet, des effets de bords sont nécessaires pour l'affichage du robot ou de la pile. Mais ils sont limités puisque la fonction qui permet de parcourir la pile pour l'afficher ou la vider est récursive. De plus, pour remplir le plateau du puzzle des fonctions d'ordre supérieur sont utilisées, notamment la fonction forEach. Celle-ci permet de faciliter l'écriture de l'algorithme et de parcourir chaque case du plateau pour lui associer son contenu. L'avantage d'une telle fonction est de pouvoir composer les forEach pour pouvoir traiter les cases de chaque ligne.

3.6 Solveur

Dans cette partie nous allons étudier la mise en place de méthodes pour résoudre des puzzles. En effet, il est non seulement intéressant de savoir si un puzzle est résoluble, mais il est aussi intéressant d'obtenir sa solution.

Le solveur fonctionne de manière simple, il prend en entrées un puzzle, un tableau avec le numéro des fonctions que l'on peut utiliser (dans la version actuelle du solveur ce tableau est nécessairement égal à [“$f1$”, “$f2$”, “$f3$”]), ainsi qu'un tableau avec le nombre d'instructions maximum par fonction. Ensuite, on génère des programmes de longueur 1, c'est-à-dire une seule instruction, représentés par des chaînes de caractères de même longueur où un caractère correspond à une instruction définie dans le fichier $solutionSwitch.js$. Par exemple, le “0” correspond à l'instruction “$move$” sur n'importe quelle couleur (“” = $gray$) et pour la fonction “$f1$”. Ces chaînes de caractères sont générées à l'aide d'une fonction de génération $generate$ qui les stocke dans un tableau de longueur égale à 16 si seule $f1$ peut contenir des instructions, égale à 40 si $f1$ et $f2$ peuvent contenir des instructions, et enfin égale à 72 si $f1$, $f2$ et $f3$ peuvent contenir des instructions. En effet, la fonction $f1$ peut contenir pour un numéro d'instruction donnée un $move$, un $rotateLeft$, un $rotateRight$, un appel à $f1$, un appel à $f2$ et un appel à $f3$ sur n'importe quelle couleur (“” ($gray$), $blue, red, green$), ce qui constitue 24 instructions différentes pour une seule fonction. Cependant, lorsque seule $f1$ est utilisée, ce qui correspond au cas où le nombre d'instructions de $f2$ et celui de $f3$ sont fixés à 0, on élimine les appels à $f2$ et $f3$ des possibilités ce qui donne les 16 instructions différentes, le fonctionnement est le même lorsque seul le nombre d'instructions de $f3$ est fixé à 0, on supprime les appels à $f3$ des possibilités. Les chaînes de caractères ainsi générées sont ensuite transformées en programme lisible par l'évaluateur grâce à la fonction $strProgram$.

Les programmes sont ensuite testés par un évaluateur modifié qui retourne un compteur qui correspond au nombre de cases parcourues par le robot, ou à -1 dans le cas où l'exécution se termine sur une erreur, comme par exemple dans le cas où le robot sort du puzzle. Dans le cas où certains programmes résolvent le puzzle, on extrait ces programmes et on renvoie les chaînes de caractères ayant permises de les générer, on peut ainsi connaître la solution du puzzle en faisant correspondre les caractères de la chaîne avec leur instruction correspondante dans la fonction $strProgram$, on peut également retransformer ces chaînes de caractères en programme en utilisant cette même fonction.

Si aucun des programmes générés n'a permis de résoudre le puzzle et que l'on se trouve dans le cas où seule $f1$ contient des instructions, on effectue une sélection des chaînes de caractères au meilleur compteur d'avancement,

ce qui signifie qu'on garde seulement les chaînes de caractères ayant permis de générer les programmes faisant avancer le robot le plus loin possible. Dans le cas où plusieurs fonctions peuvent contenir des instructions, on sélectionne les chaînes de caractères dont le compteur est compris entre le compteur de plus grande valeur moins le nombre d'instructions maximal pour une fonction et le compteur de plus grande valeur. Cette sélection permet de diminuer le nombre de possibilités, dans le but d'éviter l'explosion combinatoire.

On regénère ensuite des nouvelles chaînes de caractères en augmentant la longueur des chaînes sélectionnées de un caractère, le tout pour chaque caractère possible, ce qui augmente considérablement le nombre de possibilités à tester (ce nombre est égal au nombre de chaînes sélectionnées multiplié par le nombre de caractères). On répète le processus de sélection et regénération jusqu'à trouver une solution au puzzle, ou jusqu'à remplir chaque fonction. Dans le cas où chaque fonction est remplie et où on n'a pas trouvé de programme résolvant le puzzle, le puzzle est déclaré non résoluble.

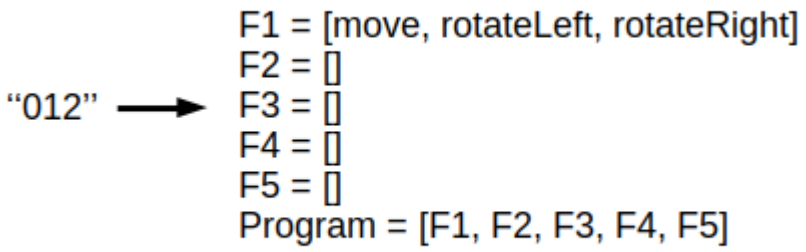


FIGURE 5 – Exemple de conversion d'une chaîne de caractères en programme

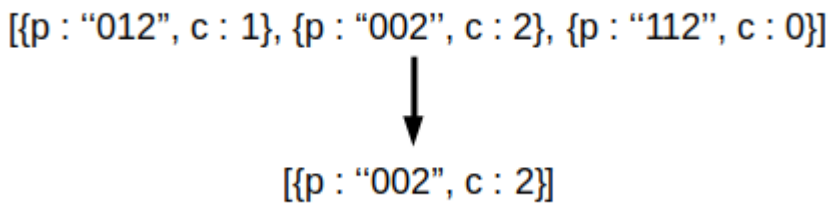


FIGURE 6 – Exemple de sélection au meilleur compteur

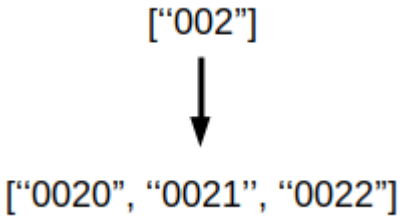


FIGURE 7 – Exemple de regénération à partir d'une chaîne sélectionnée

4 Tests

Les tests sont une partie importante du développement dans un projet informatique. Ils permettent notamment d'assurer le bon fonctionnement des différentes fonctions, ainsi que de déceler les bugs présents dans le code.

Pour vérifier le comportement de nos fonctions, nous avons pour chaque fichier $file.js$, un fichier $file.test.js$ contenant les tests unitaires de toutes les fonctions de $file.js$. Dans le projet, nous avons fait en majorité des tests unitaires, cela nous permet de s'assurer du bon comportement des fonctions en les prenant séparément, mais pas de vérifier qu'elles fonctionnent lors de leur utilisation conjointe. Nous avons également effectué des tests d'intégration pour l'évaluateur qui nous ont permis de tester conjointement le bon fonctionnement des différentes instructions définies dans le fichier *langage.js*, le bon fonctionnement des fonctions d'utilisation de la pile définies dans *stack.js*, ainsi que le bon fonctionnement de l'évaluateur lui-même. Il aurait cependant fallu faire des tests fonctionnels pour tester et valider un module dans son entièreté. Cependant, le visionnement de parties types grâce à l'afficheur permet de vérifier visuellement le bon fonctionnement des fonctionnalités de base telles que la construction d'un puzzle, ou encore le fonctionnement correct de l'évaluateur avec l'affichage de la pile d'appel. L'utilisation de *jest* et de l'option *--coverage* permet également de vérifier les lignes testées dans nos différents tests et donc de valider leur comportement.

5 Limites

Des tests pourraient être ajoutés pour l'afficheur. En effet, les tests permettent d'augmenter la qualité du code, néanmoins, ils sont coûteux en temps. Or, nous voulions obtenir un prototype pour pouvoir faire une démonstration de notre projet, par conséquent on n'a pas eu le temps d'ajouter

les tests pour la partie affichage. Pour les modules que l'on a testé, on a réalisé des tests unitaires, c'est à dire que l'on a testé les fonctions de ces modules. Il restent des tests à ajouter pour améliorer la qualité de notre code, par exemple les tests fonctionnels. Ce sont des tests qui permettent de valider notre code au niveau d'un module. Des tests de recette seraient aussi intéressants pour valider notre code cette fois-ci au niveau applicatif.

L'évaluateur est utilisé dans la partie de génération de puzzle, la partie pour résoudre ces puzzles et pour l'affichage. On a réalisé une copie de sa structure dans chacune de ces parties. Il pourrait être intéressant de rendre le code de l'évaluateur plus générique en transformant cette fonction en fonction d'ordre supérieur. On peut imaginer lui passer en paramètres des fonctions qui traitent les cas limites puisque les cas limites sont communs aux trois parties mais ne nécessitent pas le même traitement.

L'afficheur pourrait être améliorer en proposant un programme pour résoudre les puzzles générés aléatoirement. De plus, si jamais l'utilisateur ne réussit pas à le résoudre, on pourrait lui proposer un programme solution et l'exécuter devant lui.

Dans le solveur, d'autres stratégies de sélection pourrait être explorées pour optimiser le temps de calcul et endiguer le phénomène d'explosion combinatoire. Une méthode pour restreindre le nombre d'instructions différentes pour une fonction pour restreindre les possibilités pourrait également contribuer à améliorer le solveur.

Nous avons choisi de ne pas utliser TypeScript car lors du dernier projet, nous avons utilisé le langage C qui est typé. Nous avons vu alors en ce projet l'opportunité de tester sa réalisation avec un langage non typé. Les types peuvent servir de documentation minimale. Or, nous avons documenté le code avec le type des variables par conséquent, nous n'en avons pas ressenti le besoin ([2]). Cependant, le manque d'outils de vérification pour assurer des propriétés de sûreté liées par exemple au type a impacté le développement. Par exemple, des bugs dans le code ont pris du temps à être trouvés car un paramètre n'était pas passé dans des fonctions congelées. Ce manque de vérification rend le debug plus difficile et provoque du retard pour des erreurs d'inattention. Le bénéfice de pouvoir produire du code plus rapidement en est fortement réduit.

6 Conclusion

Ce projet a permis de nous mettre à l'épreuve sur les nouvelles techniques et compétences obtenues en cours de programmation fonctionnelle. Certains aspects de la programmation fonctionnelle se sont révélés être plus difficile à respecter que d'autres, notamment la pureté. Néanmoins, la plus grande réussite de ce projet réside peut-être dans le fait que notre code est quasi-intégralement pur.

7 Annexes

[1] `https://en.wikipedia.org/wiki/Inline_expansion`

[2] `https://vlohezic.gitlab.io/documentation_robot_programming/`