LFSAB1402: Project 2017: BomberSnake Oz

Maxime Istasse, Guillaume Maudoux, Charles Thomas, Hélène Verhaeghe

Novembre 2017

Durant le début de ce quadrimestre, les assistants n'ont pu s'empêcher de remarquer la fascination de plusieurs étudiants quant à la présence d'un jeu *Snake* dans *Emacs*, l'éditeur conseillé pour programmer en Oz. Votre déception qu'il ne soit pas écrit en Oz aurait presque suffit à ternir l'ambiance des premières séances de TP. Nous l'avons bien remarqué, et avons ressenti qu'il était de notre devoir d'y remédier.

C'est pourquoi nous avons voulu écrire pour vous un *Snake*, en Oz. Multijoueur, avec des bombes et des bonus. Le *Snake* dernier cri. S'il a beaucoup plus de potentiel que celui présent dans *Emacs*, il aura néanmoins besoin de votre intervention pour fonctionner.

Contexte

Le moteur de jeu est presque totalement implémenté, il reste juste à ajouter deux fonctions dans un style déclaratif pour permettre aux *snakes* de décoder la stratégie qui leur est imposée, et de se mouvoir en fonction de cette dernière et des effets qui sont censés les affecter.

Consignes

Le projet se fait par groupe de 2 maximum. Tous les groupes doivent implémenter au minimum le suivi des instructions, le décodage des stratégies et les 3 effets de bonus obligatoires présentés ci-après. Tout ceci, avec le rapport, comptera pour au moins 14 points sur 20.

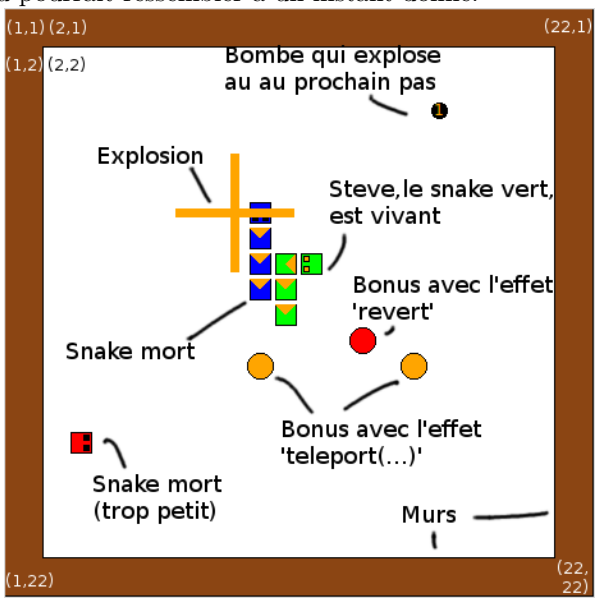
Une fois que vous êtes confiants de cette base, nous vous demandons d'étendre les possibilités du jeu en réalisant au moins 2 extensions qui décideront du reste de la note. Vous pouvez bien sûr implémenter autant d'extensions supplémentaire que vous le souhaitez. Veillez toutefois à vous concentrer sur le projet en lui-même. De nombreuses extensions ne pourront pas sauver un projet incomplet ou mal implémenté.

Vous serez évalués sur la qualité de votre code et de votre rapport. Nous attendons de vous un code correct, compréhensible (décomposé en sous-problèmes commentés) et relativement efficace (pensez au récursif terminal). Nous vous demandons d'utiliser uniquement les constructions déclaratives du langage (pas de `Cell` ni `Array`) et d'éviter la librairie standard (sauf `Adjoin` et `AdjoinAt` pour modifier vos snakes de manière plus flexible). Si vous n'arrivez pas à respecter ces conditions, nous vous demandons de le spécifier et de le justifier dans le rapport que vous devrez remettre avec votre code et votre scénario de test.

Enfin, les échanges de code entre étudiants de différents groupes sont interdits, excepté pour des scénarios. Notez cependant que le scénario final que vous nous remettrez devra être unique. Pour le reste, vous pouvez évidemment discuter et échanger des idées.

La base

Le jeu se déroule sur une grille de 22 cases de largeur et 22 cases de hauteur. Chaque case est représentée par les coordonnées (x, y) telles que $1 \leq x \leq 22$ et $1 \leq y \leq 22$. La coordonnée $(1, 1)$ correspond à la case au coin supérieur gauche (ou nord-ouest) de la zone de jeu. Chaque case de la grille peut être occupée par une partie d'un snake, une bombe, un bonus, un mur, ou plusieurs de ces choses. Voici, en pratique, ce à quoi le jeu pourrait ressembler à un instant donné:



Une partie se décompose en plusieurs pas. A chacun des pas sont exécutées les actions suivantes:

1. Dépôt des bombes par les *snakes* qui doivent le faire
2. Décrémentation du compte à rebours des bombes sur le plateau, celles à 0 explosent
3. **Mise à jour des *snakes* en fonction de leurs effets et de l'instruction prévue à ce tour** (c'est là que vous intervenez)
4. Ajout des effets des bonus sur les snakes et détection des morts. Un serpent meurt dans les cas suivants:
   - Sa tête entre en collision avec un autre serpent ou un mur;
   - Il touche l'explosion d'une bombe;
   - Il occupe strictement moins de 2 positions;
   - Il sort de la carte.

   A cette étape, le moteur du jeu regarde aussi si une équipe peut être déclarée vainqueur, auquel cas la partie s'arrête et le vainqueur est proclamé.

Pour chaque partie, le *scénario* définit les snakes qui vont être présents, les emplacements des bonus et les bombes éventuellement présentes dès le début. Chaque *snake* occupe différentes positions à un instant donné, chacune ayant son orientation dénotée par un des points cardinaux: `north`, `south`, `west` ou `east`. La première position spécifiée est sa tête et la dernière le bout de sa queue.

```
P ::= <entier tel que 1 <= x <= 22 >
direction ::= north | south | west | east

snake ::= snake(
```

```
        positions: [
            pos(x:<P> y:<P> to:<direction>) % Tête
            ...
            pos(x:<P> y:<P> to:<direction>) % Queue
        ]
        effects: nil
)
```

Chaque *snake* a une stratégie prédéfinie que vous devrez lui faire suivre et c'est là que le problème se situe actuellement. Premièrement, à chaque pas, le serpent va devoir suivre une instruction: `forward`, `turn(left)` ou `turn(right)`. Suivre cette instruction consiste à réorienter la tête du serpent dans la direction voulue (tout droit, à gauche ou à droite par rapport à sa direction actuelle) puis le faire avancer d'une case dans cette direction. C'est ce que votre fonction `Next` devra simplement faire dans un premier temps.

```
relative_direction ::= left | right
instruction ::= forward | turn(<relative_direction>)
```

Deuxièmement, le moteur de jeu attend, en guise de stratégie décodée pour chaque serpent, une simple **liste de fonctions** dont chacune prend uniquement en argument le *snake* et renvoit un record *snake* avec les nouveau attributs. La $n^{ième}$ fonction de la stratégie décodée de chaque serpent lui sera appliquée au pas n de la partie. Il va donc falloir écrire `DecodeStrategy` pour décoder les stratégies de la forme suivante

```
strategie ::= <instruction> '|' <strategie>
            | repeat(<strategie> times:<entier>) '|' <strategie>
            | nil
```

en une liste de fonctions dont chacune va utiliser votre fonction `Next` pour modifier le serpent avec l'instruction devant être exécutée à chaque pas. Notez qu'une stratégie à décoder accepte, en plus des instructions, des répétitions de sous-stratégies via le sucre syntaxique `repeat(...)` présenté un peu plus haut.

Récapitulatif

Pour commencer, vous devez implémenter deux fonctions dans `Code.oz`:

- `fun {Next Snake Instruction} ... end` qui doit appliquer l'instruction `Instruction` au serpent `Snake` et retourner le nouveau serpent. Sauf effet spécifiant le contraire, le serpent retourné doit avoir la même taille que celui reçu en paramètre, avoir réorienté sa tête et avoir avancé dans la direction spécifiée par l'instruction.

  Par exemple, pour les arguments `snake(positions:[pos(x:4 y:2 to:east) pos(x:3 y:2 to:east) pos(x:2 y:2 to:east)] effects:nil)` et `turn(right)`, le résultat devrait être `snake(positions:[pos(x:4 y:3 to:south) pos(x:4 y:2 to:south) pos(x:3 y:2 to:east)] effects:nil)`.

  Les effets seront expliqués dans la section suivante.

- `fun {DecodeStrategy Strategy} ... end` dont `Strategy` est spécifié sous le format `strategie` défini plus haut. `DecodeStrategy` convertit la stratégie en une liste de fonctions de la forme `fun {$ Snake} {Next Snake Instruction} end`. Chacune de ces fonctions renvoie le snake après application de l'instruction devant se passer à cette étape de la stratégie.

  Par exemple, `{DecodeStrategy [repeat([turn(right)] times:2) forward]}` pourrait calculer la stratégie simplifiée `[turn(right) turn(right) forward]` et devrait retourner une liste

de 3 fonctions `fun {$ Snake} {Next Snake Instruction} end` dont les deux premières appliquent l'instruction `turn(right)` sur le snake et la troisième applique dessus l'instruction `forward`.

Les effets de base

Les effets sont ajoutés à la liste associée à la feature `effects` du serpent par le moteur jeu lorsque le serpent a la tête sur un bonus. C'est à vous de les appliquer et de les supprimer de sa liste une fois pris en compte dans votre fonction `Next`. Le moteur de jeu va lui se contenter de faire croître cette liste. Les effets devraient être appliqués dans l'ordre dans lequel ils apparaissent dans la liste. Si plusieurs effets du même type sont dans la liste, seul le premier de chaque type devrait être effectif, les autres devraient être ignorés et vous devriez les supprimer.

- `grow`: le serpent devrait croître et donc occuper une position de plus à la fin de son mouvement. La tête bougera comme d'habitude.

Table 1: Evolution d'un serpent avançant tout droit, affecté par l'effet `grow` durant l'instant `2`

Instant 0	Instant 1	Instant 2	Instant 3
			

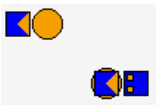
- `revert`: le serpent devrait être inversé avant la prise en compte de l'instruction. Sa tête devient donc sa queue et sa queue sa tête avant de bouger dans le sens de cette dernière.

Table 2: Evolution d'un serpent avançant tout droit, affecté par l'effet `revert` durant l'instant `2`

Instant 0	Instant 1	Instant 2	Instant 3
			

- `teleport(x:X y:Y)`: la tête du serpent (qui est sur la case du bonus) devrait être téléportée à la position `(X, Y)` avant la prise en compte de l'instruction, tout en conservant sa direction.

Table 3: Evolution d'un serpent avançant tout droit, affecté par l'effet `teleport(...)` durant l'instant `2`

Instant 0	Instant 1	Instant 2	Instant 3
			

Quelques conseils pour bien vous y prendre

Tout ce qui précède constitue donc la base du projet, et il vous est suggéré de procéder de la manière suivante:

- D'abord gérer les instructions de base, que vos *snakes* répondent bien dans le scénario `'scenario_test_moves.oz'`, qui ne comprend aucun bonus.
- Essayez de gérer l'effet `grow`, et testez avec le scénario `'scenario_test_grow.oz'`.
- Implémentez ensuite les effets `teleport(...)` et `revert` et testez les avec `'scenario_test_teleport.oz'` et `'scenario_test_revert.oz'` respectivement.

Dans ces scénarios de test, si tout se passe bien, *Steve*, le serpent vert, gagne toujours. Cela dit, la situation de départ est telle qu'une erreur d'implémentation devrait le faire perdre. Toutefois, survivre à ces scénarios ne garantit pas que votre implémentation soit tout à fait correcte. N'hésitez pas à écrire vos propres scénarios et à tester le jeu au clavier pour explorer plus de possibilités. (`'scenario_pvp.oz'` est un exemple de scénario permettant à deux joueurs de s'affronter avec les touches ←, →, `q` et `d` sur un terrain avec les 3 bonus de base) Vous êtes vivement encouragés à partager vos scénarios sur le forum Moodle.

Extensions

Comme précisé dans les consignes, nous vous demandons ensuite d'implémenter 2 extensions. A vrai dire, votre imagination est la seule limite. Nous vous proposons cependant quelques idées simples au cas où vous en manqueriez, ainsi qu'une piste pour faire des choses plus originales. Nous vous encourageons à sortir des chantiers battus, nous prendrons en effet cela en compte.

Notez que pour tester vos extensions, vous devrez créer votre propre scénario, ou en modifier un existant en ajoutant ou en remplaçant des bonus. Ces derniers ont une position, une couleur pour le rendu, un effet qui sera ajouté aux serpents affectés, et l'attribut `target` qui spécifie quels serpents vont être affectés après que l'un ait eu la tête sur le bonus. Dans la plupart de nos scénarios, `catcher` est la valeur attribuée à `target` pour tous les bonus, ce qui fait que le bonus s'applique aux serpents qui ont le même nom que celui qui est dessus. Néanmoins, `others`, `allies`, `opponents` et `all` sont d'autres valeurs possibles.

```
color ::= <n'importe quelle couleure reconnue par Tk, voir
           https://www.tcl.tk/man/tcl8.4/TkCmd/colors.htm>
bonus ::= bonus(position: pos(x:<P> y:<P>)
                color: <color>
                effect: grow | teleport(x:<P> y:<P>) | revert | ...
                target: catcher | others | allies | opponents | all
)
```

Gérer un terrain circulaire

Il est possible de créer un scénario sans murs sur les côtés (mettre l'attribut `walls` du scénario à `false`), ce qui veut dire que vos serpents ne se font pas tuer quand ils arrivent au bord de la fenêtre. Si la tête de l'un de vos serpents est censée dépasser de l'écran, nous vous proposons de la faire réapparaître de l'autre côté du terrain, afin de lui éviter la mort.

L'effet `shrink`

Il y a actuellement un effet qui fait grandir les snakes, mais rien pour les faire rétrécir. C'est ce que nous aimerions voir ajouté avec l'effet `shrink`. Le danger étant que certains serpents pourraient devenir trop petit que pour survivre!

L'effet `frozen(N)`

Cet effet empêcherait simplement le serpent de bouger durant `N` pas, ignorant les instructions et éventuellement les effets appliqués.

L'effet `confusion(N)`

Cet effet inverserait les commandes `turn(left)` et `turn(right)` durant `N` pas. Certainement l'un des plus drôles à rajouter à l'adversaire en multijoueur!

Et bien d'autres

Si vous voulez exploiter pleinement les possibilités du jeu, nous vous invitons à vous inspirer du code des scénarios existants pour voir un peu comment cela fonctionne. Vous remarquerez qu'en plus des attributs discutés plus haut, les snakes disposent d'une équipe (`team`, n'importe quelle couleur reconnue par Tk) et d'une stratégie prédéfinie pour déposer les bombes (`bombing`, une liste de booléen dont un est consommé à chaque pas, spécifiant si le *snake* dépose une bombe à l'endroit de sa queue à cet instant).

Il est tout à fait possible pour vous de créer des effets qui affectent ces attributs grâce à votre méthode `Next` comme vous le faisiez pour les features `positions` et `effects`.

Enfin, sachez qu'un serpent bénéficiant de l'effet `invincibility` à la fin d'un tour sera épargné s'il est censé mourir, et qu'un serpent touché par l'effet `death` à la fin d'un tour sera considéré comme mort pour le reste de la partie.

Vous pouvez également créer vos propres instructions ou votre propre sucre syntaxique pour les stratégies. La seule contrainte est qu'un scénario ne comprenant que les stratégies et effets de base ne soit pas affecté par vos extensions.

D'autres effets, attribués dès le début du jeu, pourraient tout à fait dénaturer ce *Snake* en *TRON* ou autre jeu de votre choix.

Remise du travail

Vous devez rendre votre travail pour le mercredi 6 décembre à 18h00 sur INGInious (https://inginious.info.ucl.ac.be/course/FSAB1402). Votre travail devra prendre la forme d'un unique fichier **NOMA1-NOMA2.zip**, contenant trois fichiers.

1. `Code.oz` contiendra votre code. Veillez à appeler la fonction `SnakeLib.play` une et une seule fois!
2. `Rapport.pdf` contiendra votre rapport de **maximum** 5 pages avec une introduction, un corps décomposé en sections, et une conclusion. Nous souhaitons y retrouver:
   - Vos choix de conception et les difficultés rencontrées. Avez-vous dû faire appel la librairie standard ou utiliser des constructions autres que déclaratives?
   - Une analyse de complexité de vos différentes fonctions.
   - La description de vos extensions. En êtes-vous satisfaits?
3. `Scenario.oz` est un scénario de jeu complètement automatisé qui utilise toutes les fonctionnalités que vous avez implémentées. Comme vous pouvez implémenter virtuellement n'importe quoi, c'est votre seule possibilité de nous montrer vos extensions. Nous vous rappelons enfin qu'il est interdit pour deux groupes de remettre le même scénario.

Pensez à déposer sur INGInious vos solutions temporaires, et à vous organiser pour ne pas dépasser la date limite. Les travaux remis en retard (peu importe la raison) seront pénalisés. À l'inverse, il n'est pas exclu que nous appliquions un bonus aux travaux rendus au moins un jour avant la deadline. Dans le doute, considérez que la deadline est fixée au mardi 5 décembre, c'est plus sûr ;-).

Questions et permanences

Pour des questions d'organisation pratique, nous ne pouvons pas répondre individuellement à toutes vos questions. Un forum est disponible sur Moodle pour y poser vos questions. Nous tâcherons de répondre rapidement aux nouvelles questions. Pensez à lire les questions des autres étudiants, et à vous entre-aider. Il va de soi que nous ne voulons pas voir de morceaux de solution sur le forum. Si vous avez un problème spécifiquement lié à votre code, vous pouvez demander de l'aide à vos tuteurs par email ou à la fin des séances.

Bon travail à tous.