LFSAB1402: Project 2017: BomberSnake Oz

Maxime Istasse, Guillaume Maudoux, Charles Thomas, Hélène Verhaeghe

November 2017

At the beginning of the semester, the teaching assistants noticed that some students were fascinated by the *Snake* game available in *Emacs*, the editor suggested to write code in Oz. Your disappointment knowing it wasn't written in Oz could almost tarnish the mood of the first Lab sessions. We felt it was our duty to fix it.

This is why we wrote for you a *Snake*, in Oz: multi-player, with bombs and bonuses, the trendy *Snake*. Even if it has more potential than the one in *Emacs*, it will need your intervention to work.

Context

The game engine is almost finished, only two functions are left to implement in a declarative way to allow the *snakes* to decode the given strategies and to move according to them as well as the effects affecting them.

Instructions

The project has to be done by groups of two students maximum. All groups have to implement at least the following of instructions, the decoding of strategies and the three mandatory additional effects presented below. All of this, along with the report, will account for at least 14 points on the 20 of the project.

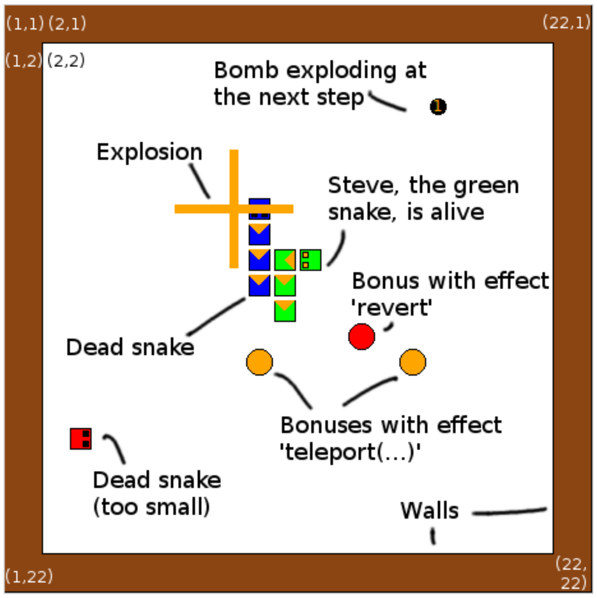
Once you are confident in this basic part, we ask you to extend the game possibilities by adding at least 2 extensions that will determine the remaining of the grade. You can of course implement as much extensions as you want. Just try to focus on the mandatory part of the project first. Lots of extensions will not save an incomplete or badly implemented base.

You will be evaluated on the quality of your code and your report. We expect from you a correct code, understandable (cut into sub-problems, commented) and as efficient as possible (think about terminal recursion). We ask you to use only declarative constructions of the language (no `Cell` nor `Array`) and to avoid the use of the standard library (except `Adjoin` and `AdjoinAt` to modify your snakes in a more flexible way). If you don't manage to respect these conditions, we ask you to specify it and justify it in the report that you have to give with your code and test scenario.

Finally, exchanges of code between students of different groups are not allowed, with the exception of scenarios. Your final submitted test scenario cannot be the same as one from another group. You can also discuss and share ideas.

The basic part

The game takes place on a 22 squares width by 22 squares height grid. Each square is represented by coordinates (x, y) such as $1 \leq x \leq 22$ and $1 \leq y \leq 22$. The coordinate $(1, 1)$ corresponds to the upper left corner (or north-west) of the game zone. Each square of the grid can be occupied by a part of a snake, a bomb, a bonus, a wall, or several of those elements. Here is what the game could look like at a given time:


(1,1)
(2,1)
(22,1)
(1,2)
(2,2)
Bomb exploding at the next step
Explosion
Steve, the green snake, is alive
Bonus with effect 'revert'
Dead snake
Bonuses with effect 'teleport(...)'
Dead snake (too small)
Walls
(1,22)
(22, 22)

A game is decomposed in multiple steps. At each of these steps the following actions are executed:

1. Dropping bombs by the *snakes* that have to do it
2. Decreasing of the timers of the bombs on the grid, the ones reaching 0 explode
3. **Updating the *snakes* based on their effects and the instruction scheduled for this turn** (this is where you intervene)
4. Adding the bonus effect on the snakes and detection of the dead snakes. A snake dies in the following cases:
   - Its head collides with another snake or a wall;
   - It touches the explosion of a bomb;
   - It occupies strictly less than two positions;
   - It goes out of the map;

   At this step, the game engine also checks if a team can be declared winner, in this case the game stops and the winner is proclaimed.

For each game, the *scenario* defines the starting position of the snakes on the grid, the positions of the bonuses and of bombs eventually placed from the beginning. Each *snake* occupies different positions at a given time, each of them having an orientation described by one of the cardinal directions: `north`, `south`, `west` or `east`. The first position specified is the head and the last, the tail.

```
P ::= <integer such as 1 <= x <= 22 >
direction ::= north | south | west | east

snake ::= snake(
        positions: [
            pos(x:<P> y:<P> to:<direction>) % Head
            ...
            pos(x:<P> y:<P> to:<direction>) % Tail
        ]
        effects: nil
)
```

Each *snake* has a predefined strategy that it will have to follow. That is where the problem is situated. First, at each step, the snake will have to follow an instruction: `forward`, `turn(left)` or `turn(right)`. Following this instruction consists of reorienting the head of the snake in the desired direction (forward, to the left or to the right with respect to the current direction) then make it move one square forward in that direction. This is what your `Next` function will have to do in a first time.

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

Then, the game engine expects, as a decoded strategy for each snake, a simple **list of functions**. Each of them takes only one argument, a *snake*, and returns a new *snake* with the new attributes. The n^{th} function of the decoded strategy of each snake will be applied at the step n of the game. You will have to write the `DecodeStrategy` function to decode the strategies that have the following form

```
strategy ::= <instruction> '|' <strategy>
           | repeat(<strategy> times:<integer>) '|' <strategy>
           | nil
```

in a list of functions. Each of them will have to use your `Next` function to modify the snake with the instruction that should be executed at each step. Note that, in addition to the basic instructions, a strategy to decode accepts repetitions of sub-strategies using the syntactic sugar `repeat(...)` presented earlier.

Summary

To begin, you should implement two functions in `Code.oz`:

- `fun {Next Snake Instruction} ... end` which should apply the instruction `Instruction` on the snake `Snake` and return the new snake. The snake returned should have the same size as the one given in parameter (except if an effect specifies otherwise), reoriented its head and moved in the specified direction spe by the instruction.

For example, given the 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)` and `turn(right)`, the result should be `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)`.

Effects will be explained in the next section.

- `fun {DecodeStrategy Strategy} ... end` with `Strategy` specified under the format `strategy` defined earlier. `DecodeStrategy` converts the strategy in a list of functions looking like `fun {$ Snake} {Next Snake Instruction} end`. Each of these functions returns the snake after application of the instruction that should happen at this step of the strategy.

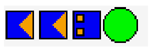
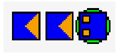
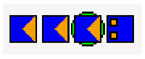
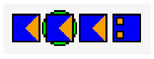
For example, `{DecodeStrategy [repeat([turn(right)] times:2) forward]}` could compute that the simplified strategy `[turn(right) turn(right) forward]` and should return a list of 3 functions `fun {$ Snake} {Next Snake Instruction} end` with the two first applying the instruction `turn(right)` on the snake and the third one applying the `forward` instruction. Each of these functions should do a call to `Next`.

The basic effects

Effects are added to the associated list of the feature `effects` of the snake by the game engine when the head of the snake reaches a bonus. It is your job to apply them and remove them from the list once taken into account by your `Next` function. The game engine will only add items to this list. Effects should be applied in the list order. If multiple effects of the same type are on the list, only the first of each type should be effective, the others) should be ignored and you should remove them.

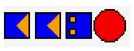
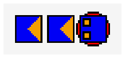
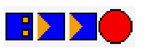
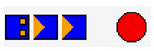
- `grow`: the snake should grow and take one more square at the end of its move. Head will move as usual.

Table 1: Evolution of a snake going forward affected by the `grow` effect at step 2

Step 0	Step 1	Step 2	Step 3
			

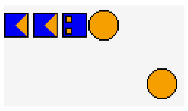
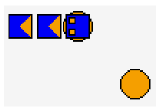
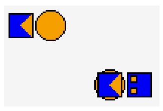
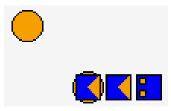
- `revert`: the snake should be inverted before taking care of the move instruction. Its head become its tail and its tail become its head. Directions should be reversed too. in the direction of it.

Table 2: Evolution of a snake going forward, affected by the `revert` effect at step 2

Step 0	Step 1	Step 2	Step 3
			

- `teleport(x:X y:Y)`: the head of the snake (which is on the bonus square) should be teleported at position `(X, Y)` before taking care of the move instruction while keeping the same direction.

Table 3: Evolution of a snake going forward, affected by the `teleport(...)` effect at step 2

Step 0	Step 1	Step 2	Step 3
			

Some tips

Everything described up to now is the basis of the project, and it is suggested to you to proceed in the following way:

- First, handle the basic instructions, in order for your *snakes* to respond correctly to the `'scenario_test_moves.oz'` scenario, which does not contains any bonuses.
- Try to handle the `grow` effect, and test it with the scenario `'scenario_test_grow.oz'`.
- Then, implement the `teleport(...)` and `revert` effects and test them with `'scenario_test_teleport.oz'` and `'scenario_test_revert.oz'` respectively.

In these test scenarios, if everything goes well, *Steve*, the green snake, wins every time. The start situation is such that an implementation error should make it loose. However, surviving these scenarios does not guarantees that the implementation is correct. Do not hesitate to write your own scenarios and test the game with the keyboard to explore more possibilities. (`'scenario_pvp.oz'` is an example of scenario allowing two players to battle each other with the keys ←, →, `q` and `d` on a board with all 3 basic bonuses) You are strongly encouraged to share your scenarios on the Moodle forum (just do not share the one you plan to submit).

Extensions

As specified in the instructions, we ask you to implement 2 other extensions. Your imagination is the only limit. However, we propose you some simple ideas if you do not see what to do, as well as some leads to make things more interesting. We encourage you to aim for originality as this will be taken into account. Note that to test your own extensions, you need to create your own scenario, or to modify an existing one by adding or replacing bonuses. These last have a position, a color for the display, an effect added to the snake(s) affected, and the attribute `target` which specifies which snake is going to be affected after one had its head on the bonus. In most of our scenarios, `catcher` is the value given to `target` for all the bonuses. It makes the bonus applied on the snakes having the same name as the one on the bonus. However, `others`, `allies`, `opponents` and `all` are some other possible values.

```
color ::= <any colors recognised by Tk, see
           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
)
```

Handle a circular grid

It is possible to create a scenario without walls on the side (put attribute `walls` of the scenario to `false`). This means the snakes won't be killed when they arrive at the edge of the window. If the head of one of your snake is supposed to leave the screen, we propose you to make it reappear on the other side of the grid, making him avoiding death (by using the same effect as the teleport bonus).

The `shrink` effect

For now, there is an effect making the snakes grow, but nothing to make them shrink. This is what we want to see with the additional `shrink` effect. The danger is that some snakes can become too small to survive!

The `frozen(N)` effect

This effect will prevent the snake from moving during `N` steps, ignoring the instructions and eventually the possible effects applied.

The `confusion(N)` effect

This effect would invert the `turn(left)` and `turn(right)` during `N` steps. Probably one of the funniest effect to add in the multiplayer mode!

And lots of others!

If you want to exploit fully the possibilities of the game, we invite you to inspire yourself of the code of the existing scenarios to see a bit how this can work. You will notice that, in addition to the attribute discussed earlier, snakes have a team (`team`, any color recognised by Tk) and a predefined strategy to drop the bombs (`bombing`, a list of booleans, one consumed at each step specifying if the *snake* drops a bomb at the place of its tail at this time).

It is totally possible for you to create effects that affect these attributes by using your `Next` method as you would do for the `positions` and `effects` features.

Finally, know that a snake having the `invincibility` effect at the end of a turn would be spared if he was supposed to die, and that a snake affected by the `death` effect at the end of a turn will be considered dead until the end of the game.

You can also create your own instructions or your own syntactic sugar for the strategies. The only constraint is that a scenario that only contains basic strategies and effects won't be affected by your extensions.

Other effects, attributed since the beginning of the game could easily transforme this *Snake* into a *TRON* or any similar game of your choice.

Submission of the project

You have to submit your work for Wednesday 6th December at 18:00 on INGInious (https://inginious.info.ucl.ac.be/course/FSAB1402). Your work has to be submitted as an unique archive file **NOMA1-NOMA2.zip**, containing three files:

1. `Code.oz` will contain your code. Call once and only once the function `SnakeLib.play` in your file!
2. `Rapport.pdf` will contain your report of **maximum** 5 pages with an introduction, a body decomposed using sections and a conclusion. In it we want to find:
    - Your choices of conception and difficulties encountered. Have you made calls to the standard library or use of non-declaratives constructions?
    - An analysis of the complexity of your different functions.
    - The description of your extensions. Are you satisfied with what you have done?
3. `Scenario.oz` will be a game scenario fully automatised using all the functionalities that you have implemented. As you can create virtually everything, this is the only opportunity to show us your extensions. Please remember that it is forbidden for two groups to submit the same scenario.

Think to submit on INGInious your partial solutions, and to organise yourself to not miss the deadline. Late works (whatever the reason) will be penalised. It is also not excluded that we apply bonuses to the work submitted more than one day before the deadline. In doubt, consider that the deadline in the 5 December, it is more secure ;-)

Questions

Do not hesitate to use slack to ask questions. A forum is also available on Moodle. We will try to answer them quickly. Think to read the previously asked questions from the students and help each other. Obviously, we do not want any part of the code shared on slack or Moodle.

Good work everyone!