Advanced Functional and Logic Programming

Mini-project: Programming with Functionals

November 2018

The objective of this mini-project is to check if you master programming with functionals. A functional is a function that takes one or more functions as argument(s), or computes a function as returned value.

1 Description

The goal of this homework is to implement the game 2048. The way of playing this game is described at

`https://en.wikipedia.org/wiki/2048_(video_game)#Gameplay`

In the end, the implemented program

1. should allow a human user to play the game
2. should be able to play alone, based on a simple heuristics

2 Requirements

To solve this problem, you are expected to add the missing implementations of some operations in the source file `game2048.rkt` which can be downloaded from the website of this lecture.

2.1 The table and the moves

The current state of the game is represented by a structure defined by

```
(struct Game (board score) #:transparent)
```

where `(Game-board game)` holds a representation of the tiles of the 4×4 grid of the game, and `(Game-score game)` is an integer that represents the current score of the game. For example, the structure for the game with score 88 and snapshot

is equal to

```
(Game '((4 8 16 2) (0 0 4 4) (0 0 2 8) (0 0 0 2)) 88)
```

Thus, we represent the board by a list of four rows of length four, whose elements store the values of tiles on the grid of the game. 0 represents an empty tile (or cell).

To complete the program, you should implement the following functions from the file `game2048.rkt`:

1. `(zero-replace n v l)`
   replaces the `n`-th zero in list `l` with value `v`. For example,
   - `(zero-replace 1 4 '(4 0 8 0 0 0 5 8))` yields `'(4 4 8 0 0 0 5 8)`
   - `(zero-replace 3 2 '(4 0 8 0 0 0 5 8))` yields `'(4 0 8 0 2 0 5 8)`
2. `(isWon? game)`
   detects if the game is won (that is, there is a tile with value 2048 on the board of the game)
3. `(isLost? game)`
   detects if the game is lost. A game is lost if all of the following conditions hold:
   (a) it is not won,
   (b) there are no more zero tiles, and
   (c) there are no collisions of tiles with same value on horizontal or vertical direction.
4. `(moveRight game)`
   computes the new game obtained by a right shift of all tiles. Note that this operation collapses all rows of game by a right shift, and increments the previous score with the sum of values produced by collisions along all four rows.
5. `(moveDown game)`
   computes the new game obtained by a down shift of all tiles. Note that this operation collapses all columns of game, and increments the previous score with the sum of values produced by collisions along all four columns.

After you implement these methods, you should be able to play this game in interactive mode by running `game-2048.rkt` and evaluating the nullary function

```
> (interactive)
```

A snapshot of two consecutive moves of the game is shown below:

```
Score: 152
|.   |.   |.   |4   |
|.   |4   |.   |.   |
|4   |8   |.   |16  |
|2   |32  |4   |4   |
Next move [w/a/s/d]: d
Score: 160
|.   |.   |.   |4   |
|.   |2   |.   |4   |
|.   |4   |8   |16  |
|.   |2   |32  |8   |
Next move [w/a/s/d]:
```

eof

The keys to be pressed for up/down/left/right shift of the tiles are `w/s/a/d`

2.2 A simple heuristics for playing 2048

The function `(choose-next-game game)` brings the game game to a new state by choosing a move that produces the maximum number of empty cells. The moves that do not modify the game are not taken into account.

This function is used to implement the nullary function `(solitary)` which, when called, allows the user to play with the computer, where the compute uses this simple heuristic to choose the next move.

Review of useful functionals

1. `((compose` f_1 f_2 ... f_n`)` v`) = (`f_1 `(`f_2 ... `(`f_n v`)...))`

   `compose` takes as inputs n unary functions $f_1, f_2, \ldots, f_n$ and returns a function that behaves like the functional composition $f_1 \circ f_2 \circ \ldots \circ f_n$.

2. `(filter` p l`)`

   returns the list of elements e from list l for which `(`p e`)` holds.

3. `(foldl` f v_0 `(list` v_1 v_2 ... v_n`))`
   computes `(`f v_n `(`f v_{n-1} `(... (`f v_1 v_0`) ...)))`

4. `(foldr` f v_0 `(list` v_1 v_2 ... v_n`))`
   computes `(`f v_1 `(`f v_2 `(... (`f v_n v_0`) ...)))`

5. `(map` f l_1 l_2 ... l_n`)`

   where f is a function which takes n arguments, and $l_1, \ldots, l_n$ are lists of the same length. This function call computes ... (see lecture notes)

6. `(apply` f `(list` v_1 ... v_n`))` computes the result of the function call `(`f v_1 ... v_n`)`

Review of some other useful functions on lists

1. `(drop L n)` drops the first `n` elements from list `L`. If `L` has less than `n` elements, it returns the empty list `null`.

2. `(take L n)` returns the list made of the first `n` elements of list `L`. If `L` has less than `n` elements, it returns list `L`.