Labworks

The objective of this labwork 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 here. (Click on the underlined words to follow the hyperlinks)

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 (100p)

To solve this problem, you are expected to add the missing implementations of some operations in the source file `game2048.rkt`. (Click on the underlined words to follow the hyperlinks)

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 reprints an empty tile (or cell).

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

- `(zero-replace n v l)` **(10p)**
  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)`
- `(collapseRow row)` **(10p)**
  takes as input a list of maximum four values (if it is shorter than 4, the missing values are assumed to be 0), and returns the list of five values constructed as follows:
  1. Construct the list of four values for the row obtained by collapsing `row` when the tiles are moved to the left. Missing zeroes (if any) are added to the end of the list.
  2. Append the total value produced by collisions to the front of the list.

  Example:
  - `(collapseRow '(2 2 16)` yields `(4 4 16 0 0)` because the collapse of the row `'(2 2 16 0)` is `'(4 16 0 0)`, and we appended in front of it the value of the newly-produced tile, which is 4.
  - `(collapseRow '(1024))` yields `'(0 1024 0 0 0)`; `(collapseRow '(2 2 2 2))` yields `'(8 4 4 0 0)`; `(collapseRow '(2 8 8 2))` yields `'(16 2 16 2 0)`; `(collapseRow '(2 4 256 2))` yields `'(0 2 4 256 2)` etc.
- `(isWon? game)` **(10p)**
  which detects if the game is won (that is, there is a tile with value 2048 on the board of the game)
- `(isLost? game)` **(10p)**
  which detects if the game is lost. A game is lost if it is not won, there are no more zero tiles, and no possible collisions of tiles with same value on horizontal or vertical direction.

- `(moveRight game)` **(20p)**
  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.

- `(moveDown game)` **(20p)**
  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 executing

```
> (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]:
```

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 (20p)

Implement the function `(choose-next-game game)` which 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.

After implementing thin heuristics, you can call

```
> (solitary)
```

to observe how the program works alone by always choosing a next move that yields the maximum number of empty cells.

NOTE: You can check if two games `game1` and `game2` are identical by evaluating

```
> (equal? game1 game2)
```

Review of useful functionals

- `(compose` f_1 `...`f_n`)` takes as input arguments the unary functions $f_1, \ldots, f_n$ and returns a function that behaves like the functional composition $f_1 \circ \ldots \circ f_n$. Thus the following equality holds:

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

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

- `(apply` f `(list` v_1 `...` v_n`))` returns the value of the function call `(`f v_1 `...` v_n`)`

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

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

- ...