Programming Paradigms Fall 2022
Homework Assignment №2

Innopolis University

October 27, 2022

About this assignment

You are encouraged to use homework as a playground: don't just make some code that solves the problem, but instead try to produce a readable, concise and well-documented solution.

Try to divide a problem until you arrive at simple tasks. Create small functions for every small logical task and combine those functions to build a complex solution. Try not to repeat yourself: for every logical task try to have just one small function and reuse it in multiple places.

The assignment is split into mutliple exercises that have complexity specified in terms of stars ($\star$). The more stars an exercise has the more difficult it is. Exercises with three or more stars ($\star\star\star$) might be really challenging, so please make sure you are done with simple exercises before trying out the more difficult ones.

The assignment provides clear instructions and some examples. However, you are welcome to make the result extra pretty or add more functionality as long as it does not change significantly the original problem and does not make the code *too complex*.

Submit a homework with all solutions as a single file. The file should work by loading it in DrRacket development environment and simply running it.

This homework will be graded out of 100 point:

1. 60 points for the correctness
2. 40 points for the coding style, accounting for
   - idiomatic use of the functional programming paradigm,
   - code structure,
   - correctly used abstractions,
   - error handling,
   - idiomatic naming of variables and functions,
   - helpful and concise comments, and
   - overall documentation, explanation of taken approaches.

This homework also contains an extra credit exercise.

1 Discrete lines and spaces

In this assignment you will need to implement the tools for working with discrete lines and discrete 2D spaces with a point (cell) in focus. You can think of a discrete line as a tape (e.g., like one used in Turing machine).

1.1 Lines

A discrete line consists of a point (cell) in focus and a (possibly infinite) list of cells to the left and to the right of it:

```
-- | A line with a focus.
-- Line xs y zs represents a descrete line:
-- * xs  represents all elements to the left (below)
-- * y   is the element in focus
-- * zs   represents all elements after (above)
data Line a = Line [a] a [a]
  deriving (Show) -- required to enable printing (for finite lines)
```

A simplest example of a line is the integer numbers:

```
-- | A line of integers with focus at 0.
integers :: Line Integer
integers = Line [-1, -2..] 0 [1, 2..]
```

To be able to print lines, we need to be able to select finite portions:

Exercise 1.1 (⋆, 3 points)**.** Implement the following functions:

```
-- | Keep up to a given number of elements in each direction in a line.
-- cutLine 3 integers = Line [-1,-2,-3] 0 [1,2,3]
cutLine :: Int -> Line a -> Line a
```

Exercise 1.2 (⋆, 3 points)**.** Implement the following function:

```
-- | Generate a line by using generating functions.
-- (genLine f x g) generates a line with x in its focus,
-- then it applies f to x until reaching Nothing to produce
-- a list of elements to the left of x,
-- and, similarly, applies g to x until reaching Nothing to
-- produce a list of elements to the right of x.
genLine :: (a -> Maybe a) -> a -> (a -> Maybe a) -> Line a
```

Exercise 1.3 (⋆, 3 points)**.** Implement the following function:

```
-- | Apply a function to all elements on a line.
-- mapLine (^2) integers = Line [1, 4, 9, ..] 0 [1, 4, 9, ..]
mapLine :: (a -> b) -> Line a -> Line b
```

Exercise 1.4 (⋆⋆, 5 points)**.** Implement the following functions:

```
-- | Zip together two lines.
-- zipLines integers integers
--   = Line [(-1,-1),(-2,-2),..] (0,0) [(1,1),(2,2),..]
zipLines :: Line a -> Line b -> Line (a, b)

-- | Zip together two lines with a given combining function.
-- zipLinesWith (*) integers integers
--   = Line [1,4,9,..] 0 [1,4,9,..]
zipLinesWith :: (a -> b -> c) -> Line a -> Line b -> Line c
```

1.2 Rule 30

In this subsection you need to implement Rule 30 cellular automaton[1] using our definition of discrete lines.

When working with a cellular automaton, you need to figure out the next state for each cell simultaneously. However, start by considering only the cell in focus:

[1]see https://en.wikipedia.org/wiki/Rule_30

Exercise 1.5 (⋆, 3 points)**.** Implement the following function, computing the next state of the cell in focus, according to Rule 30:

```haskell
data Cell = Alive | Dead
  deriving (Show)

rule30 :: Line Cell -> Cell
```

Now, we need tools to allow us to focus on all cells.

Exercise 1.6 (⋆, 3 points)**.** Implement the following functions, shifting the focus on the line by one position (if possible):

```haskell
shiftLeft  :: Line a -> Maybe (Line a)
shiftRight :: Line a -> Maybe (Line a)
```

Now that we can shift by one position, we can shift repeatedly, to reach each cell on the line. In fact, we can get an entire line of different focus shifts:

Exercise 1.7 (⋆⋆, 5 points)**.** Implement the following function, which maps every cell in a line into a version of the original line where that cell is in focus. The new line of lines should have the original line in focus.

```haskell
lineShifts  :: Line a -> Line (Line a)
```

With `lineShifts`, we can now simply apply `rule30` to each shifted version of the line to get the new state for each cell:

```haskell
applyRule30 :: Line Cell -> Line Cell
applyRule30 line = mapLine rule30 (lineShifts line)
```

Now let's add some visualization:

Exercise 1.8 (⋆⋆, 5 points)**.** Implement the following functions:

```haskell
-- | Render a line of 1x1 pictures.
renderLine :: Line Picture -> Picture

-- | Render the fist N steps of Rule 30,
-- applied to a given starting line.
renderRule30 :: Int -> Line Cell -> Picture
```

1.3 Discrete spaces

A discrete 2D space can be represented by a (vertical) line of (horizontal) lines:

```haskell
-- | A descrete 2D space with a focus.
-- A 2D space is merely a (vertical) line
-- where each element is a (horizontal) line.
data Space a = Space (Line (Line a))
```

Exercise 1.9 (⋆⋆⋆, +0.5% extra credit)**.** Implement the following function, computing the cartesian product of two lines to produce a 2D space:

```haskell
productOfLines :: Line a -> Line b -> Space (a, b)
```

Exercise 1.10 (⋆⋆, 6 points)**.** Implement the following utility functions:

```haskell
mapSpace :: (a -> b) -> Space a -> Space b
zipSpaces :: Space a -> Space b -> Space (a, b)
zipSpacesWith :: (a -> b -> c) -> Space a -> Space b -> Space c
```

Exercise 1.11 (⋆⋆, 6 points)**.** Implement the following utility functions:

```haskell
mapSpace :: (a -> b) -> Space a -> Space b
zipSpaces :: Space a -> Space b -> Space (a, b)
zipSpacesWith :: (a -> b -> c) -> Space a -> Space b -> Space c
```

1.4 Conway's Game of Life

Here you need to implement Conway's Game of Life by using our definition of discrete space.

Start by implementing the main rule of the automaton:

Exercise 1.12 (⋆, 3 points)**.** Implement the following function, computing the next state of the cell in focus, according to the rules of Conway's Game of Life:

```
data Cell = Alive | Dead
  deriving (Show)

conwayRule :: Space Cell -> Cell
```

Since we already can shift focus in lines, we can also shift focus in space. In fact, we can get an entire space of different focus shifts:

Exercise 1.13 (⋆⋆⋆, 9 points)**.** Implement the following function that converts each cell in a discrete space into a version of the original space with focus shifted to that cell. The new space (of spaces) must have the original space in focus.

```
spaceShifts :: Space a -> Space (Space a)
```

With `spaceShifts`, we can now simply apply `conwayRule` to each shifted version of the space to get new state for every cell:

```
applyConwayRule :: Space Cell -> Space Cell
applyConwayRule space = mapSpace conwayRule (spaceShifts space)
```

Now let's add some visualization:

Exercise 1.14 (⋆⋆, 6 points)**.** Implement the following functions:

```
-- | Render a space of 1x1 pictures.
renderSpace :: Space Picture -> Picture

-- | Animate Conway's Game of Life,
-- starting with a given space
-- and updating it every second.
animateConway :: Space Cell -> IO ()
```