Declaratieve Talen

Haskell 4

1 Fractals and Turtle Graphics

In this exercise we focus on extending existing code rather than building new code from scratch. A template file with the required code is provided.
Note on testing: While e-systant will check your answers for you as usual, for debugging purposes it can be useful to render your images. Since e-systant does not support rendering svg's directly, here are some methods to check the output of your program.

- If you are working on e-systant, you can render an svg as a copy-pastable string by going to the interactive environment (the green play button), using the `linesToSVG` function to obtain a string and then using `putStr` on the result. It's important to not use `print` or `show` here! These functions insert escape characters which break the svg format. You can copy-paste the svg string into an online HTML viewer like https://codepen.io/pen/.
- If you are working on your own system, you can write your svg strings directly to files and view them with your system's image viewer or web browser. On Ubuntu, the image viewer also lets you use the cursor keys to navigate between images, which is particularly useful for the exercise on animations (you can output a series of numbered files).

Exercises.

- Some svg viewers might not render your image properly if you're working too far outside of their default view box. We have partially implemented functionality to output an accurate view box along with the image, but this implementation depends on the currently unimplemented `boundingBox :: [Line] -> (Point,Point)` function. Given a list of lines, the first point in its output is the point with the lowest x and y coordinate (top-left in svg), and the second point is the one with the highest x and y coordinate (bottom-right in svg), out of all the points in the list. You may assume that the turtle draws at least one line.

  Example:

```
> boundingBox square
((500.0,500.0),(550.0,550.0))
```

- Debugging fractals is difficult if we can't easily inspect how they are constructed step-by-step. It is useful to be able to render our fractals frame-by-frame: we output a `svg` file with the "current state" of the image for each step the turtle makes. Implement the function `animate :: Turtle -> [Turtle]`. The output is a list of turtles, where the first turtle in the list only draws the very first line of the drawing, the second turtle draws the first and the second line, and so on. Make sure not to create frames that don't draw any lines!

  Example:

  ```
  > concretize 10 triangle
  Step 10.0 (Turn (-120.0) (Step 10.0 (Turn (-120.0) (Step 10.0 Done))))
  > animate $ concretize 10 triangle
  [ Step 10.0 Done
  , Step 10.0 (Turn (-120.0) (Step 10.0 Done))
  , Step 10.0 (Turn (-120.0) (Step 10.0 (Turn (-120.0) (Step 10.0 Done))))
  ]
  ```

- Add support for drawing discontinuously. Do this by adding two commands to the `Turtle` instruction set: a `PenUp` command and a `PenDown` command. When a `PenUp` command is encountered, subsequent `Step` commands will still move the current position of the turtle, but not actually draw anything, until a `PenDown` command is encountered, at which point `Step` commands behave normally again. Lifting the pen when it is already lifted has no effect (the pen remains lifted), and putting the pen down when it is already put down also has no effect (the pen remains put down). You will need to edit several functions to support this feature.

  – Extend the `(>>>)` operator to support the new operators.

  – Extend the `turtleToLines` function to support lifting the pen and putting it down. You will need an additional `Bool` field in the state to indicate whether the pen is currently down or not.

    ```
    > turtleToLines square
    [((500.0,500.0),(500.0,550.0)),((500.0,550.0),(550.0,550.0)),
     ((550.0,550.0),(550.0,500.0)),((550.0,500.0),(500.0,500.0))]
    > turtleToLines xi
    [((500.0,500.0),(550.0,500.0)),((540.0,485.0),(510.0,485.0)),
     ((500.0,470.0),(550.0,470.0))]
    ```

  – Finally also extend the `animate` function. It should not emit frames for movements of the pen that do not draw anything.

```
> length (animate xi)
3
> length (animate square)
4
```

- Implement the function `dashedStep :: Int -> Dist -> Turtle`. `dashedStep n dist` produces a turtle that traverses a distance `dist`, placing `n` lines (dashes) in its path. The lines are each of the same length, and the empty stretches between the lines are also of the same length. So each line (dash) has length $\texttt{dist}/(2\texttt{n} - 1)$.

  ```
  > dashedStep 3 10
  PenDown (Step 2.0 (PenUp (Step 2.0 (PenDown
    (Step 2.0 (PenUp (Step 2.0 (PenDown (Step 2.0 Done))))))))
  ```

- Provide an implementation for the function `dash :: Int -> Turtle -> Turtle`. `dash n t` replaces every every line that is produced by running `t`, by a dashed line with `n` dashes (as produced by the `dashedStep` function). Watch out: you only want to replace lines by dashed lines if those lines would have been drawn in the first place (in other words, keep track of when the pen is lifted to avoid placing dashed lines where there were no lines to begin with)!

  ```
  > dash 2 xi
  Turn 90.0 (PenDown (Step 16.666666666666668 (PenUp
  (Step 16.666666666666668 (PenDown (Step 16.666666666666668
  (Turn 90.0 (PenUp (Step 15.0 (Turn 90.0 (Step 10.0 (PenDown
  (PenDown (Step 10.0 (PenUp (Step 10.0 (PenDown (Step 10.0 (PenUp
  (Step 10.0 (Turn (-90.0) (Step 15.0 (Turn (-90.0) (PenDown (PenDown
  (Step 16.666666666666668 (PenUp (Step 16.666666666666668
  (PenDown (Step 16.666666666666668 Done)))))))))))))))))))))))))))
  ```

- Currently our fractal refining abilities are a bit hampered by the fact that we only have one type of step available, which cannot carry any data. This makes it impossible to use our system for rendering certain interesting fractals such as the dragon curve (`https://en.wikipedia.org/wiki/Dragon_curve`). In order to render a classic dragon curve, we need to be able to alternate between right turns and left turns. This can be done by having steps which are tagged with either "right" or "left", which refine differently. A "right"-tagged step refines into

  1. Turn right (negative angle) 45 degrees,
  2. step forward, tagged "left"
  3. turn left 90 degrees
  4. step forward, tagged "right"
  5. turn right 45 degrees

A “left”-tagged step refines into:

1. Turn left (positive angle) 45 degrees,
2. step forward, tagged “left”
3. turn right 90 degrees
4. step forward, tagged “right”
5. turn left 45 degrees

Because of how our algorithm is set up now though, it is not possible to distinguish between different kinds of steps. **Extend the `FStep` constructor** to carry this information with it in an additional field. The type of this field should be a new datatype `Dir` (already provided for you in the template), which has one constructor for “right” and one constructor for “left”. To avoid breaking the tests, don’t change the type signature of the `fstep` function, but introduce a new function `fstep' :: Dir -> Fractal`.

Implement the dragon curve as a function `dragon :: Dir -> Fractal`. Write the function `compileFractal' :: Fractal -> (Dir -> Fractal) -> Int -> Double -> Turtle` and which operates analogously to its variant without prime, but the refinement step is parameterized on the direction.

```
> compileFractal' (dragon L) dragon 0 10.0
Turn 45.0 (Step 10.0 (Turn (-90.0) (Step 10.0 (Turn 45.0 Done))))

compileFractal' (dragon L) dragon 2 10.0
Turn 45.0 (Turn 45.0 (Turn 45.0 (Step 10.0 (Turn (-90.0) (Step 10.0
(Turn 45.0 (Turn (-90.0) (Turn (-45.0) (Step 10.0 (Turn 90.0 (Step 10.0
(Turn (-45.0) (Turn 45.0 (Turn (-90.0) (Turn (-45.0) (Turn 45.0 (Step 10.0
(Turn (-90.0) (Step 10.0 (Turn 45.0 (Turn 90.0 (Turn (-45.0) (Step 10.0
(Turn 90.0 (Step 10.0 (Turn (-45.0) (Turn (-45.0) (Turn 45.0 Done
))))))))))))))))))))))))))))

> compileFractal' (dragon L) dragon 2 10.0
Turn 45.0 (Turn 45.0 (Turn 45.0 (Step 10.0 (Turn (-90.0) (Step 10.0 (Turn
45.0 (Turn (-90.0) (Turn (-45.0) (Step 10.0 (Turn 90.0 (Step 10.0
(Turn (-45.0) (Turn 45.0 (Turn (-90.0) (Turn (-45.0) (Turn 45.0 (Step
10.0 (Turn (-90.0) (Step 10.0 (Turn 45.0 (Turn 90.0 (Turn (-45.0) (Step 10.0
(Turn 90.0 (Step 10.0 (Turn (-45.0) (Turn (-45.0) (Turn 45.0 Done
))))))))))))))))))))))))))))
```