Haskell

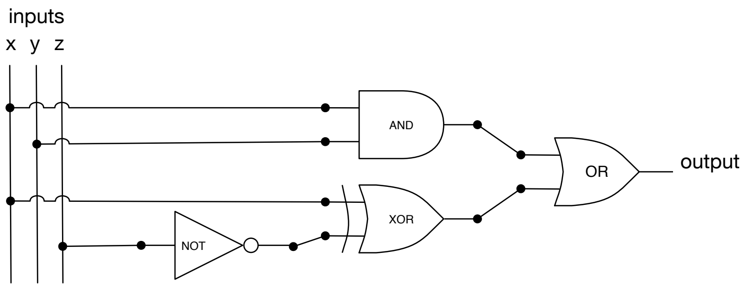
Part 1: Logical Circuits

This assignment concerns *networks of logical gates* or *circuits* for short. A circuit is a network of logical gates (AND, OR, NOT, . . .) that implement boolean logic.

This assignment considers only logical gates with one or more inputs and exactly one output. In general a circuit can consist of multiple gates whose inputs and outputs are arbitrarily connected with one another. In this assignment we only consider circuits that correspond to boolean formulas. For example, a particular circuit can be presented by the formula:

$$(x \wedge y) \vee (\neg z \oplus x)$$

Every variable presents an input of the circuit and the logical connectives represent logical gates ($\wedge$ for AND, $\vee$ for OR, $\neg$ for NOT, and $\oplus$ for XOR). The result of the formula corresponds to the output of the circuit. In short, the above formula corresponds to the circuit below:



Task 1a. Define a new datatype `Circuit` that represents a circuit (or a logical formula). A circuit is one of:

1. one of the inputs – every input is characterised by its name, a `String`;
2. a logical NOT gate applied to an underlying circuit;
3. a logical AND gate applied to two underlying circuits;
4. a logical OR gate applied to two underlying circuits; or
5. a logical XOR gate applied to two underlying circuits.

Task 1b. Use the `Circuit` datatype to define a few trivial circuits:

1. `cinput :: String -> Circuit` is the circuit that returns the input with the given name.
2. `cnot :: Circuit -> Circuit` is the circuit that returns the logical NOT of the given circuit.
3. `cand :: Circuit -> Circuit -> Circuit` is the circuit that returns the logical AND of the two given circuits.

4. `cor :: Circuit -> Circuit -> Circuit` is the circuit that returns the logical OR of the two given circuits.
5. `cxor :: Circuit -> Circuit -> Circuit` is the circuit that returns the logical XOR of the two given circuits.

Task 1c. Use the functions from Task 1b to define the above example circuit as `example :: Circuit`.

Task 1d. Write the function `candMany :: [Circuit] -> Circuit` that returns the logical AND of the given list of circuits. You may assume that the list is non-empty.

Part 2: Fun with Circuits

Task 2a. Write an instance of the `Show` type class for `Circuit` that turns a circuit into a `String` that represents the corresponding formula in a textual form. The expected textual form can best be explained with an example:

```
> example
OR(AND(x,y),XOR(NOT(z),x))
```

Note that there are no spaces in the text.

Task 2b. Our circuits contain 4 different kinds of gates (NOT, AND, OR, XOR). The last two are actually not essential. Indeed, we can express XOR in terms of the first three, and OR in terms of the first two kinds of gate. Namely:

$$\begin{aligned} x \oplus y &= (x \wedge \neg y) \vee (\neg x \wedge y) \\ x \vee y &= \neg(\neg x \wedge \neg y) \end{aligned}$$

Write the function `simplify :: Circuit -> Circuit` which eliminates all OR and XOR gates from the given circuit and returns an equivalent circuit that contains only NOT and AND gates. Make use of the above two equations.

```
> simplify example
NOT(AND(NOT(AND(x,y)),NOT(NOT(AND(NOT(AND(NOT(z),NOT(x))),NOT(AND(NOT(NOT(z)),x)))))))
```

Task 2c. Write a function `size :: Circuit -> Int` which returns the number of gates in a given circuit. The inputs are not gates and do not count.

```
> size example
4
> size (cinput "x")
0
> size (simplify example)
15
```

Task 2d. The *gate delay* of a circuit is the maximum number of gates on a path from an input of the circuit to its output. For the example the gate delay is 3 because there are 3 gates (NOT, XOR and OR) on the path between the input z and the output of the circuit. Write the function `gateDelay :: Circuit -> Int` which returns the gate delay of the given circuit.

```
> gateDelay example
3
> gateDelay (cinput "x")
0
> gateDelay (simplify example)
9
```

Task 2e. Write the function `inputs :: Circuit -> [String]` which returns the list of input names of the given circuit. The names can be in any order, but the list should not contain any duplicates.

```
> inputs (cinput "x")
["x"]
> inputs example
["x","y","z"]
```

Part 3: Simulate Circuits

Task 3a. Write the function `simulate :: Circuit -> [(String,Bool)] -> Bool` that simulates which output the given circuit returns if the inputs take the given values.

```
> simulate (cnot (cinput "x")) [("x",True)]
False
> simulate (cnot (cinput "x")) [("x",False)]
True
> simulate example [("x",True),("y",False),("z",True)]
True
```

Task 3b. Write the function `combinations :: Int -> [[Bool]]` such that `combinations` n generates the list of all possible combinations of n boolean values. **Note:** the combinations have to be ordered in a particular way. You can derive the required order from the examples below:

```
> combinations 0
[[]]
> combinations 1
[[False],[True]]
> combinations 2
[[False,False],[False,True],[True,False],[True,True]]
> combinations 3
[[False,False,False],[False,False,True],[False,True,False],[False,True,True]
,[True,False,False],[True,False,True],[True,True,False],[True,True,True]]
```

Task 3c. Write the function `tabulate :: Circuit -> IO ()` which prints the given circuit in the form of a table as illustrated below. Make use of the functions `inputs`, `simulate` and `combinations`. For the sake of simplicity you may assume that the names of inputs consist of a single character.

```
> tabulate (cinput "x")
x | output
0 | 0
1 | 1
> tabulate (cnot (cinput "x"))
x | output
0 | 1
```

```
1 | 0
> tabulate example
x y z | output
0 0 0 | 1
0 0 1 | 0
0 1 0 | 1
0 1 1 | 0
1 0 0 | 0
1 0 1 | 1
1 1 0 | 1
1 1 1 | 1
```

1 Fractals and Turtle Graphics

In this exercise we focus on extending existing code rather than building new code from scratch. We return to the fractals and turtle graphics example from class. A template file with the code (slightly modified) from class 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 make it impossible to copy-paste the output. You can then 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 supports the cursor keys which is particularly useful for the exercise on animations (you can output a series of numbered files).

- Some svg viewers might not render your image properly if you're working too far outside of their default view box. We edited our svg rendering function such that an accurate view box is also generated, but this new implementation requires a working implementation of the `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. To support drawing d you should **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`, which has one constructor for “right” and one constructor for “left”. For testing purposes, 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 type.

```
> 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
))))))))))))))))))))))))))))
```