# Haskell

## Part 1: Turtle Graphics

Turtle graphics was a popular way for introducing programming that was conceived as part of the original Logo programming language developed by Wally Feurzig and Seymour Papert in 1966.

In the first part of the assignment, we are recreating a simple form of turtle graphics programs. Such a program consists of a sequence of instructions for a turtle. A turtle has a particular position and orientation on the X-Y plane (e.g., starting at the origin $(0,0)$ facing towards $(0,1)$). An instruction consists either of changing the orientation of the turtle, turning counterclockwise ($\circlearrowleft$) by a number of degrees, or of moving forward some distance.

When the turtle executes a program of instructions, it leaves a trace of the path it follows on the screen. Hence, clever programs lead to intricate shapes or pretty pictures.

**Task 1a.** Create a new datatype `Turtle` that denotes a turtle graphics program. A turtle graphics program is one of:

1. a trivially empty program that denotes there is nothing more to be done;
2. a counterclockwise turn ($\circlearrowleft$) by a given angle $\alpha$ (with $\alpha \in [-360°, 360°]$ of type `Double`) followed by the remainder of the program;
3. a step forward by a given distance $d$ (of type `Double`) followed by the remainder of the program.

**Task 1b.** Use the `Turtle` datatype to create several trivial programs:

1. `done :: Turtle` is the trivial empty program;
2. `turn :: Double -> Turtle` is the program that consists of a single turn.
3. `step :: Double -> Turtle` is the program that consists only of a step forward.

**Task 1c.** Define the operator `(>>>) :: Turtle -> Turtle -> Turtle` that concatenates two turtle programs. This means that the instructions of the second program are to be executed after those of the first program.

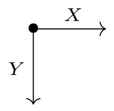
**Task 1d.** Put all of the above together to write a small turtle program `square :: Turtle` that draws a square with sides of size 50.

## Part 2: Visualizing Turtle Graphics

To visualize a turtle graphics program, we proceed in two steps. Firstly, we convert it into a list of line segments. Secondly, we turn the line segments into an `svg`[1] image that can be viewed by an appropriate image viewer program (e.g., most web browsers).

[1]scalable vector graphics

Note that `svg` displays the coordinate system upside down, e.g., $(0, 1)$ is below $(0, 0)$. This is not a reason for concern; you just need to be aware that images are depicted upside down from what you'd expect.

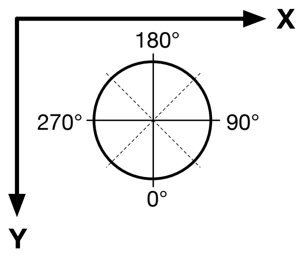


We introduce basic vocabulary for the coordinate system with the type synonyms `Point` and `Line`, defined as follows:

```
type Point = (Double,Double)
type Line  = (Point,Point)
```

The former denotes a point on the X-Y plane and is represented by a tuple of its x and y coordinates in `Double` precision. The latter denotes a line segment on the X-Y plane and is represented by its a tuple of start and end `Point`s.

**Task 2a.** Define the function `turtleToLines :: Turtle -> [Line]` that turns a turtle graphics program into a list of line segments. To perform this conversion you need to know that the turtle starts out at the point $(500, 500)$ and its heading is $0°$ where the following picture explains the meaning of the heading:



If you are at position $(x, y)$ in the plane heading towards $d$ degrees and take a step of length $l$, you end up in the new position $(x', y')$ where:

$$\begin{cases} x' &= x + l \sin d \frac{2\pi}{360} \\ y' &= y + l \cos d \frac{2\pi}{360} \end{cases}$$

In Haskell, you can make use of the predefined functions `sin, cos :: Double -> Double` and `pi :: Double` to implement the above formulas.

Here is an example:

```
> 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))]
```

**Task 2b.** Write the function `linesToSVG :: [Line] -> String` that returns a string with the svg representation of the line segments. The string consists of a number of XML-like tags:

- a header tag of the form: `<svg xmlns="http://www.w3.org/2000/svg" version="1.1">`
- any number of line tags of the form

```
<line x1="..." y1="..." x2="..." y2="..." stroke="blue" stroke-width="4" />
```

- a closing footer tag of the form: `</svg>`

For example, the following denotes an `svg` image of a square.

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
<line x1="500.0" y1="500.0" x2="550.0" y2="500.0" stroke="blue" stroke-width="4" />
<line x1="550.0" y1="500.0" x2="550.0" y2="550.0" stroke="blue" stroke-width="4" />
<line x1="550.0" y1="550.0" x2="500.0" y2="550.0" stroke="blue" stroke-width="4" />
<line x1="500.0" y1="550.0" x2="500.0" y2="500.0" stroke="blue" stroke-width="4" />
</svg>
```

Note that in order to include quote character `"` in a Haskell string literal, you have to *escape* it by preceding it with a backslash character. For instance, the string literal that contains the quoted text `hello` is written `"\"hello\""`.

**Task 2c.** Write the function `writeSVG :: FilePath -> Turtle -> IO ()` that takes a file name and a turtle program and creates a file of the given name that contains an `svg` image of the given turtle program.

Make use of the predefined function `writeFile :: FilePath -> String -> IO ()` that creates a file with the given name and content. Note that `FilePath` is just a type synonym for `String`.

## Part 3: Turtle Fractals

In the last part of the assignments, we show how to use turtle graphics for drawing fractals.

**Task 3a.** Define a new datatype `Fractal` for programs that are in all aspects similar to turtle programs *except* that no distance is specified for the steps in the program. (We say that the steps in `Fractal` programs are *abstract*.)

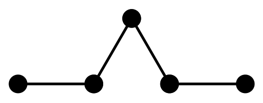
**Task 3b.** Define the functions `fdone :: Fractal`, `fturn :: Double -> Fractal`, `fstep :: Fractal` and `(>->) :: Fractal -> Fractal -> Fractal` that are the counterparts of the corresponding `Turtle` functions.

**Task 3c.** Write a function `concretize :: Double -> Fractal -> Turtle` that turns a given `Fractal` program into a `Turtle` program by replacing all abstract steps by concrete steps of the same given distance.

**Task 3d.** Write a function `refine :: Fractal -> Fractal -> Fractal` where `refine expansion program` expands (i.e., replaces) every distance-less step in the given `program` into the given expansion, which itself can be a multi-instruction program.

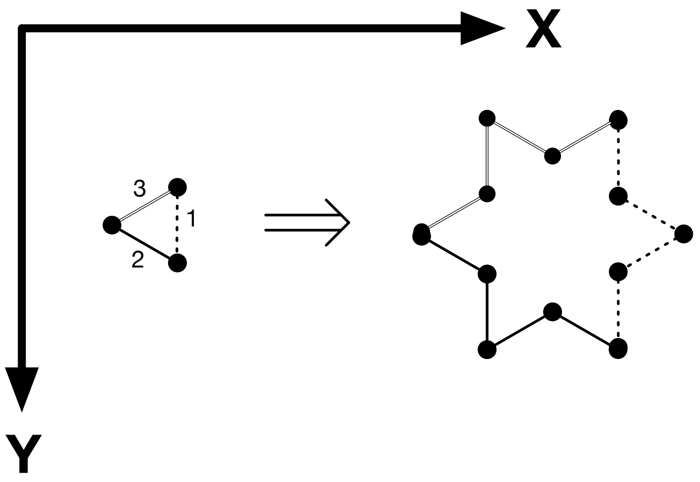
For example, if we expand every step into the following four steps with the rotations $60°$, $-120°$ and $60°$ between them, i.e.,

```
expansion =
  fstep >-> fturn (60)   >->
  fstep >-> fturn (-120) >->
  fstep >-> fturn (60)   >->
  fstep
```



the image on the left turns into the image on the right:

```
program =
  fstep >-> fturn (-120) >->
  fstep >-> fturn (-120) >->
  fstep
```



In other words, for the example we have:

```
refine expansion program   ==   expansion >-> fturn (-120) >->
                                expansion >-> fturn (-120) >->
                                expansion
```

**Task 3e.** Write a function `times :: Int -> (a -> a) -> (a -> a)` where (`times` $n$ $f$ $x$) returns $f(f...(fx))$ where $f$ is stacked $n$ times.

**Task 3f.** Wrapping up: write a function `exam :: Fractal -> Fractal -> Int -> Double -> FilePath -> IO ()` where `exam program expansion n d filename` refines the given `program` `n` times with the given `expansion`, concretizes it into a turtle program with the given step size `d` and then writes it out as an `svg` file with the given `filename`.

For example, starting from the triangle above and expanding twice with the expansion above yields: