Haskell

Deel 1: Turtle Graphics

Turtle graphics was een populaire aanpak om programmeren aan te leren die deel uitmaakte van de Logo programmeertaal die ontwikkeld werd door Wally Feurzig en Seymour Papert in 1966.

In dit eerste deel van de opgave reconstrueren we een eenvoudige vorm van turtle graphics programma's. Een dergelijk programma bestaat uit een sequentie van instructies voor een schildpad. Een schildpad heeft een bepaalde positie en oriëntatie in het X-Y vlak (bv., ze start in de oorsprong $(0, 0)$ en kijkt naar $(0, 1)$). Een instructie bestaat ofwel uit het veranderen van de oriëntatie van de schildpad door een aantal graden tegen de klok in te draaien ($\circlearrowleft$), ofwel door een bepaalde afstand voorwaarts te stappen.

Taak 1a. Maak een nieuwe datatype `Turtle` that een turtle graphics programma voorstelt. Een turtle graphics programma is één van:

1. een triviaal leeg programma dat voorstelt dat er niets meer te gebeuren valt;
2. een draai tegen de klok in ($\circlearrowleft$) van een gegeven hoek α (met $\alpha \in [-360°, 360°]$ van type `Double`) gevolgd door de rest van het programma;
3. een stap voorwaarts van een gegeven afstand (van type `Double`) gevolg door de rest van het programma.

Taak 1b. Gebruik het `Turtle` datatype om enkele triviale programma's te schrijven:

1. `done :: Turtle` is het triviale lege program;
2. `turn :: Double -> Turtle` is het programma dat bestaat uit één enkele draai.
3. `step :: Double -> Turtle` is het programma dat bestaat uit één enkele stap voorwaarts.

Taak 1c. Definieer de operator `(>>>) :: Turtle -> Turtle -> Turtle` die twee programma's achter elkaar plakt. Dit betekent dat de instructies van het tweede programma uitgevoerd worden na die van het eerste programma.

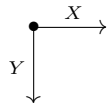
Taak 1d. Combineer alle bovenstaande definities om het kleine programma `square :: Turtle` te schrijven dat een vierkant met zijde 50 tekent.

Deel 2: Turtle Graphics Visualiseren

We visualiseren een turtle graphics programma in twee stappen. Eerst zetten we het om in een lijst van lijnsegmenten. Vervolgens zetten we deze lijnsegmenten om in een `svg`[1] afbeelding die bekeken kan worden met een geschikte viewer (zoals de meeste web-browsers).

[1] scalable vector graphics

Merk op dat `svg` het coördinaatsysteem ondersteboven weergeeft, bv. $(0, 1)$ bevindt zich onder $(0, 0)$. Dit is geen reden tot bezorgdheid; je moet gewoon weten dat afbeeldingen ondersteboven worden weergegeven.

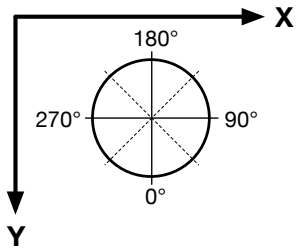


We introduceren wat basiswoordenschat voor het coördinaatsysteem met de typesynoniemen `Point` en `Line`, gedefinieerd als volgt:

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

Het eerste stelt een punt voor in het X-Y vlak en wordt voorgesteld door het tupel met de x en y coördinaten in `Double` precisie. Het tweede stelt een lijnsegment voor in het X-Y vlak en wordt voorgesteld door het tupel met begin-en eindpunt.

Taak 2a. Definieer de functie `turtleToLines :: Turtle -> [Line]` die een turtle graphics programma omzet naar een lijst van lijnsegmenten. Om deze omzetting uit te voeren moet je weten dat de schildpad start op positie $(500, 500)$ en georiënteerd is naar $0°$; de volgende afbeelding legt uit wat deze hoek betekent:



Als je je op positie (x, y) bevindt in het vlak en georiënteerd bent naar hoek d graden, en je neemt een stap met lengte l, dan kom je uit op positie (x', y') zodat:

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

In Haskell kan je gebruik maken van de voorgedefinieerde functies `sin, cos :: Double -> Double` en `pi :: Double` om bovenstaande formules te implementeren.

Hier is een voorbeeld.

```
> 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))]
```

Taak 2b. Schrijf de functie `linesToSVG :: [Line] -> String` die een string teruggeeft met de `svg` voorstelling van de lijnsegmenten. De string bestaat uit een aantal XML-achtige tags:

- een hoofding van de vorm: `<svg xmlns="http://www.w3.org/2000/svg" version="1.1">`
- een willekeurig aantal `line` tags van de vorm:

  `<line x1="..." y1="..." x2="..." y2="..." stroke="blue" stroke-width="4" />`
- een afsluitende tag van de vorm: `</svg>`

Bijvoorbeeld, het volgende stelt een `svg` afbeelding voor van een vierkant.

```
<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>
```

Merk op dat als je een aanhalingsteken `"` wil opnemen in een Haskell string literal, je het moet *escapen* door het vooraf te gaan met een backslash karakter. Bijvoorbeeld, de string literal die de aangehaalde tekst `hello` bevat wordt geschreven als `"\"hello\""`.

Taak 2c. Schrijf de functie `writeSVG :: FilePath -> Turtle -> IO ()` die een bestand aanmaakt met de gegeven naam dat een `svg` afbeelding bevat van het gegeven turtle programma.

Maak gebruik van de voorgedefinieerde functie `writeFile :: FilePath -> String -> IO ()` die een bestand aanmaakt met de gegeven naam en inhoud. Merk op dat `FilePath` een typesynoniem is van `String`.

Deel 3: Turtle Fractals

In het laatste deel van de opgave tonen we hoe we turtle graphics kunnen gebruiken om fractals te tekenen.

Taak 3a. Definieer een nieuw datatype `Fractal` voor programma's die in alle aspecten overeenkomen met turtle programma's *behalve* dat er geen afstand opgegeven is voor de stappen in het programma. (We noemen de stappen in `Fractal` programma's *abstract*.)

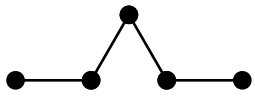
Taak 3b. Definieer de functies textttfdone :: Fractal, `fturn :: Double -> Fractal`, `fstep :: Fractal` en `(>->) :: Fractal -> Fractal -> Fractal` die de `Fractal` tegenhangers zijn van de overeenkomstige `Turtle` functies.

Taak 3c. Schrijf de functie `concretize :: Double -> Fractal -> Turtle` that turns a given `Fractal` program into a `Turtle` program programma door alle abstracte stappen om te zetten naar concrete stappen met allen dezelfde gegeven afstand.

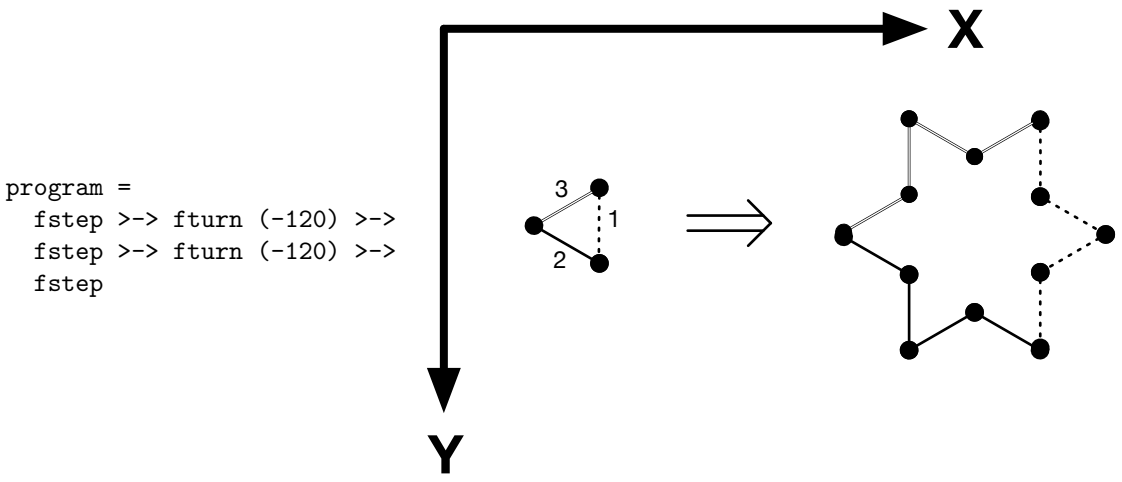
Taak 3d. Schrijf de functie `refine :: Fractal -> Fractal -> Fractal` zodat `refine expansion program` elke abstracte afstandsloze stap in het gegeven `program`ma expandeert (dwz. vervangt) tot de gegeven `expansion` die bestaat uit een programma met potentieel meerdere instructies.

Bijvoorbeeld, als we elke stap expanderen tot de volgende vier stappen met de rotaties $60°$, $-120°$ and $60°$ ertussen, dus

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



dan wordt de afbeelding links omgezet naar de afbeelding rechts:



Met andere woorden, voor het voorbeeld geldt:

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

Taak 3e. Schrijf de functie `times :: Int -> (a -> a) -> (a -> a)` waar `(times` n f x`)` het resultaat $f(f...(fx))$ geeft waar de functie f n keer wordt toegepast.

Taak 3f. Samenvattend: schrijf de functie `exam :: Fractal -> Fractal -> Int -> Double -> FilePath -> IO ()` waar `exam program expansion n d filename` het gegeven `program` `n` eerst keer na elkaar expandeert met de gegeven `expansion`, dan het resultaat omzet naar een `Turtle` programma met gegeven stapgrootte `d` en tot slot uitschrijft als `svg` bestand met de gegeven `filename`.

Bijvoorbeeld, als we de bovenstaande driehoek twee keer expanderen krijgen we: