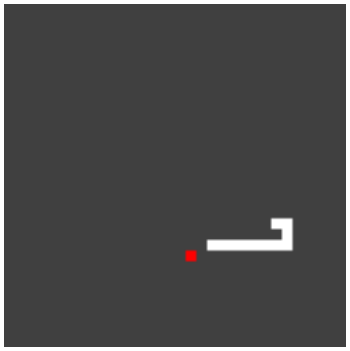
1 Haskell: Spieleprogrammierung mit GIF-Streams

Snake ist ein Computerspiel, bei dem man eine Schlange duch ein Spielfeld steuert. Futter zu essen verlängert die Schlange. Kollidiert die Schlange mit einer Wand oder sich selbst, so endet das Spiel.



In dieser Aufgabe implementieren Sie Snake in Haskell. Das dazu benötigte Rahmenwerk finden Sie auf der Übungshomepage.

Die Ausgabe des Spiels erfolgt über einen animierten GIF-Stream, den man im Browser anschauen kann. Es werden 64 Farben unterstützt, die als Int-Tupel von (0,0,0) bis (3,3,3) angesprochen werden.

```
type RGB = (Int,Int,Int)
```

Ein einzelner Frame einer GIF ist definiert als Liste von Zeilen, wobei jede Zeile eine Liste von RGB-Werten ist.

```
type Frame = [[RGB]]
```

Das Rahmenwerk stellt eine Funktion `server` zur Verfügung, die einen HTTP-Server unter dem angegebenen Port startet. Der Server schickt jedem Client in einem festgelegten Intervall einen neuen Frame der GIF-Animation. In der übergebenen Logic-Funktion werden dynamisch neue Frames generiert.

```
server :: PortNumber -> Int -> Logic -> IO ()
```

Die Datei `Snake.hs` enthält das Grundgerüst für das zu schreibende Snake-Spiel. Kompilieren Sie das Spiel und führen Sie es aus:

```
$ ghc -O3 -threaded Snake.hs
[1 of 2] Compiling GifStream        ( GifStream.hs, GifStream.o )
[2 of 2] Compiling Main             ( Snake.hs, Snake.o )
Linking Snake ...
$ ./Snake
Listening on http://127.0.0.1:5002/
```

Öffnen Sie die angegebene Adresse in einem Browser. Durch Drücken der Tasten WASD im Terminal lässt sich die GIF im Browser beeinflussen.

Anderen Teilnehmern Ihres Netzwerks ist es ebenfalls möglich den GIF-Stream zu betrachten, indem Sie statt `127.0.0.1` Ihre Netzwerk-IP-Adresse eintragen.

Desweiteren ist es möglich den GIF-Stream aufzunehmen um ihn später anzuschauen:

```
wget -O game.gif http://127.0.0.1:5002/
```

Die wichtigste Funktion in `Snake.hs` ist `logic`:

```
logic wait getInput sendFrame = initialState >>= go
  where
    go (State oldAction snake food) = do
      input <- getInput

      -- Generate new state
      let action = charToAction input oldAction
      let newSnake = snake
      let newFood = food

      let frame = case action of
            MoveUp    -> replicate height (replicate width (3,0,0))
            MoveDown  -> replicate height (replicate width (0,3,0))
            MoveLeft  -> replicate height (replicate width (0,0,3))
            MoveRight -> replicate height (replicate width (3,3,3))

      sendFrame (scale zoom frame)

      wait
      go (State action newSnake newFood)
```

Die Funktion `logic` erzeugt einen initialen Zustand für das Snake-Spiel und übergibt diesen an die `go`-Funktion. Diese liest mit `getInput` die zuletzt gedrückte Taste. Anschließend wird ein neuer Spielzustand generiert. Der anzuzeigende Frame wird dabei abhängig von der gedrückten Taste gewählt. Schließlich wird mit `sendFrame` ein neuer Frame an alle verbundenen Clients geschickt. Dabei wird jeder Frame durch `scale` vergrößert. Der Aufruf von `wait` bewirkt ein Warten für die vereinbarte Zeit `delay`, die standardmäßig auf $100ms$ gesetzt ist. Am Ende der Funktion ruft diese sich selbst endrekursiv mit dem neu generierten Zustand auf.

Ziel dieser Aufgabe ist es die Spiellogik in der Funktion `logic` schrittweise zu erweitern, so dass man am Ende Snake spielen kann.

1.1 Spielfeld ausgeben

Erzeugen Sie aus dem aktuellen Zustand ein Bild und geben Sie dieses statt der einfachen farbigen Bilder aus.

Schreiben Sie dazu eine Liste `fieldPositions`, die die Koordinaten des Spielfelds an ihrer jeweiligen Position speichert.

```
fieldPositions :: [[Position]]
```

Die Größe des Feldes ist dabei in `width` und `height` gespeichert. Für ein Feld der Größe 3x4 würde fieldPositions wie folgt aussehen:

```
fieldPositions = [[(0,0),(1,0),(2,0)]
                 ,[(0,1),(1,1),(2,1)]
                 ,[(0,2),(1,2),(2,2)]
                 ,[(0,3),(1,3),(2,3)]]
```

Implementieren Sie eine Funktion `colorize`, die eine einzelne Bildposition auf eine Farbe abbildet, so dass sich der neue Frame durch `let frame = map (map (colorize newSnake newFood)) fieldPositions` erzeugen lässt. Ein Feld soll verschieden eingefärbt werden, je nachdem ob diese Position Teil der Schlange, Futterstück oder Hintergrund ist.

```
colorize :: [Position] -> Position -> Position -> RGB
```

1.2 Verhalten Schlange

Implementieren Sie nun die Zustandsänderung der Schlange, so dass Sie in der Spiellogik `let newSnake = moveSnake snake food action` schreiben können.

```
moveSnake :: [Position] -> Position -> Action -> Position
```

Eine Schlange ist als Liste von Positionen definiert. Die neue Schlange erhält abhängig von der übergebenen Aktion einen neuen Kopf. `Action` ist wie folgt definiert:

```
data Action = MoveLeft | MoveRight | MoveUp | MoveDown deriving Eq
```

Beim Schwanz wird das letzte Element abgeschnitten, außer wenn die Schlange gerade auf Futter gestoßen ist.

Es muss sichergestellt werden, dass die vom Benutzer gewählte Aktion überhaupt möglich ist. Schreiben Sie dazu eine Funktion `validateAction`, so dass Sie in der Spiellogik `let action = validateAction oldAction (charToAction input oldAction)` schreiben können. Dazu soll `validateAction` nur dann eine neue Aktion zurückgeben, wenn diese möglich ist. Ansonsten soll die alte Aktion zurückgegeben werden.

1.3 Verhalten Futter

Implementieren Sie nun die Zustandsänderung des Futters, so dass Sie in der Spiellogik `newFood <- moveFood newSnake food` schreiben können.

```
moveFood :: [Position] -> Position -> IO Position
```

Wenn die Schlange gerade das Futter nicht isst, kann direkt die alte Position des Futters zurückgegeben werden. Ansonsten soll die neue Position des Futters zufällig gewählt werden. Vermeiden Sie dass das Futter im Körper der Schlange erscheint.

Zufallszahlen zwischen x und y (einschließlich) lassen sich bei Verwendung der do-Syntax mit `r <- randomRIO (x,y)` generieren. Importieren Sie dazu `System.Random`.

1.4 Spielende

Passen Sie das Ende von `logic` so an, dass mit `checkGameOver newSnake` die Gültigkeit des neuen Zustands überprüft wird. Bei einem ungültigen Zustand soll das Spiel durch Aufruf von `initialState >>= go` neugestartet werden.

```
checkGameOver :: [Position] -> Bool
```

1.5 Kür

Programmieren Sie ein weiteres Spiel mit GIF-Stream-Ausgabe, zum Beispiel Pong, Tetris oder Conway's Game of Life.