Navigating the Maze

Ken Friis Larsen (`kflarsen@diku.dk`)

Last revision: September 20, 2012

Objective

The objective of this assignment is to gain hands-on programming experience with Haskell and to get an understanding of how to implement and program with monads.

The goal is to implement an interpreter for MEL (Maze Exploring Language) for controlling a robot moving around in a grid based maze.

The assignment consists of two mandatory parts, and an optional third part:

- Part 1: Implement a module `World` for representing mazes and the state of the robot.
- Part 2: Implement an monadic interpreter for MEL that will move a robot around in maze.
- Part 3: Suggestions for various extensions. This part is **optional**, it will have no influence whatsoever on your grading.

What to hand in

You should hand in two things:

1. Your code.
2. A short report explaining the code, and an assessment of the quality of code including what this assessment is based on.

Scope

The following topics are not within the scope (of Part 1 and 2) of this assignment:

- Maze generation
- Efficient representation of mazes

Part 1: Modelling the world

In this part of the assignment you declare a Haskell module for modelling mazes and robots.

A maze is a grid of cells with walls between some of the cells. We say that a maze $M \times N$ has M rows each with N columns. We use pairs of integers for denoting a specific cell. We number the cells starting from the lower left:

(0,2)	(1,2)	(2,2)
(0,1)	(1,1)	(2,1)
(0,0)	(1,0)	(2,0)

We use the types `Direction` and `Position` in the following:

```
data Direction = North | South | West | East
               deriving (Show,Eq,Read,Enum)
type Position = (Int, Int)
```

and we say that cell at position (0,1) is to the north of (0,0), and likewise for the other directions.

Declare a Haskell module `World` (in the file with name `World.hs`) with the following content:

- The above type declaration for `Direction`.
- A type `Cell` for modelling a cell. That is, a cell is the list walls for that cell. For instance, if a cell has walls to the north and east it should be modelled by the list `[North, East]`.
- Functions for manipulating positions.
- A type `Maze` for modelling a maze. A maze should have a mapping from positions to cells, and the height (number of rows) and the width (number of columns) of the maze. Describe what invariants should hold for your data structure. This type should be abstract and not exported from the module
- Functions for querying the maze in various ways. For instance, you might want a function `validMove m p1 p2`:

  ```
  validMove :: Maze -> Position -> Position -> Bool
  ```

  That checks that it's OK to move from position `p1` to position `p2`. That is, there is no wall between `p1` and `p2`.

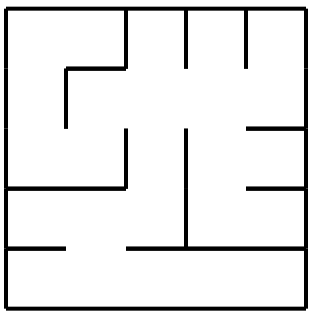
- A function `fromList` for constructing mazes:

  ```
  fromList :: [(Position, [Direction])] -> Maze
  ```

For instance, the value `testMaze`:

```
testMase = fromList [((0,0),[North,South,West]),((0,1),
[North,South,West])
                    ,((0,2),[South,West]),((0,3),[West,East])
                    ,((0,4),[North,West]),((1,0),[South]),((1,1),
[North])
                    ,((1,2),[South,East]),((1,3),[North,West])
                    ,((1,4),[North,South,East]),((2,0),
[North,South])
                    ,((2,1),[South,East]),((2,2),[West,East]),
((2,3),[])
                    ,((2,4),[North,West,East]),((3,0),
[North,South])
                    ,((3,1),[South,West]),((3,2),[West])
                    ,((3,3),[]),((3,4),[North,West,East])
                    ,((4,0),[North,South,East]),((4,1),
[North,South,East])
                    ,((4,2),[North,South,East]),((4,3),
[South,East])
                    ,((4,4),[North,West,East])]
```

Corresponds to the maze:



Part 2: A MEL interpreter

We use the following types for modelling abstract syntax trees for MEL programs:

```
data Relative = Ahead | ToLeft | ToRight | Behind
              deriving (Eq, Show)

data Cond = Wall Relative
          | And  Cond Cond
          | Not  Cond
          | AtGoalPos
          deriving (Eq, Show)

data Stm = Forward
         | Backward
         | TurnRight
         | TurnLeft
```

```
          | If Cond Stm Stm
          | While Cond Stm
          | Block [Stm]
          deriving (Eq, Show)

type Program = Stm
```

A MEL program is just a statement (`Stm`) which could be a block consisting of a list of statements. The semantics of each statement should be straightforward. The condition `AtGoalPos` is true if the robot is in the top right corner, otherwise it is false.

- A type `Robot` for modelling the state of a robot in a maze. That is, which position in maze the robot is currently at and which direction the robot is facing. Furthermore, the robot should also keep a history of the path it has travelled so far (a list of positions).
- A type `World` for modelling a maze and a robot.
- A function `initialWorld` with type

  ```
  initialWorld :: Maze -> World
  ```

  That takes a maze an create an initial world from a maze, where the robot is at position (0,0) and facing north.
- Complete the type `RobotCommand` and make it a monad instance:

  ```
  newtype RobotCommand a = RC { runRC :: World -> ... }
  ```

  Note that the maze does not change during execution, your type should reflect this. Your type should also specify how you deal with errors and your should document that.
- Define a function `interp`, that computes effect of executing a statement, with the type:

  ```
  interp :: Stm -> RobotCommand ()
  ```
- Define a function `runProg`

  ```
  runProg :: Maze -> Program -> Result ([Postion], Direction)
  ```

  `runProg` runs a program started from the initial world derived from a given maze. The return value is the path traversed by the robot, and the final direction of the robot. The `Result` type above is not specified because it depends on how you decide to handle errors.

  For example, `runProg testMaze TurnRight` should return `([(0,0)],West)` (perhaps with a suitable constructor applied as well).

- Suggestion for testing: write a robot program that will find a path from the initial position to the goal position in all (loop free) mazes.

Part 3: Extensions

- Extend MEL so that the robot can inspect its own history.

  Is this an interesting extension? Why?

- Extend MEL with a `Fork` statement that allows a robot to fork itself into two robots: one facing the original directing and one turned right. Robots can only be forked four times (resulting in a maximum of 16 robots in one world). You may also want to extend MEL with an extra condition to allow robots to detect which of the forked robots it is.