Exercise Set 0, Haskell

Ken Friis Larsen, Maya Saietz

September, 2019

Touring Haskell

- Finish the declaration of the function `move`
- `type Pos = (Int, Int)`
- `data Direction = North | South | East | West`
-
- `move :: Direction -> Pos -> Pos`

-
  ```
  move North (x,y) = (x, y+1)
  move West  (x,y) = (x-1, y)
  ```

- Declare a function `moves` with the type

  ```
  moves :: [Direction] -> Pos -> Pos
  ```

  The function should return the net result of performing all the moves in sequence.

- Declare functions for adding and multiplying natural numbers, using the following data type declaration:

-
  ```
  data Nat = Zero | Succ Nat
      deriving (Eq, Show, Read, Ord)
  ```

  `Succ` stands for "successor," and the idea is that `Zero` represents the natural number 0, `Succ Zero` represents the natural number 1, `Succ (Succ Zero)` represents the natural number 2, and so on.

  Your functions should work on the above unary representation *directly*, not by converting to and from native Haskell integers.

- Declare functions `nat2int` and `int2nat` to go from natural numbers (the above data type) to Haskell integers and back.
- Given the data type for representing binary search trees with integers stored in the nodes:
- `-- All leaves in left/right subtree are less/greater than node value`

-
  ```
  data Tree = Leaf | Node Int Tree Tree
      deriving (Eq, Show, Read, Ord)
  ```

  Declare a function `insert` that takes an integer `n` and a binary search tree `t`, and returns a new search tree `t'` with `n` inserted correctly into `t`. If `n` was already present in `t`, the function should just return `t` as `t'`. Insert new elements at the leaves only; do not worry about keeping the tree balanced.

Morse Code

[Rephrased from ruby quiz #121]

Morse code is a way to encode telegraphic messages in a series of long and short sounds or visual signals. During transmission, pauses are used to group letters and words, but in written form the code can be ambiguous.

For example, using the typical dot (`.`) and dash (`-`) for a written representation of the code, the word `...---..-....-` in Morse code could be an encoding of the names Sofia or Eugenia depending on where you break up the letters:

```
...|---|..-.|..|.-    Sofia
.|..-|--.|.|-.|..|.-  Eugenia
```

We will only focus on the alphabet for this quiz to keep things simple. Here are the encodings for each letter:

A	.-	N	-.
B	-...	O	---
C	-.-.	P	.--.
D	-..	Q	--.-
E	.	R	.-.
F	..-.	S	...
G	--.	T	-
H	....	U	..-
I	..	V	...-
J	.---	W	.--
K	-.-	X	-..-
L	.-..	Y	-.--
M	--	Z	--..

- Declare a function, `encode`, that takes a string (of letters) as argument and translate it to Morse code (as a string).
- Declare a function, `decode`, that takes a string encoded in morse code as argument and return a list of all possible translations.

Tic-Tac-Toe

These exercises assume that you have downloaded the file `TicTacToe.hs`, which you can find on Absalon under *Files*.

- Define the undefined values `startState`, `makeMove`, `validMove`, `allValidMoves`, and `makeTree`.

  Define one of them at a time, and make sure that your Haskell file still compiles after each change.

- Make sure that you understand the `allNodes` function, especially what the functions `concatMap`, `snd`, and `.` (dot) does. Try to write each of these functions yourself.
- Observe the difference in running time of

  ```
  length (allNodes (makeTree startState))    -- should return 986410
  ```

  and

  ```
  take 3 (allNodes (makeTree startState))
  ```

  Can you explain that?

- Watch Haskell Live episode 1, and make `showBoard` and `readBoard` functions for tic-tac-toe boards in a similar style as Rein Henrichs (the Haskell Live creator) does for chess boards.
- Implement the minimax algorithm for finding an optimal move in a given game state.

Type classes

Following is an interface for types that can be said to have a size.

```
class Sizeable t where
  size :: t -> Int
```

We say that a primitive type like `Int` have size one:

```
instance Sizeable Int where
  size _ = 1
```

When we want to ascribe a size to aggregate types like lists, we need to take a decision: Should the size of a list be just the length of the list, or is the size of a list the sum of the size of the elements plus the length of the list (plus one). Complete the following two instance declarations (one for each interpretation of what sizeable means)

```
instance Sizeable [a] where
  ...

instance Sizeable a => Sizeable [a] where
  ...
```

Note: You cannot have both instance declarations active at the same time.

Make pairs, `String`, and `Tree` sizeable.