Programming assignment 2

Hardy Jones

999397426

Professor Köppe

This file is a literate haskell file. So it could be run directly if that were required.

We only have to make slight modifications to our previous program in order to handle `EUC_2D`and `GEO` files.

Again, we first require some pragmas to let us know if something slipped by.

```
> {-# OPTIONS_GHC -Wall #-}
> {-# OPTIONS_GHC -Werror #-}
> {-# OPTIONS_GHC -fno-warn-unused-do-bind #-}
```

And also for lens generation:

```
> {-# LANGUAGE FlexibleInstances #-}
> {-# LANGUAGE GADTs #-}
> {-# LANGUAGE TemplateHaskell #-}
> {-# LANGUAGE TypeSynonymInstances #-}
```

Now on to our actual program:

```
> module P2 where
```

We import a whole mess of helper things

```
> import Control.Lens -- ((^.), makeLenses, to)
```

import Control.Applicative – (Const(..))

```
> import Data.Function (on)
> import Data.Int (Int32)
> import Data.Tuple (swap)
> import Data.Typeable (Typeable)
> import System.FilePath ((</>))
> import Text.Parsec ( ParseError, anyChar, endOfLine, manyTill
>                    , spaces, string, try
>                      )
> import Text.Parsec.String (Parser, parseFromFile)
> import Text.ParserCombinators.Parsec.Number ( fractional, nat
>                                             , sign
>                                             )
> import Text.Printf (PrintfType, printf)
```

And off we go.

The first thing we want is to model the data somehow. It'd be much more ideal to use `Data.Geo.Coordinate`, however, it uses a slightly different algorithm for computing the distance, and the difference in roundoff error is too great.

That said, we just model it directly as given:

```
> data Node = Node
>     { _number :: Int
>     , _r1     :: Double
>     , _r2     :: Double
>     } deriving (Eq, Ord, Show, Typeable)
```

And of course generate some lenses so we don't end up doing too much work ourselves.

```
> makeLenses ''Node
```

Next, we want to be able to talk about the different types of input formats. At the moment we have to handle `EUC_2D` and `GEO`. Since there's no actual difference between the format of the data, we can use void types to represent the different formats. The reason being, we don't need to have values of these types, just tag `Node`s with them.

```
> data EUC_2D
> data GEO
```

We want to be able to take a value and create a type.

```
> data NodeType a where
>     GeoNode' :: Node -> NodeType GEO
>     EucNode :: Node -> NodeType EUC_2D
> makeFields ''NodeType
```

Then we can use `Const` to create tagged type synonyms:

```
> type GeoNode = Const Node GEO
> type EucNode = Const Node EUC_2D
```

For each tagged type, we need a different distance function. Since this is the only thing that changes in the whole algorithm, we abstract over it with a typeclass.

```
> class Distance d where
>     distance :: Integral a => d -> d -> a
```

Then we can provide an implementation for each `Node` synonym.

The algorithm given in the TSPLIB pdf. This is basically a direct copy. Any other implementation ends up with slightly different values.

```
> instance Distance GeoNode where
>     distance = go `on` getConst
>         where
>         go i j = floor (rrr * acos (((1 + q1) * q2 - (1 - q1) * q3) / 2) + 1)
>             where
>             rrr, q1, q2, q3, latI, latJ, longI, longJ, pi' :: Double
>             q1 = cos $ longI - longJ
>             q2 = cos $ latI - latJ
>             q3 = cos $ latI + latJ
>             latI = radians i latDegrees latMinutes
>             latJ = radians j latDegrees latMinutes
>             longI = radians i longDegrees longMinutes
>             longJ = radians j longDegrees longMinutes
>             radians node d m = pi' * (node^.d + 5 * node^.m / 3) / 180
>             latDegrees = r1.to (fromInteger . round)
>             latMinutes = r1.to ((-) <*> fromInteger . round)
>             longDegrees = r2.to (fromInteger . round)
>             longMinutes = r2.to ((-) <*> fromInteger . round)
>             pi' = 3.141592
>             rrr = 6378.388

> instance Distance (NodeType GEO) where
>     distance = go `on` run
>         where
>         run :: NodeType GEO -> Node
>         run (GeoNode' n) = n
>         go i j = floor (rrr * acos (((1 + q1) * q2 - (1 - q1) * q3) / 2) + 1)
>             where
>             rrr, q1, q2, q3, latI, latJ, longI, longJ, pi' :: Double
>             q1 = cos $ longI - longJ
>             q2 = cos $ latI - latJ
>             q3 = cos $ latI + latJ
>             latI = radians i latDegrees latMinutes
>             latJ = radians j latDegrees latMinutes
>             longI = radians i longDegrees longMinutes
>             longJ = radians j longDegrees longMinutes
>             radians node d m = pi' * (node^.d + 5 * node^.m / 3) / 180
>             latDegrees = r1.to (fromInteger . round)
>             latMinutes = r1.to ((-) <*> fromInteger . round)
>             longDegrees = r2.to (fromInteger . round)
>             longMinutes = r2.to ((-) <*> fromInteger . round)
>             pi' = 3.141592
>             rrr = 6378.388
```

Euclidean distance on nodes.

```
> instance Distance EucNode where
>     distance = go `on` getConst
>         where
>         go i j = round $ sqrt $ dx ** 2 + dy ** 2
>             where
>             dx = i^.r1 - j^.r1
>             dy = i^.r2 - j^.r2
```

Now we need some way to parse the file. Since we don't have to concern ourselves with the entire format at this time, we do the most simplistic approach possible.

First we ignore everything up until `NODE_COORD_SECTION`, then parse a whole bunch of nodes until `EOF`.

```
> parseNodes :: Parser [Const Node a]
> parseNodes = do
>     manyTill anyChar   $ try $ string "NODE_COORD_SECTION"
>     manyTill parseNode $ try $ string "EOF"
```

The format for the nodes is: `<integer> <real> <real>`.

We create a parser for this schema.

```
> parseNode :: Parser (Const Node a)
> parseNode = do
>     n <- Node <$> (spaces *> nat)
>          <*> (spaces *> (sign <*> fractional))
>          <*> (spaces *> (sign <*> fractional))
>          <*  endOfLine
>     pure $ Const n
```

We construct a tour by pairing up each node with its immediate successor, and wrapping the end of the list around.

```
> tour :: [a] -> [(a, a)]
> tour = zip <*> (uncurry (++) . swap . splitAt 1)
```

```
> tourDistance :: Distance a => [(a, a)] -> Int32
> tourDistance = sum . fmap (uncurry distance)
```

We provide some output formatting

```
> format :: PrintfType a => Int32 -> a
> format = printf
>     "The distance of the tour in order 1-2-...-22-1 is: %d km\n"
```

Finally, we actually attempt to run this program. First parse the file. If it doesn't parse, then print out the message and be done. If it parses fine, then create a tour, find the distance, and print it.

```
> main :: IO ()
> main = do
>     nodes <- parseFromFile parseNodes ("all_tsp" </> "ulysses22.tsp")
>     either print (format . tourDistance . tour) (nodes :: Either ParseError [GeoNode])
```

We can see this in action by running on the terminal: