```haskell
module LSystems ( LSystem(LSystem), ColouredLine, Command(..)
                , angle, axiom, rules, lookupChar
                , expandOne, expand, move, trace1, trace2
                , expandLSystem, commandMap ) where

import IC.Colour

type Rules a = [(Char, [a])]
data LSystem = LSystem Float [Char] (Rules Char)
type Vertex = (Float, Float)
type TurtleState = (Vertex, Float)
data Command = F | L | R | B [Command]
type ColouredLine = (Vertex, Vertex, Colour)

-----------------------------------------------------------
-- Functions for working with systems.

-- Returns the rotation angle for the given system.
angle :: LSystem -> Float
angle (LSystem a _ _) = a

-- Returns the axiom string for the given system.
axiom :: LSystem -> [Char]
axiom (LSystem _ ax _) = ax

-- Returns the set of rules for the given system.
rules :: LSystem -> Rules Char
rules (LSystem _ _ rs) = rs

--
-- Pre: the character has a binding in the Rules list
--
lookupChar :: Rules a -> Char -> [a]
lookupChar rules c = head [s | (ch, s) <- rules, ch == c]

--
-- Expand command string s once using rule table r
--
expandOne :: Rules Char -> [Char] -> [Char]
expandOne = concatMap . lookupChar

--
-- Expand command string s n times using rule table r
--
expand :: [Char] -> Int -> Rules Char -> [Char]
expand s n r = iterate (expandOne r) s  !!  n

-- Move a turtle.
--
-- F moves distance 1 in the current direction.
-- L rotates left according to the given angle.
-- R rotates right according to the given angle.
move :: Command -> Float -> TurtleState -> TurtleState
move L turnAngle (position, angle) = (position, angle + turnAngle)
move R turnAngle (position, angle) = (position, angle - turnAngle)
move F turnAngle ((x, y), angle)   =
  ((x + cos (degreesToRadians angle), y + sin (degreesToRadians angle)), angle)

-- Converts from degrees to radians.
degreesToRadians :: Float -> Float
degreesToRadians x = (x / 180) * pi

parse :: Rules Command -> [Char] -> [Command]
parse rs = fst . go where
  go :: [Char] -> ([Command], [Char])
  go [] = ([], [])
  go ('[':cs) = let (cmds, cs') = go cs
                    (cmds', cs'') = go cs'
                in (B cmds : cmds', cs'')
  go (']':cs) = ([], cs)
  go (c:cs) = let (cmds, cs') = go cs
              in (lookupChar rs c ++ cmds, cs')

trace1 :: [Command] -> Float -> Colour -> [ColouredLine]
trace1 cs turnAngle colour = go cs ((0,0), 90)
  where
    go :: [Command] -> TurtleState -> [ColouredLine]
    go [] _ = []
    go (B cs : cs') state = go cs state ++ go cs' state
    go (F : cs) state@(p, _) = (p, p', colour) : go cs state'
      where state'@(p', _) = move F turnAngle state
    go (c : cs) state = go cs (move c turnAngle state)

-- This version uses an explicit stack of residual commands and turtle states
type Stack = []
trace2 :: [Command] -> Float -> Colour -> [ColouredLine]
trace2 cs turnAngle colour = go cs ((0, 0), 90) [] where
  go :: [Command] -> TurtleState -> Stack ([Command], TurtleState) -> [ColouredLine]
  go [] _ [] = []
  go [] _ ((cs, state):stack) = go cs state stack
  go (F : cs) state@(p, _) stack = (p, p', colour) : go cs state' stack
    where state'@(p', _) = move F turnAngle state
  go (B cs : cs') state stack = go cs state ((cs', state) : stack)
  go (c : cs) state stack     = go cs (move c turnAngle state) stack

-- Provided Functions
------------------------------------------------------------------------------

expandLSystem :: LSystem -> Int -> [Command]
expandLSystem (LSystem _ axiom rs) n = parse commandMap (expand axiom n rs)

commandMap :: Rules Command
commandMap = [ ('M', [F])
             , ('N', [F])
             , ('X', [])
             , ('Y', [])
             , ('A', [])
             , ('+', [L])
             , ('-', [R])
             ]
```