

CI4251 - Programación Funcional Avanzada

Primera Entrega Proyecto Final

Héctor Guilarte	Armando Mejía
05-38294	05-38524
<hectorg87@gmail.com>	<armemo215@gmail.com>

Junio 18, 2010

1. Primera entrega de HRubik

1.1. En esta entrega:

- Representación del cubo de Rubik utilizando tipos abstractos de datos.
- Definición e implementación de las operaciones sobre el cubo de Rubik.
- Utilización del monad StateT para la secuenciación de instrucciones aplicadas sobre el cubo e impresión del mismo tras cada movimiento.
- Generación aleatoria de Instrucciones con la utilización de QuickCheck para probar el correcto funcionamiento de las operaciones.

1.2. Pendiente para la siguiente entrega:

- Dibujar el Cubo de Rubik por pantalla con la utilización de HOpenGL.
- Hacer pruebas exhaustivas con QuickCheck.
- Verificar que el cubo fue resuelto, i.e. el cubo está ordenado.
- Realizar instancia personalizada de Eq del tipo de datos @Cube@ para realizar pruebas con QuickCheck.
- Realizar parser con Parsec para traducir la entrada del lenguaje embebido por el usuario a las instrucciones definidas por el tipo de datos @Inst@.

```
{-/  
  /Primera entrega de HRubik/  
  
    Programación Funcional Avanzada CI3725  
  
    05-38294 Héctor Guilarte <mailto:hectorg87@gmail.com>  
    05-38524 Armando Mejía <mailto:armemo215@gmail.com>  
  
  En esta entrega:  
  - Representación del cubo de Rubik utilizando tipos abstractos  
    de datos.  
  - Definición e implementación de las operaciones sobre el cubo  
    de Rubik.  
  - Utilización del monad StateT para la secuenciación de  
    instrucciones aplicadas sobre el cubo e impresión del mismo  
    tras cada movimiento.
```

```

- Generacion aleatoria de Instrucciones con la utilización de
  QuickCheck para probar el correcto funcionamiento de las
  operaciones.

Pendiente:
- Dibujar el Cubo de Rubik por pantalla con la utilizacion de
  HOpenGL.
- Hacer pruebas exhaustivas con QuickCheck.
- Verificar que el cubo fue resuelto, i.e. el cubo está ordenado.
- Realizar instancia personalizada de Eq del tipo de datos
  @Cube@ para realizar pruebas con QuickCheck.
- Realizar parser con Parsec para traducir la entrada del
  lenguaje embebido por el usuario a las instrucciones definidas
  por el tipo de datos @Inst@.

-}
module Main where

import Data.Sequence
import Data.Word (Word8)
import Control.Monad.State
import Test.QuickCheck
import Test.QuickCheck.Gen
import System.Random

-- Corners letters:      Edge letters:
--          *---i---*
--      f-----e      /l      /l
--      /l      /l      e l      f j
--      a-----b l      *---a---* l
--      l g---/-h      l *---k/---*
--      l/      l/      d /      b /
--      d-----c      /h      /g
--          *---c---*

data CubeState = CS { cube :: !Cube,
                     insts :: [Inst]
                     }

data Cube = Cube { _corners :: !Corners,
                  _edges :: !Edges,
                  _centers :: !Centers
                  }
    deriving(Eq, Show)

type Corners = Seq Corner

type Edges = Seq Edge

```

```

type Centers = Seq Center

data Corner = Corner { z :: !Face, -- ^ Z axis
                      y :: !Face, -- ^ Y axis
                      x :: !Face  -- ^ X axis
                    }

    deriving(Eq)

instance Show Corner where
    show (Corner z y x) = show "Z: " ++ show z ++ show ", Y: "
                        ++ show y ++ show ", X: " ++ show x ++ "\n"

data Edge = Edge { fe :: !Face, -- ^ Front Edge
                  xy :: !Face  -- ^ X or Y axis
                }

    deriving(Eq)

instance Show Edge where
    show (Edge fe xy) = show "fe: " ++ show fe ++ show ", XY: "
                    ++ show xy ++ "\n"

data Center = Center { center :: !Face -- ^ Center tile
                    }

    deriving(Eq)

instance Show Center where
    show (Center center) = show "center: " ++ show center ++ "\n"

type Face = Word8

type Color = String

{-| @Inst@ Conjunto de instrucciones para manejar
    el cubo de Rubik.
-}
data Inst = F    -- ^ Rotate Front Clockwise
          | F'   -- ^ Rotate Front CounterClockwise
          | R    -- ^ Rotate Right Clockwise
          | R'   -- ^ Rotate Right CounterClockwise
          | L    -- ^ Rotate Left Clockwise
          | L'   -- ^ Rotate Left CounterClockwise
          | B    -- ^ Rotate Back Clockwise
          | B'   -- ^ Rotate Back CounterClockwise
          | U    -- ^ Rotate Up Clockwise
          | U'   -- ^ Rotate Up CounterClockwise
          | D    -- ^ Rotate Down Clockwise
          | D'   -- ^ Rotate Down CounterClockwise
          | MF   -- ^ Rotate Middle Front Clockwise
          | MF'  -- ^ Rotate Middle Front CounterClockwise

```

```

| MU  -- ^ Rotate Middle Up Clockwise
| MU' -- ^ Rotate Middle Up CounterClockwise
| MR  -- ^ Rotate Middle Right Clockwise
| MR' -- ^ Rotate Middle Right CounterClockwise
| TF  -- ^ Turn Cube ClockWise on Z axis
| TF' -- ^ Turn Cube CounterClockwise on Z axis
| TU  -- ^ Turn Cube ClockWise on Y axis
| TU' -- ^ Turn Cube CounterClockwise on Y axis
| TR  -- ^ Turn Cube ClockWise on X axis
| TR' -- ^ Turn Cube CounterClockwise on X axis
deriving(Show)

instance Arbitrary Inst where
  arbitrary = elements [F, F', R, R', L, L', B, B', U, U', D, D',
                        MF, MF', MU, MU', MR, MR',
                        TF, TF', TU, TU', TR, TR' ]

-- 0 front, 1 up, 2, right, 3 left, 4 down, 5 back

{-/ @goalState@ genera el cubo ordenado.
-}
goalState = let ful = Corner { z = 1, y = 17, x = 23 }
              fur = Corner { z = 3, y = 19, x = 41 }
              fdr = Corner { z = 9, y = 33, x = 47 }
              fdl = Corner { z = 7, y = 31, x = 29 }
              bul = Corner { z = 51, y = 13, x = 43 }
              bur = Corner { z = 53, y = 11, x = 21 }
              bdr = Corner { z = 59, y = 37, x = 27 }
              bdl = Corner { z = 57, y = 39, x = 49 }
              fu = Edge { fe = 2, xy = 18 }
              fr = Edge { fe = 6, xy = 44 }
              fd = Edge { fe = 8, xy = 32 }
              fl = Edge { fe = 4, xy = 26 }
              mu = Edge { fe = 14, xy = 22 }
              mr = Edge { fe = 16, xy = 42 }
              md = Edge { fe = 36, xy = 48 }
              ml = Edge { fe = 34, xy = 28 }
              bu = Edge { fe = 52, xy = 12 }
              br = Edge { fe = 56, xy = 24 }
              bd = Edge { fe = 58, xy = 38 }
              bl = Edge { fe = 54, xy = 46 }
              f = Center 5
              u = Center 15
              r = Center 45
              l = Center 25
              d = Center 35
              b = Center 55
              in Cube (fromList [ful, fur, fdr, fdl,
                                bul, bur, bdr, bdl])

```

```

        (fromList [fu, fr, fd, fl,
                   mu, mr, md, ml, bu, br, bd, bl])
        (fromList [f,u,r,l,d,b])

{-| @nextMove@ combinador monadico que efectua las acciones
    en el monad @StateT@ e imprime todos los movimientos que
    le va realizando al cubo. En la siguiente entrega aqui
    se hará la impresión con HOpenGL para dibujar el cubo
    en tiempo real mientras se juega.
-}
nextMove:: Inst -> StateT CubeState IO ()
nextMove inst = do
    s <- get
    let newCube = move inst (cube s)
    lift $ print newCube
-- lift $ draw newCube inst
    let a = insts s
    put $ s { cube = newCube, insts = (inst:a) }

{-| @move@ funcion que interpreta la instrucción de movimiento
    a realizarse en el cubo y la realiza.
-}
move:: Inst -> Cube -> Cube
move inst cube =
    let corners = _corners cube
        edges   = _edges cube
        centers  = _centers cube
    in case inst of
        F   -> Cube (rotate [0,1,2,3] (swap 0) corners)
                    (rotate [0,1,2,3] id edges) centers
        F'  -> Cube (rotate [3,2,1,0] (swap 0) corners)
                    (rotate [3,2,1,0] id edges) centers
        R   -> Cube (rotate [1,4,7,2] (swap 1) corners)
                    (rotate [1,5,11,6] id edges) centers
        R'  -> Cube (rotate [2,7,4,1] (swap 1) corners)
                    (rotate [6,11,5,1] id edges) centers
        L   -> Cube (rotate [0,3,6,5] (swap 1) corners)
                    (rotate [3,7,9,4] id edges) centers
        L'  -> Cube (rotate [5,6,3,0] (swap 1) corners)
                    (rotate [4,9,7,3] id edges) centers
        B   -> Cube (rotate [4,5,6,7] (swap 0) corners)
                    (rotate [8,9,10,11] id edges) centers
        B'  -> Cube (rotate [7,6,5,4] (swap 0) corners)
                    (rotate [11,10,9,8] id edges) centers
        U   -> Cube (rotate [0,5,4,1] (swap 2) corners)
                    (rotate [0,4,8,5] swapEdge edges) centers
        U'  -> Cube (rotate [1,4,5,0] (swap 2) corners)
                    (rotate [5,8,4,0] swapEdge edges) centers
        D   -> Cube (rotate [3,2,7,6] (swap 2) corners)

```

```

        (rotate [2,6,10,7] swapEdge edges) centers
D'  -> Cube (rotate [6,7,2,3] (swap 2) corners)
        (rotate [7,10,6,2] swapEdge edges) centers
MF  -> Cube corners (rotate [4,5,6,7] swapEdge edges)
        (rotate [1,2,4,3] id centers)
MF' -> Cube corners (rotate [7,6,5,4] swapEdge edges)
        (rotate [3,4,2,1] id centers)
MU  -> Cube corners (rotate [1,3,9,11] swapEdge edges)
        (rotate [0,3,5,2] id centers)
MU' -> Cube corners (rotate [11,9,3,1] swapEdge edges)
        (rotate [2,5,3,0] id centers)
MR  -> Cube corners (rotate [0,8,10,2] swapEdge edges)
        (rotate [0,1,5,4] id centers)
MR' -> Cube corners (rotate [2,10,8,0] swapEdge edges)
        (rotate [4,5,1,0] id centers)
TF  -> move B' $ move MF $ move F cube
TF' -> move B $ move MF' $ move F' cube
TU  -> move D' $ move MU $ move U cube
TU' -> move D $ move MU' $ move U' cube
TR  -> move L' $ move MR $ move R cube
TR' -> move L $ move MR' $ move R' cube

{-| @rotate@ realiza la rotacion de una secuencia de esquinas
o lados del cubo.
-}
rotate :: [Word8] -> (a -> a) -> (Seq a) -> (Seq a)
rotate list swap sequence =
    let [i,j,k,l] = map (fromIntegral) list
        a = index sequence i
        b = index sequence j
        c = index sequence k
        d = index sequence l
    in update l (swap c) $ update k (swap b) $ update j (swap a)
        $ update i (swap d) sequence

{-| @swapEdge@ realiza el intercambio de valores en las caras
de un lado del cubo.
-}
swapEdge :: Edge -> Edge
swapEdge edge = edge { fe = xy edge,
                      xy = fe edge
                      }

{-| @swap@ realiza el intercambio de valores en las caras de
una esquina del cubo.
-}
swap :: Word8 -> Corner -> Corner
swap 0 corner = corner { y = x corner,
                        x = y corner
                        }

```

```

swap 1 corner = corner { y = z corner,
                        z = y corner
                      }
swap 2 corner = corner { z = x corner,
                        x = z corner
                      }
swap _ corner = error ("Por alguna razón invocamos a swap" ++
                      "con un valor invalido")

{- @noop@ -- Transformación de estado que no hace nada -}
noop :: StateT CubeState IO ()
noop = return ()

{- @initial@ -- Estado inicial del cubo de Rubik -}
initial :: CubeState
initial = CS { cube = goalState, insts = [] }

playRubik :: [Inst] -> StateT CubeState IO ()
playRubik = seq . map nextMove
  where seq [] = noop
        seq l  = foldl1 (>>) l

main = do
  sample <- sample' $ vectorOf 100 $ (arbitrary :: Gen Inst)
  let p = concat sample
  a <- (execStateT (playRubik p) (initial))
  print $ cube a
  return ()

```