{-

Name :- Harshkumar Patel

CSci 450/503, Fall 2014

Homework #4: Sandwich DSL

H. Conrad Cunningham

27 Ocotber 2014

1234567890123456789012345678901234567890123456789012345678901234567890

This is the SandwichDSL program.

-}

module SandwichDSL

where

-- Used functions from these modules in my implementation

import Data.Maybe

import Data.List

--import Data.Set

{- Haskell data type definitions from "Building the DSL" -}

data Platter = Platter [Sandwich]

deriving Show

data Sandwich = Sandwich [Layer]

deriving (Eq,Show)

data Layer = Bread Bread | Meat Meat |

Cheese Cheese | Vegetable Vegetable |

Condiment Condiment

deriving (Eq,Show,Ord)

data Bread = White | Wheat | Rye

deriving (Eq,Show,Ord)

data Meat = Turkey | Chicken | Ham | RoastBeef | Tofu

deriving (Eq,Show,Ord)

data Cheese = American | Swiss | Jack | Cheddar

deriving (Eq,Show,Ord)

data Vegetable = Tomato | Onion | Lettuce | BellPepper

deriving (Eq,Show,Ord)

data Condiment = Mayo | Mustard | Ketchup | Relish | Tabasco

deriving (Eq,Show,Ord)

-- Function type signatures given in section

-- newSandwich :: Bread -> Sandwich

-- addLayer :: Sandwich -> Layer -> Sandwich

-- newPlatter :: Platter

-- addSandwich :: Platter -> Sandwich -> Platter

{- Haskell data type definitions from

"Compiling the Program for the SueChef Controller"

-}

data SandwichOp = StartSandwich | FinishSandwich |

AddBread Bread | AddMeat Meat |

AddCheese Cheese | AddVegetable Vegetable |

AddCondiment Condiment |

StartPlatter | MoveToPlatter | FinishPlatter

deriving (Eq, Show)

data Program = Program [SandwichOp]

deriving (Eq,Show)

--Exercise Set 1

-- Que-1

newSandwich :: Bread -> Sandwich

newSandwich br = Sandwich [Bread br]

addLayer :: Sandwich -> Layer -> Sandwich

addLayer (Sandwich xs) y = Sandwich (y:xs)

newPlatter :: Platter

newPlatter = (Platter [])

addSandwich :: Platter -> Sandwich -> Platter

addSandwich (Platter xs) y = Platter (y:xs)

test\_set1\_q1 = do

putStr "newSandwich Wheat = "

putStrLn (show (newSandwich Wheat))

putStr "Success: "

putStrLn (show ((newSandwich Wheat) == (Sandwich [Bread Wheat])))

putStr "addLayer (Sandwich [Bread Rye]) (Cheese Jack) = "

putStrLn (show (addLayer (Sandwich [Bread Rye]) (Cheese Jack)))

putStr "Success: "

putStrLn (show ((addLayer (Sandwich [Bread Rye]) (Cheese Jack)) == (Sandwich [Cheese Jack,Bread Rye])))

-- Que-2

isBread :: Layer -> Bool

isBread (Bread \_) = True

isBread \_ = False

isMeat :: Layer -> Bool

isMeat (Meat \_) = True

isMeat \_ = False

isCheese :: Layer -> Bool

isCheese (Cheese \_) = True

isCheese \_ = False

isVegetable :: Layer -> Bool

isVegetable (Vegetable \_) = True

isVegetable \_ = False

isCondiment :: Layer -> Bool

isCondiment (Condiment \_) = True

isCondiment \_ = False

test\_set1\_q2 = do

putStr "isBread (Bread White) = "

putStrLn (show (isBread (Bread White)))

putStr "Success: "

putStrLn (show ((isBread (Bread White)) == True))

putStr "isBread (Condiment Mayo) = "

putStrLn (show (isBread (Bread White)))

putStr "Success: "

putStrLn (show ((isBread (Bread White)) == True))

-- Que-3

noMeat :: Sandwich -> Bool

noMeat (Sandwich []) = True

noMeat (Sandwich (x:xs))

| (x == (Meat Turkey)) = False

| (x == (Meat Chicken)) = False

| (x == (Meat Ham)) = False

| (x == (Meat RoastBeef)) = False

| (x == (Meat Tofu)) = False

| otherwise = noMeat (Sandwich xs)

set1\_q3a = noMeat (Sandwich [Bread Rye,Condiment Mayo,Vegetable Onion,Bread Rye])

set1\_q3b = noMeat (Sandwich [Bread Rye,Condiment Mayo,Vegetable Onion,Meat Ham,Bread Rye])

test\_set1\_q3 = do

putStr "noMeat (Sandwich [Bread Rye,Condiment Mayo,Vegetable Onion,Bread Rye]) = "

putStrLn (show (set1\_q3a))

putStr "Success: "

putStrLn (show ((set1\_q3a) == True))

putStr "noMeat (Sandwich [Bread Rye,Condiment Mayo,Vegetable Onion,Meat Ham,Bread Rye]) = "

putStrLn (show (set1\_q3b))

putStr "Success: "

putStrLn (show ((set1\_q3b) == False))

-- Que-4

checkBread :: [Layer] -> Bool -- Function that checks for any bread between top and bottom bread

checkBread [] = True

checkBread xs = foldr (\x acc -> ((x/= Bread White) && (x/= Bread Wheat) && (x/= Bread Rye)) && acc) True xs

checkMeat :: [Layer] -> Bool -- Fucntion to check meat is in order or not

checkMeat [] = True

checkMeat (x:xs)

| (x == Meat Ham || x==Meat Chicken || x==Meat Turkey || x==Meat RoastBeef || x==Meat Tofu) = checkMeat (dropWhile (\y -> or [y==x | x <- [Meat Ham,Meat Chicken,Meat Tofu,Meat RoastBeef,Meat Turkey]]) xs)

| otherwise = False

checkCheese :: [Layer] -> Bool -- Function to check cheese is in order or not

checkCheese [] = True

checkCheese (x:xs)

| (x == Cheese American || x==Cheese Cheddar || x==Cheese Jack || x==Cheese Swiss) = checkMeat (dropWhile (\y -> or [y==x | x <- [Cheese American,Cheese Swiss,Cheese Jack,Cheese Cheddar]]) xs)

| otherwise = checkMeat (x:xs)

checkVegetable :: [Layer] -> Bool -- Function to check vegetable is in order or not

checkVegetable [] = True

checkVegetable (x:xs)

| (x == Vegetable Tomato || x==Vegetable Onion || x==Vegetable Lettuce || x==Vegetable BellPepper) = checkCheese (dropWhile (\y -> or [y==x | x <- [Vegetable Tomato,Vegetable Onion,Vegetable Lettuce,Vegetable BellPepper]]) xs)

| otherwise = checkCheese (x:xs)

checkCondiment :: [Layer] -> Bool -- Function to check condiment is in order or not

checkCondiment [] = True

checkCondiment (x:xs)

| (x == Condiment Mayo ||x== Condiment Mustard || x==Condiment Ketchup || x==Condiment Relish || x==Condiment Tabasco) = checkVegetable (dropWhile (\y -> or [y==x | x <- [Condiment Mayo,Condiment Mustard,Condiment Ketchup,Condiment Relish,Condiment Tabasco]]) xs)

| otherwise = checkVegetable (x:xs)

inOSO :: Sandwich -> Bool

inOSO (Sandwich []) = False

inOSO (Sandwich (x:[])) = False

inOSO (Sandwich (x:y:[])) = (x == y) && (x == Bread White || x == Bread Wheat || x == Bread Rye)

inOSO (Sandwich xs) = ((head xs) == (last xs)) && (head xs == Bread White || head xs == Bread Wheat || head xs == Bread Rye) && (f (xs \\ [head xs,last xs]))

where f ys = ((checkBread ys) && (checkCondiment ys)) -- Recursively checking layers are in OSO or not

-- First checking for top and bottom layer, if they both are same bread, then only checking for remaining list by removing first and last element

set1\_q4a = inOSO (Sandwich [Bread Rye, Condiment Mustard,Vegetable Tomato,Vegetable Onion,Vegetable Lettuce,Cheese Jack,Meat Ham, Meat Tofu,Bread Rye])

set1\_q4b = inOSO (Sandwich [Bread Rye,Meat Ham,Cheese Jack,Meat Tofu,Bread Rye])

test\_set1\_q4 = do

putStr "inOSO (Sandwich [Bread Rye, Condiment Mustard,Vegetable Tomato,Vegetable Onion,Vegetable Lettuce,Cheese Jack,Meat Ham, Meat Tofu,Bread Rye] = )"

putStrLn (show (set1\_q4a))

putStr "Success: "

putStrLn (show ((set1\_q4a) == True))

putStr "inOSO (Sandwich [Bread Rye,Meat Ham,Cheese Jack,Meat Tofu,Bread Rye] = )"

putStrLn (show (set1\_q4b))

putStr "Success: "

putStrLn (show ((set1\_q4b) == False))

intoOSO :: Sandwich -> Bread -> Sandwich

intoOSO (Sandwich []) \_ = Sandwich []

-- Que-5

prices = [(Bread White,20),(Bread Wheat,30),(Bread Rye,30),

(Meat Turkey,100),(Meat Chicken,80),(Meat Ham,120),

(Meat RoastBeef,140),(Meat Tofu,50),

(Cheese American,50),(Cheese Swiss,60),

(Cheese Jack,60),(Cheese Cheddar,60),

(Vegetable Tomato,25),(Vegetable Onion,20),

(Vegetable Lettuce,20),(Vegetable BellPepper,25),

(Condiment Mayo,5),(Condiment Mustard,4),

(Condiment Ketchup,4),(Condiment Relish,10),

(Condiment Tabasco,5) ]

priceSandwich :: Sandwich -> Int

priceSandwich (Sandwich xs) = sum [b | (a,b) <- prices, x <- xs, a==x ]

set1\_q5 = priceSandwich (Sandwich [Bread White,Condiment Relish,Vegetable Onion,Meat Tofu,Bread White])

test\_set1\_q5 = do

putStr "priceSandwich (Sandwich [Bread White,Condiment Relish,Vegetable Onion,Meat Tofu,Bread White]) = "

putStrLn (show (set1\_q5))

putStr "Success: "

putStrLn (show ((set1\_q5) == 120))

-- Que-6

eqSandwich :: Sandwich -> Sandwich -> Bool

eqSandwich (Sandwich xs) (Sandwich ys)

| (((xs \\ ys) == []) && ((ys \\ xs) == []))= True --Using difference operator(\\) in Data.List to find equality of Sandwich

| otherwise = False --If difference is empty then all the layers in Sandwiches would be same.

set1\_q6a = eqSandwich (Sandwich [Bread Rye,Condiment Mayo,Cheese Jack,Meat Ham,Bread Rye]) (Sandwich [Bread Rye,Cheese Jack,Meat Ham,Condiment Mayo,Bread Rye])

set1\_q6b = eqSandwich (Sandwich [Bread Rye,Condiment Mayo,Cheese Cheddar,Meat Ham,Bread Rye]) (Sandwich [Bread Rye,Cheese Jack,Meat Ham,Condiment Mayo,Bread Rye])

test\_set1\_q6 = do

putStr "eqSandwich (Sandwich [Bread Rye,Condiment Mayo,Cheese Jack,Meat Ham,Bread Rye]) (Sandwich [Bread Rye,Cheese Jack,Meat Ham,Condiment Mayo,Bread Rye]) = "

putStrLn (show (set1\_q6a))

putStr "Success: "

putStrLn (show (set1\_q6a == True))

putStr "eqSandwich (Sandwich [Bread Rye,Condiment Mayo,Cheese Cheddar,Meat Ham,Bread Rye]) (Sandwich [Bread Rye,Cheese Jack,Meat Ham,Condiment Mayo,Bread Rye]) = "

putStrLn (show (set1\_q6b))

putStr "Success: "

putStrLn (show ((set1\_q6b) == False))

-- Exercise Set 2

-- Que-1

compileSandwich :: Sandwich -> [SandwichOp]

compileSandwich (Sandwich xs) = [StartSandwich] ++ (foldr (\x acc -> (f x) : acc) [] (reverse xs)) ++ [FinishSandwich,MoveToPlatter]

where f x

| (x == Bread White) = AddBread White

| (x == Bread Wheat) = AddBread Wheat

| (x == Bread Rye) = AddBread Rye

| (x == Meat Turkey) = AddMeat Turkey

| (x == Meat Chicken) = AddMeat Chicken

| (x == Meat Ham) = AddMeat Ham

| (x == Meat RoastBeef) = AddMeat RoastBeef

| (x == Meat Tofu) = AddMeat Tofu

| (x == Cheese American) = AddCheese American

| (x == Cheese Swiss) = AddCheese Swiss

| (x == Cheese Jack) = AddCheese Jack

| (x == Cheese Cheddar) = AddCheese Cheddar

| (x == Vegetable Tomato) = AddVegetable Tomato

| (x == Vegetable Onion) = AddVegetable Onion

| (x == Vegetable Lettuce) = AddVegetable Lettuce

| (x == Vegetable BellPepper) = AddVegetable BellPepper

| (x == Condiment Mayo) = AddCondiment Mayo

| (x == Condiment Mustard) = AddCondiment Mustard

| (x == Condiment Ketchup) = AddCondiment Ketchup

| (x == Condiment Relish) = AddCondiment Relish

| (x == Condiment Tabasco) = AddCondiment Tabasco

-- Que-2

compile :: Platter -> Program

compile (Platter xs) = Program ([StartPlatter] ++ (f xs) ++ [FinishPlatter])

where f [] = []

f (y:ys) = (compileSandwich y) ++ (f ys)

set2 = compile (Platter [Sandwich [Bread Rye,Condiment Mayo,Vegetable Tomato,Cheese Jack,Bread Rye],Sandwich [Bread Rye,Condiment Ketchup,Vegetable Onion,Meat Ham,Meat Tofu,Bread Rye]])

test\_set2 = do

putStr "compile (Platter [Sandwich [Bread Rye,Condiment Mayo,Vegetable Tomato,Cheese Jack,Bread Rye],Sandwich [Bread Rye,Condiment Ketchup,Vegetable Onion,Meat Ham,Meat Tofu,Bread Rye]]) = "

putStrLn (show (set2))

putStr "Success: "

putStrLn (show ((set2) == Program [StartPlatter,StartSandwich,AddBread Rye,AddCheese Jack,AddVegetable Tomato,AddCondiment Mayo,AddBread Rye,FinishSandwich,MoveToPlatter,StartSandwich,AddBread Rye,AddMeat Tofu,AddMeat Ham,AddVegetable Onion,AddCondiment Ketchup,AddBread Rye,FinishSandwich,MoveToPlatter,FinishPlatter]))

------------ End of SandwichDSL----------