Data.hs

{- Author: Richard Eisenberg
File: Data.hs
Demonstrates the use of datatypes in Haskell
-}

define module Data where

-- A day is one of seven possibilities:
define data DayOfWeek
  = Sunday
  | Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
define deriving (Eq, Show)
define -- This last line allows us to compare weekdays for equality (Eq)
define -- and for GHCi to print them (Show)
define -- The constants Sunday, Monday, etc., are called *constructors* of DayOfWeek

define isWeekday :: DayOfWeek -> Bool
define isWeekday Sunday   = False
define isWeekday Saturday = False
define isWeekday _        = True

define nextDay :: DayOfWeek -> DayOfWeek
define nextDay Sunday    = Monday
define nextDay Monday    = Tuesday
define nextDay Tuesday   = Wednesday
define nextDay Wednesday = Thursday
define nextDay Thursday  = Friday
define nextDay Friday    = Saturday
define nextDay Saturday  = Sunday

define -- Datatypes can also hold data. Suppose we have homework during the week,
define -- but not over weekends.
define -- This makes a type synonym, saying that Homework is just a String.
define type Homework = String
define data HWDayOfWeek
  = SundayHW
  | MondayHW Homework
define getHomework :: HWDayOfWeek -> Maybe Homework
define getHomework (MondayHW hw)    = Just hw
define getHomework (TuesdayHW hw)   = Just hw
define getHomework (WednesdayHW hw) = Just hw
define getHomework _                = Nothing

define -------------------------------------------------------------
define -- This type is just like the built-in list, but with different
define -- names.
define data List a
  = Nil
  | Cons a (List a)
Data.hs

73: deriving (Eq, Show)
74:
75: -- List is a *parameterized type*, meaning that it takes a type parameter
76: -- named "a". Like other type variables, this parameter can be anything.
77: -- In the Cons constructor, we see that a Cons holds both an "a" and a
78: -- list.
79:
80: -- get the length of a list
81: myLength :: List a -> Int
82: myLength Nil = 0
83: myLength (Cons _ xs) = 1 + myLength xs
84:
85: -- convert to a normal list
86: toNormalList :: List a -> [a]
87: toNormalList Nil = []
88: toNormalList (Cons x xs) = x : toNormalList xs
89:
90: -- find an element
91: find :: Eq a => a -> List a -> Maybe Int
92: find _ Nil = Nothing
93: find x (Cons y ys)
94:   | x == y = Just 0
95:   | Just n <- find x ys = Just (n+1)
96:   | otherwise = Nothing
97:
98: {- The following types are defined in the Haskell Prelude, which is
99: automatically imported into every module.
100:----------------------------------------------------------
101:-- Here is a definition of a binary search tree:
102:
103: data BST a
104:   = Leaf
105:   | Node (BST a)   -- left child
106:          a         -- data
107:          (BST a)   -- right child
108:   deriving Show
109:    -- We don't derive Eq, because two trees are the same
110:   -- as long as the hold the same data, even if they are
111:   -- structurally distinct
112:
113:   -- insert into a tree
114: insert :: Ord a => a -> BST a -> BST a
115: insert x Leaf = Node Leaf x Leaf
116: insert x (Node left dat right)
117:   | x <= dat  = Node (insert x left) dat right
118:   | otherwise = Node left dat (insert x right)
119:
120:    -- check if an element is in a tree
121: elemBST :: Ord a => a -> BST a -> Bool
122: elemBST _ Leaf = False
123: elemBST x (Node left dat right)
124:   | x == dat = True
125:   | x < dat = elemBST x left
126:   | otherwise = elemBST x right
127:
128:    -- make a tree from the elements in a list
129: insertAll :: Ord a => [a] -> BST a
130: insertAll [] = Leaf
Data.hs

145: insertAll (x:xs) = insert x (insertAll xs)
146:
147: -- some QuickCheck properties that should hold of all trees
148:
149: -- inserting an element means it’s in the tree
150: prop_insertAll :: Int -> [Int] -> Bool
151: prop_insertAll = \x xs -> elemBST x (insert x (insertAll xs))
152:
153: -- If x isn’t in xs, then it’s not in the tree.
154: prop_notInTree :: Int -> [Int] -> Bool
155: prop_notInTree = \x xs -> (x 'elem' xs) || (not (x 'elemBST' insertAll xs))