module Data where

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

-- The constants Sunday, Monday, etc., are called *constructors* of DayOfWeek

-- Is this a weekday?
isWeekday :: DayOfWeek -> Bool
isWeekday Sunday = False
isWeekday Saturday = False
isWeekday _ = True

-- What’s the next day after this one?
nextDay :: DayOfWeek -> DayOfWeek
nextDay Sunday = Monday
nextDay Monday = Tuesday
nextDay Tuesday = Wednesday
nextDay Wednesday = Thursday
nextDay Thursday = Friday
nextDay Friday = Saturday
nextDay Saturday = Sunday

-- Datatypes can also hold data. Suppose we have homework during the week,
-- but not over weekends.

-- This makes a type synonym, saying that Homework is just a String.
type Homework = String
data HWDayOfWeek
  = SundayHW
  | MondayHW Homework
  | TuesdayHW Homework
  | WednesdayHW Homework
  | ThursdayHW Homework
  | FridayHW
  | SaturdayHW
  deriving (Eq, Show)

-- extract the day’s homework, if any
getHomework :: HWDayOfWeek -> Maybe Homework
getHomework (MondayHW hw) = Just hw
getHomework (TuesdayHW hw) = Just hw
getHomework (WednesdayHW hw) = Just hw
getHomework (ThursdayHW hw) = Just hw
getHomework _ = Nothing

-- This type is just like the built-in list, but with different
-- names.
data List a
  = Nil
  | Cons a (List a)
```haskell
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 xs)
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: data Bool
102:   = False
103:   | True
104:
105: data Maybe a
106:   = Nothing
107:   | Just a
108:
109: data [a]
110: = []
111: | a : [a]
112: |
113: -------------------------------
114: -- Here is a definition of a binary search tree:
115:
116: data BST a
117: = Leaf
118: | Node (BST a) -- left child
119: a -- data
120: (BST a) -- right child
121: deriving Show
122:   -- We don’t derive Eq, because two trees are the same
123:   -- as long as the hold the same data, even if they are
124:   -- structurally distinct
125:
126: -- insert into a tree
127: insert :: Ord a => a -> BST a -> BST a
128: insert x Leaf = Node Leaf x Leaf
129: insert x (Node left dat right)
130: | x <= dat = Node (insert x left) dat right
131: | otherwise = Node left dat (insert x right)
132: |
133: -- check if an element is in a tree
134: elemBST :: Ord a => a -> BST a -> Bool
135: elemBST _ Leaf = False
136: elemBST x (Node left dat right)
137: | x == dat = True
138: | x < dat = elemBST x left
139: | otherwise = elemBST x right
140: |
141: -- make a tree from the elements in a list
142: insertAll :: Ord a => [a] -> BST a
143: insertAll [] = Leaf
```
insertAll (x:xs) = insert x (insertAll xs)

-- some QuickCheck properties that should hold of all trees

-- inserting an element means it’s in the tree
prop_insertAll :: Int -> [Int] -> Bool
prop_insertAll = \x xs -> elemBST x (insert x (insertAll xs))

-- If x isn’t in xs, then it’s not in the tree.
prop_notInTree :: Int -> [Int] -> Bool
prop_notInTree = \x xs -> (x 'elem' xs) || (not (x 'elemBST' insertAll xs))