Data.hs

1: {- Author: Richard Eisenberg
2: File: Data.hs
3:
4: Demonstrates the use of datatypes in Haskell
5: -}
6:
7: module Data where
8:
9: -- A day is one of seven possibilities:
10: data DayOfWeek
11:   = Sunday
12:   | Monday
13:   | Tuesday
14:   | Wednesday
15:   | Thursday
16:   | Friday
17:   | Saturday
18:   deriving (Eq, Show)
19:     -- This last line allows us to compare weekdays for equality (Eq)
20:     -- and for GHCi to print them (Show)
21:
22: -- The constants Sunday, Monday, etc., are called *constructors* of DayOfWeek
23:
24: -- Is this a weekday?
25: isWeekday :: DayOfWeek -> Bool
26: isWeekday Sunday   = False
27: isWeekday Saturday = False
28: isWeekday _        = True
29:
30: -- What’s the next day after this one?
31: nextDay :: DayOfWeek -> DayOfWeek
32: nextDay Sunday    = Monday
33: nextDay Monday    = Tuesday
34: nextDay Tuesday   = Wednesday
35: nextDay Wednesday = Thursday
36: nextDay Thursday  = Friday
37: nextDay Friday    = Saturday
38: nextDay Saturday  = Sunday
39:
40: -- Datatypes can also hold data. Suppose we have homework during the week,
41: -- but not over weekends.
42:
43: -- This makes a type synonym, saying that Homework is just a String.
44: type Homework = String
45:
46: -- Note that the constructors’ names have to be different than the names
47: -- above. It is common to use a suffix like we’ve done here.
48: data HWDayOfWeek
49:   = SundayHW
50:   | MondayHW Homework
51:   | TuesdayHW Homework
52:   | WednesdayHW Homework
53:   | ThursdayHW Homework
54:   | FridayHW
55:   | SaturdayHW
56:   deriving (Eq, Show)
57:
58: -- extract the day’s homework, if any
59: getHomework :: HWDayOfWeek -> Maybe Homework
60: getHomework (MondayHW hw)    = Just hw
61: getHomework (TuesdayHW hw)   = Just hw
62: getHomework (WednesdayHW hw) = Just hw
63: getHomework (ThursdayHW hw)  = Just hw
64: getHomework _                = Nothing
65:
66: -------------------------------------------------------------
67:
68: -- This type is just like the built-in list, but with different
69: -- names.
70: data List a
71:   = Nil
72:   | 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:   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:   ----------------------------------------------------------
115:   -- Here is a definition of a binary search tree:
116:
117:   data BST a
118:     = Leaf
119:     | Node (BST a) -- left child
120:       a -- data
121:       (BST a) -- right child
122:   deriving Show
123:   -- We don’t derive Eq, because two trees are the same
124:   -- as long as the hold the same data, even if they are
125:   -- structurally distinct
126:
127:   -- insert into a tree
128:   insert :: Ord a => a -> BST a -> BST a
129:   insert x Leaf = Node Leaf x Leaf
130:   insert x (Node left dat right)
131:     | x <= dat = Node (insert x left) dat right
132:     | otherwise = Node left dat (insert x right)
133:
134:   -- check if an element is in a tree
135:   elemBST :: Ord a => a -> BST a -> Bool
136:   elemBST _ Leaf = False
137:   elemBST x (Node left dat right)
138:     | x == dat = True
139:     | x < dat = elemBST x left
140:     | otherwise = elemBST x right
141:
142:   -- make a tree from the elements in a list
143:   insertAll :: Ord a => [a] -> BST a
144:   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))