```
-- setting the "warn-incomplete-patterns" flag asks GHC to warn you
-- about possible missing cases in pattern-matching definitions
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}

-- see https://wiki.haskell.org/Safe_Haskell
{-# LANGUAGE Safe #-}

module Assessed3 (priceRange , allergyFree ,
                  isValidSpec , checkSpec ,
                  parentDir , openSubDir ,
                  cross , sieveFrom , sequenceFrom)
where

import Types
import Q3Example

-------------------------------------------------------------------------------
---------------- DO **NOT** MAKE ANY CHANGES ABOVE THIS LINE --------------------
-------------------------------------------------------------------------------

import Data.List

priceRange :: Price -> Price -> [Cupcake] -> [Cupcake]
priceRange minBound maxBound [] = []
priceRange (P minBound) (P maxBound) ((CC (P price) recipe) : cs)
        | price >= minBound && price <= maxBound = [(CC (P price) recipe)] ++
priceRange (P minBound) (P maxBound) cs
        | otherwise = []

--such that allergyFree allergens cupcakes returns just those cupcakes that do not
contain any allergens.

allergyFree :: [Ingredient] -> [Cupcake] -> [Cupcake]
allergyFree [] cupcakes = cupcakes
allergyFree _ [] = []
allergyFree ingredients cupcakes = [(CC price recipe) |  (CC price recipe) <-
cupcakes, not (any (`elem` recipe) ingredients)]


{- Exercise 2 -}

sampletin :: Tin
sampletin = [[Nuts], [Dairy,Gluten], [], [Soy]]

checkSpec :: Spec -> Tin -> Bool
checkSpec (HasCup x ingredient) tin =  (ingredient `elem` (tin !! x))
checkSpec (And spec1 spec2) tin = (checkSpec spec1 tin) && (checkSpec spec2 tin)
checkSpec (Or spec1 spec2) tin = (checkSpec spec1 tin) || (checkSpec spec2 tin)
checkSpec (Not spec) tin = not (checkSpec spec tin)


isValidSpec :: Spec -> Tin -> Bool
isValidSpec (HasCup x ingredient) tin = (x >= 0 && x <= (length tin) - 1)
isValidSpec (And spec1 spec2) tin = (isValidSpec spec1 tin) && (isValidSpec spec2
tin)
isValidSpec (Or spec1 spec2) tin = (isValidSpec spec1 tin) && (isValidSpec spec2
tin)
isValidSpec (Not spec) tin = isValidSpec spec tin
```

```
{- Exercise 3 -}


-- data ConvertableDirectory = Dir {
--   dirName :: String,
--   dirContents :: [ ConvertableDirectoryEntry ]
-- } deriving (Eq,Show, Read)

-- data ConvertableDirectoryEntry =
--     FileEntry ConvertableiFile
--   | DirEntry ConvertableDirectory
--   deriving (Eq,Show, Read)

-- data ConvertableiFile = File {
--   fileName :: String,
--   fileContents :: String
-- } deriving (Eq,Show, Read)


-- stringToVar :: [String] -> Maybe ConvertableDirectory
-- stringToVar = readMaybe . unwords

nameOf :: DirectoryEntry -> String
nameOf (DirEntry directory) = dirName directory
nameOf (FileEntry file) = ""


type PreviousDirectories = [Directory]


parentDir :: Breadcrumb -> Maybe Breadcrumb
parentDir (directory, []) = Nothing
parentDir (middle, [ED {entriesBefore = before, enteredDirName = name, entriesAfter
= after}]) = Just (Dir {dirName = name, dirContents = (before ++ [(DirEntry
middle)] ++  after)}, [])
parentDir (middle, (ED {entriesBefore = before, enteredDirName = name, entriesAfter
= after}: rest)) = Just (Dir {dirName = name, dirContents = (before ++ [(DirEntry
middle)] ++  after)}, rest)


openSubDir :: String -> Breadcrumb -> Maybe Breadcrumb
openSubDir searchName (Dir{dirName = focusName, dirContents = contents},
enteredDirectories) | (findDir searchName contents) == Nothing = Nothing

| otherwise = (Just (convertedDir, [ED {entriesBefore = (makeFirstHalf (DirEntry
convertedDir) contents), enteredDirName = focusName, entriesAfter = (makeSecondHalf
(DirEntry convertedDir) contents)}] ++ enteredDirectories))

where

convertedDir :: Directory

convertedDir = convertMaybeDirEntryToDir (findDir searchName contents)




-- split :: DirectoryEntry -> [DirectoryEntry] -> [[DirectoryEntry]]
```

```
-- split dentry (de:des)| dentry == de = [des]
--                      | otherwise = [de] ++ split dentry des

makeFirstHalf :: DirectoryEntry -> [DirectoryEntry] -> [DirectoryEntry]
makeFirstHalf dentry (de:des) | de == dentry = []
                              | otherwise = ([de] ++ makeFirstHalf dentry des)
makeFirstHalf _ [] = []

makeSecondHalf :: DirectoryEntry -> [DirectoryEntry] -> [DirectoryEntry]
-- makeSecondHalf dentry (de:des) firstHalf |  de == (head (firstHalf)) = tail(des)
--                                    |  otherwise = makeSecondHalf dentry des
firstHalf
makeSecondHalf dentry (de:des) | de == dentry  = des
                               | otherwise = makeSecondHalf dentry des
-- makeSecondHalf dentry [de] = []
makeSecondHalf dentry [] = []


-- openSubDir :: String -> Breadcrumb -> Maybe Breadcrumb
-- openSubDir searchName (Dir {dirName = focusName, dirContents = contents},
enteredDirectories)
--                                  | (findDir searchName contents) == Nothing =
Nothing
--                                  | otherwise = Just (convertedDir, [ED
{entriesBefore = (list!!0), enteredDirName = focusName, entriesAfter = (list!!1)}]
++ enteredDirectories)
-- where convertedDir :: Directory
--   convertedDir = convertMaybeDirEntryToDir (findDir searchName dirContents)
--   contentSplit list = splitOn (Dir convertedDir) contents

-- openSubDir :: String -> Breadcrumb -> Maybe Breadcrumb
-- openSubDir searchName (Dir {dirName = focusName, dirContents = contents},
enteredDirectories)
--                                  | findDir searchName contents == Nothing =
Nothing
--                                  | otherwise = Just (convertedDir, [ED
{entriesBefore = (list!!0), enteredDirName = focusName, entriesAfter = (list!!1)}]
++ enteredDirectories)
-- where convertedDir :: Directory
--       convertedDir = convertMaybeDirEntryToDir (findDir searchName dirContents)
--       contentSplit list = splitOn (Dir convertedDir) contents

convertMaybeDirEntryToDir :: Maybe DirectoryEntry -> Directory
convertMaybeDirEntryToDir (Just (DirEntry dir)) = dir
convertMaybeDirEntryToDir _ = undefined

findDir :: String -> [DirectoryEntry]-> Maybe DirectoryEntry
findDir searchName contents | length foundDir > 0 = Just (foundDir !! 0)
                            | otherwise = Nothing
 where
     foundDir :: [DirectoryEntry]
     foundDir = [content | content <- contents, nameOf content == searchName]




{- Exercise 4 -}

cross :: Int -> [Bool] -> [Bool]
```

```
cross num (b:bs) | (b:bs)!!(num -1) == True = iterateThorugh num (num+ 1) (b:bs)
                 | otherwise = (b:bs)
cross num [] = []

-- cross2 :: Int -> [Bool] -> [Bool]
-- cross2 num (b:bs) | (b:bs)!!(num + 1) == True = [True, True] ++ iterateThorugh
num (num + num) (bs)
--                  | otherwise = (b:bs)
-- cross2 num [] = []




-- cross3 :: Int -> [Bool] -> [Bool]
-- cross3 num array = []



cross2 :: Int -> [Bool] -> [Bool]
cross2 num (b:bs) | (b:bs)!!(num - 1) == True = iterateThrough2 num (num + 1) 0
(b:bs)
                  | otherwise = (b:bs)
cross2 num [] = []

--pass in 0 1 2 3  4  5  6  7  8  9 10 11 12
--        2 3 4 5  6 7 8 4 6 8 10  12 14 16 18 20
--
iterateThorugh :: Int -> Int -> [Bool] -> [Bool]
iterateThorugh num value (b:bs) | (value `rem` num) == 0 = [False] ++
iterateThorugh num (value + 1) bs
                                | otherwise = [b] ++ iterateThorugh num (value + 1)
bs
iterateThorugh num value [] = []

iterateThrough2 :: Int -> Int -> Int -> [Bool] -> [Bool]
iterateThrough2 num value position (b:bs) | ((position + 1) >= value) && ((value
`rem` num) == 0) = [False] ++ iterateThrough2 num (value + 1) (position + 1) bs
                                | ((position + 1) >= value) =  [b] ++
(iterateThrough2 num (value + 1) (position + 1) bs)
                                | otherwise = [b] ++ (iterateThrough2 num (value)
(position + 1) bs)
iterateThrough2 num value _ [] = []

-- (element 3 .~ 9) [1,2,3,4,5]
-- [1,2,3,9,5]

sieveFrom :: Int -> [Bool] -> [Bool]
sieveFrom num [] = []
sieveFrom  start array = applyMultipleTimes cross2 1000 start (((repeat True)))



applyMultipleTimes :: (Int -> [Bool] -> [Bool]) -> Int -> Int -> [Bool] -> [Bool]
applyMultipleTimes func 1 num list = func (num) list
applyMultipleTimes func count num [] = []
applyMultipleTimes func count num list = func (num + count)  (applyMultipleTimes
func (count - 1) (num + 1) (func num list))
```

```
-- cross num (cross num + 1 (cross num + 2 list ))

--     doSieve :: Int -> [Bool] -> [Bool]
--           doSieve num (ba:bas) = cross() : doSieve (num + 1) (cross (num + 1)
(bas))
--           doSieve num [] = []


-- doSieve :: Int -> [Bool] -> [Bool]
-- -- doSieve num (ba:bas) = remove
-- doSieve num [] = []

applyNtimes :: (Num n, Ord n) => n -> (a -> a) -> a -> a
applyNtimes 1 f x = f x
applyNtimes n f x = f (applyNtimes (n-1) f x)


removeHeads :: Int -> [a] -> [a]
removeHeads counter (b:bs) | counter > 0 = removeHeads (counter - 1) bs
                                | otherwise = (b:bs)
removeHeads toBeRemoved [] = []

sieve :: [Bool]
sieve = sieveFrom 2 (repeat True)

sequenceFrom :: Int -> [Bool] -> [Int]
sequenceFrom start array = ( (convBoolToNum 0 (start) (removeHeads 1(array))))

-- sequenceFrom :: Int -> [Bool] -> [Int]
-- sequenceFrom start array = ((convBoolToNum 0 (start) ((array))))
-- sequenceFrom :: Int -> [Bool] -> [Int]
-- sequenceFrom start array = ( (convBoolToNum 0 0 (start) ((array))))

convBoolToNum :: Int -> Int -> [Bool] -> [Int]
convBoolToNum count next (b:bs) | b == True = [next + count] ++ convBoolToNum (1)
(next + count) bs
                                | otherwise = convBoolToNum (count + 1) (next) bs
convBoolToNum count next [] = []

-- convBoolToNum :: Int -> Int -> Int -> [Bool] -> [Int]
-- convBoolToNum doneFirst count next (b:bs) | (doneFirst == 0) && ((count + 1) ==
next) = convBoolToNum 1 (0) (next) (b:bs)
--                                           | (doneFirst == 0) && ((count + 1) /=
next) = convBoolToNum 0 (count + 1) (next) bs
--                                           | doneFirst == 1 && b == True = [next
+ count] ++ convBoolToNum 1 (1) (next + count) bs
--                                           | otherwise = convBoolToNum 1 (count +
1) (next) bs
-- convBoolToNum doneFirst count next [] = []

--
[True,True,True,False,True,False,True,False,True,False,True,False,True,False,True,F
alse,True,False,True,False]
-- 0     1    2     3     4    5    6     7
-- 1     2    3     4    5     6     7    8

--0 1 2 3 4
--1 2 3 4
```

```
-- combine :: [Bool] -> [Bool] -> [Bool]
-- combine (f:fs) (s:ss) | f == False || s == False = [False] ++ combine fs ss
--                       | otherwise = [True] ++ combine fs ss
-- combine f [] = []


primes :: [Int]
primes = sequenceFrom 2 sieve

-- temp :: int -> [Bool] -> [[Bool]]
-- temp num list = [cross2 num list | num <- [1..], list <- (cross2 num list)]

-- f xs n = map (usum xs) [1..n]
```