Vivian Do
CSC 4330
Sep 18 2023

Homework 2

1. (15 points: 5 points for each of the three ways).

Following the three ways to define "&&" on slides 6--7 of chapter 4, show the analogous three ways how the disjunction operator "||" can be defined using pattern matching.

```
(||) :: Bool -> Bool -> Bool
True || True = True
True || _ = True
b || b = b
False || b = b
```

2.

a. (10 points). Without using any other library functions or operators, show how the meaning of the following pattern matching definition for logical conjunction "&&" can be expressed using conditional (if/else) expressions:

```
True && True = True
_    && _    = False
```

Hint: use two nested conditional expressions.

a)

```
aAnd :: Bool -> Bool -> Bool
aAnd x y =
      if x == True
            then if y == True
                  then True
                  else False
            else False
```

b. (10 points). Do the same for the following alternative definition:

```
True  && b = b
False && _ = False
```

Hint: note the difference in the number of conditional expressions that are required this time (fewer are be needed).

b)

```
bAnd :: Bool -> Bool -> Bool
bAnd x y =
      if x == True
            then y
            else False
```

3. (15 points). Show how the meaning of the following curried function definition can be formalized in terms of lambda expressions:

```
mult :: Int -> Int -> Int -> Int
mult x y z = x * y * z
```

Hint: see slide 14 of chapter 4.

```
mult :: Int -> (Int -> (Int -> Int))
mult \x -> (\y -> (\z -> x * y * z))
```

4. (20 points).

A positive integer is "perfect" if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension and the function "factors", define a function "perfects :: Int -> [Int]" that returns the list of all perfect numbers up to a given limit. For example (in GHCi):

```
> perfects 500
[6,28,496]
```

Note that "factors" is just:

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], mod n x == 0]
```

from the slides.

Hint: define an auxiliary function "isperfect :: Int -> Bool" which returns "True" if the given "Int" is perfect, and "False" otherwise. Then use "isperfect" as a guard in a list comprehension to filter out all of the non-perfect integers (i.e., to keep only the perfect integers).

```
isperfect :: Int -> Bool
isperfect n = n == sum (factors s) - n
```

- used as a guard to check list for perfect numbers by comparing its sum of its factors

```
perfects :: Int -> [Int]
perfects x = [n | n <- [1..x], isperfect n]
```

- returns a list of “perfect” numbers filtered out from “1” to “x”
- “x” is the given limit of numbers
- “imperfect” filters this list of non-perfect numbers

5. (20 points).

Redefine the function "positions" (from slide 15 of chapter 5) using the function "find". Note that "find" is just:

```
find :: Eq a => a -> [(a,b)] -> [b]
find k t = [v | (k',v) <- t, k == k']
```

from the slides.

Hint: the simplest solution will still involve "zip", but it need not be in a list comprehension anymore, due to the usage of "find".

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (x’, i) <- zip xs [0..], x == x’]
```

- returns a list of numbers from pairing an element in the list with its index

6. (10 points).

The "scalar product" of two lists of integers "xs" and "ys" of length "n" is given by the sum of the products of corresponding integers in slide 21 for chapter 5. In a similar manner to "chisqr" (below), show how a list comprehension can be used to define a function "scalarproduct :: [Int] -> [Int] -> Int" that returns the scalar product of two lists. For example (in GHCi):

```
> scalarproduct [1,2,3] [4,5,6]
32
```

Note that the mathematical equation for computing the "chi-square statistic" at the beginning of subsection "Cracking the cipher" in section 5.5 of the Haskell textbook is literally "zipping" a pair of lists "oe" and "es", and can hence be represented in Haskell as:

```
chisqr :: [Float] -> [Float] -> Float
chisqr os es = sum [(o-e)^2)/e | (o,e) <- zip os es]
```

Since the scalar product of a pair of lists is also "zipping" a pair of lists (but doing something slightly different with the pairs), it will be defined quite similarly, i.e., with a list comprehension involving "zip".

```
scalarproduct :: [Int] -> [Int] -> Int
scalarproduct xs ys = sum [x * y | (x, y) <- zip xs ys]
```

- returns the sum of the two lists from creating tuples from each list pairings & multiplying the elements together