{-# LANGUAGE NPlusKPatterns #-}

{-

Funktionale Programmierung

Übungsblatt (Abgabe: Mo., den 2.11. um 10:10 Uhr)

Author: Boyan Hristov Matrikelnummer: 4980301

Nedeltscho Petrov

Tutor: Zachrau, Alexander

Tutorium: Dienstag; 12:00 - 14:00

-}

{-

----------------------------------------------------------------------------------------------------------

1. Exercise

The function is not correct, as it checks whether the number is even or not,

but only for positive numbers. Cases where the number is negative are not

handled, which is a bad practise. It can easily be fixed with mod, which returns a positive rest.

----------------------------------------------------------------------------------------------------------

-}

ungerade :: Integer -> Bool

ungerade n = mod n 2 == 1 --I assume negative numbers can be odd or even too, if not, a guard for <0

-- must be added, where error "Negative numbers cannot be even/odd" is thrown

{-

----------------------------------------------------------------------------------------------------------

2. Exercise

----------------------------------------------------------------------------------------------------------

-}

type Point = (Double, Double)

type Rectangle = (Point, Point)

--I am checking if pointA is above pointB and to the left of pointB, so the rectangle is defined properly

--As per definition 2nd exercise gives no errors => a rect can have a side of length 0

-- (in example tests)

isRectCorrect :: Rectangle -> Bool

isRectCorrect (pointA, pointB) = ( fst pointA <= fst pointB ) && (snd pointA >= snd pointB )

--This error will be thrown in all 3 rectangle functions, so I define a function to prevent repetitive code

reportRectError :: error

reportRectError = error "The given rectangle is not correct! Please enter top-left corner first, then bottom-right."

--points could be in the negative partitions, therefore I take distance between x and y as absolute numbers

area :: Rectangle -> Double

area rect = if (isRectCorrect rect)

then abs (fst pointB - fst pointA) \* abs (snd pointA - snd pointB)

else reportRectError

where

pointA = fst rect

pointB = snd rect

contains :: Rectangle -> Rectangle -> Bool -- Tests, whether a rectangle is in the other rectangle or not

contains rectA rectB = if (isRectCorrect rectA) && (isRectCorrect rectB)

then withinX && withinY

else

reportRectError

where

-check if the x of pointA top corner is smaller that this of pointB

--and if x of pointA bottom corner is bigger that this of pointB

--this way rectA contains rectB, when looking only at X

withinX = (fst (fst rectA)) < (fst (fst rectB)) &&

(fst (snd rectA)) > (fst (snd rectB))

withinY = (snd (fst rectA)) > (snd (fst rectB)) &&

(snd (snd rectA)) < (snd (snd rectB))

overlaps :: Rectangle -> Rectangle -> Bool -- Tests, whether one rectangle overlaps the other or not

overlaps rectA rectB = if (isRectCorrect rectA) && (isRectCorrect rectB)

--I check if any side of 2nd rect is within 1st rect

--If not, then only chance to overlap is if 2nd rect contains 1st

--case contains (rectA rectB) is already checked in the xOverlps && yOverlaps

then (xOverlaps && yOverlaps) || (contains rectB rectA)

else

reportRectError

where --1st case, left side of 2nd rect within 1st rect

xOverlaps = (leftSideWithin rectA rectB) || (leftSideWithin rectB rectA)

--2nd case, right side of 2nd rect within 1st rect

-- => left side of 1st rect within 2nd rect

--1st case, top side of 2nd rect within 1st rect

yOverlaps = (topSideWithin rectA rectB) || (topSideWithin rectB rectA)

--2nd case, bottom side of 2nd rect within 1st rect

-- => top side of 1st rect within 2nd rect

leftSideWithin rect1 rect2 = ( (fst (fst rect1)) < (fst (fst rect2)) ) &&

( (fst (snd rect1)) > (fst (fst rect2)) )

topSideWithin rect1 rect2 = ( (snd (fst rect1)) > (snd (fst rect2)) ) &&

( (snd (snd rect1)) < (snd (fst rect2)) )

{-

----------------------------------------------------------------------------------------------------------

3. Exercise

----------------------------------------------------------------------------------------------------------

-}

collList :: Integer -> [Integer]

collList 1 = [1]

collList (n+1) = (n+1): collList (next (n+1))

where

next n | mod n 2 == 0 = div n 2

| otherwise = 3\*n + 1

{-

collList 5 => 5: collList (next 5) => 5: collList (16) =>

=> 5 : 16 : collList (8) => 5: 16 : 8 : collList (4) =>

=> 5 : 16 : 8 : 4 : collList(2) => 5 : 16 : 8 : 4 : 2 : collList(3\*0 + 1) =>

=> 5 : 16 : 8 : 4 : 2 : [1] => 5 : 16 : 8 : 4 : [2, 1] => 5 : 16 : 8 : [4, 2, 1] =>

5 : 16 : [8, 4, 2, 1] => 5 : [16, 8, 4, 2, 1] => [5, 16, 8, 4, 2, 1]

-}

{-

----------------------------------------------------------------------------------------------------------

4. Exercise

----------------------------------------------------------------------------------------------------------

-}

sumDigits :: Int -> Int

sumDigits number

| number < 0 = sumDigits number – my algorithm works for positive number only, the digit sum is equal for both cases

| number < 10 = number --if number consists of only 1 digit, return it

| otherwise = quickSum number 0 --otherwise sum digits, sum starts with 0

where

quickSum 0 sum --div returns only whole nums, by 0 quickSum summed all digits

| sum < 10 = sum --if sum consists of 1 digit, return it

| otherwise = quickSum sum 0 --otherwise, sum the digits of sum

--add the last digit to sum and remove it

quickSum num sum = quickSum (div num 10) (sum + (mod num 10))

{-

----------------------------------------------------------------------------------------------------------

5. Exercise

As I unedrstood, we must calclulate the sum BETWEEN 1 and n, so

I calculate 2+3+4+...+ (n-1)

----------------------------------------------------------------------------------------------------------

-}

sumUpToN :: Integer -> Integer

sumUpToN n = sumBetween1AndN n 0 2

where sumBetween1AndN num sum crnt = if (num == crnt) --If the crnt iteration is num

then sum --return it without summing, only untill n-1!

else sumBetween1AndN num (sum + crnt) (crnt + 1)

--otherwise, call again with increased sum and iteration counter

{-

----------------------------------------------------------------------------------------------------------

6. Exercise

----------------------------------------------------------------------------------------------------------

-}

--I use this function signature, as the signature for error is error :: [Char] -> a

throwError :: a

throwError = error "Input is not a boolean. Enter a valid binary boolean!"

true :: Int

true = 1

false :: Int

false = 0

und :: Int -> Int -> Int

und boolA boolB

| boolA > 1 || boolA < 0 || boolB > 1 || boolB < 0 = throwError

| otherwise = boolA\*boolB

oder :: Int -> Int -> Int

oder boolA boolB

| boolA > 1 || boolA < 0 || boolB > 1 || boolB < 0 = throwError

| otherwise = (boolA + boolB) - (boolA\*boolB)

negation :: Int -> Int

negation bool

| bool > 1 || bool < 0 = throwError

| otherwise = bool - (bool + bool) + true

exoder :: Int -> Int -> Int

exoder boolA boolB

| boolA > 1 || boolA < 0 || boolB > 1 || boolB < 0 = throwError

| otherwise = und (oder boolA boolB) (negation (und boolA boolB) )

hamming\_distance :: [Int] -> [Int] -> Int

hamming\_distance bitListA bitListB

| not (length bitListA == length bitListB) = error "The bit lists have different lengths!"

| otherwise = getHammingDistance bitListA bitListB 0

where

getHammingDistance [] [] sum = sum

getHammingDistance (x:xs) (y:ys) sum = getHammingDistance xs ys (sum + exoder x y)

{-

----------------------------------------------------------------------------------------------------------

7. Exercise

----------------------------------------------------------------------------------------------------------

-}

insertElem :: Char -> Int -> [Char] -> [Char]

insertElem char index arr

| (index < 0) || (index > length(arr)) = error "Out of bounds exception!"

| otherwise = insertAtIdx char index arr 0 []

where

insertAtIdx c idx (x:xs) crnt pastArr | (idx == crnt) = pastArr ++ (c : x : xs) | otherwise = insertAtIdx c idx xs (crnt + 1) (pastArr ++ [x])