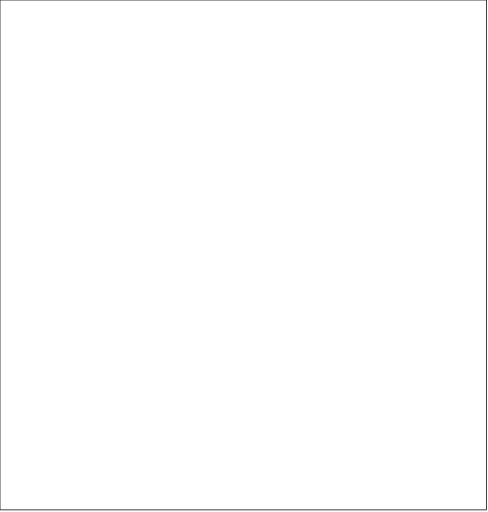
LAB WEEK 1

-- triangleArea a b c computes the area of a traingle with side lengths a, b and c respectively

triangleArea :: Float -> Float -> Float -> Float

triangleArea a b c = sqrt (s\*(s-a)\*(s-b)\*(s-c))

where

s = (a+b+c)/2

isSum :: Int -> Int -> Int -> Bool

isSum x y z

| x + y == z = True

| x + z == y = True

| z + y == x = True

| otherwise = False

triangleArea2 :: Float -> Float -> Float -> Float

triangleArea2 a b c

| a + b < c = error "Not a triangle!"

| b + c < a = error "Not a triangle!"

| a + c < b = error "Not a triangle!"

| otherwise = sqrt (s\*(s-a)\*(s-b)\*(s-c))

where

s = (a+b+c)/2

-- fibonacci computes the nth number in the fibonacci sequence using guards

fibonacci :: Int -> Int

fibonacci n

| n==1 = 0

| n==2 = 1

| n>2 = fibonacci (n-1) + fibonacci (n-2)

-- fibonacci2 computes the nth number in the fibonacci sequence using pattern matching

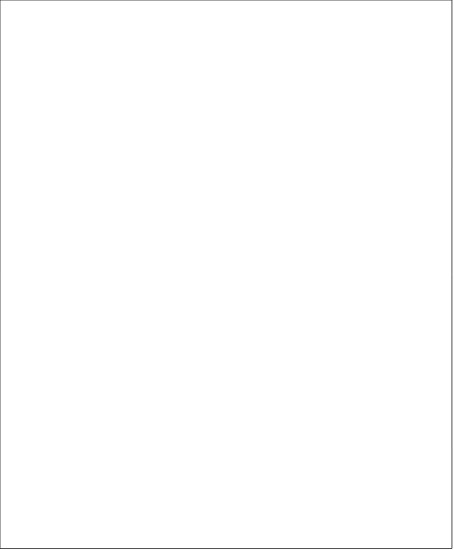
fibonacci2 :: Int -> Int

fibonacci2 1 = 0

fibonacci2 2 = 1

fibonacci2 n = fibonacci2 (n-1) + fibonacci2 (n-2)

LAB WEEK 2

import Data.Char(toUpper)

-- myProduct xs returns the product of the numbers in the list xs

myProduct :: [Int] -> Int

myProduct [] = 1

myProduct (x:xs) = x \* myProduct xs

-- stringToUpper xs converts all lower case letters in the string xs to upper case

stringToUpper :: String -> String

stringToUpper [] = []

stringToUpper (x:xs) = (toUpper x):stringToUpper xs

-- shortest xss returns the shortest list in the list of lists xss

shortest :: [[a]] -> [a]

shortest [] = []

shortest [xs] = xs

shortest (xs:xss) = let ys = shortest xss

in if length xs < length ys then xs else ys

-- isPalindrome xs indicates whether or not the list xs is a palindrome

isPalindrome :: Eq a => [a] -> Bool

isPalindrome [] = True

isPalindrome [\_] = True

isPalindrome xs = (head xs == last xs) && (isPalindrome (tail (init xs)))

-- sumPoly xs ys adds two polynomials represented by xs and ys

sumPoly :: [Int] -> [Int] -> [Int]

sumPoly xs [] = xs

sumPoly [] ys = ys

sumPoly (x:xs) (y:ys) = (x+y):sumPoly xs ys

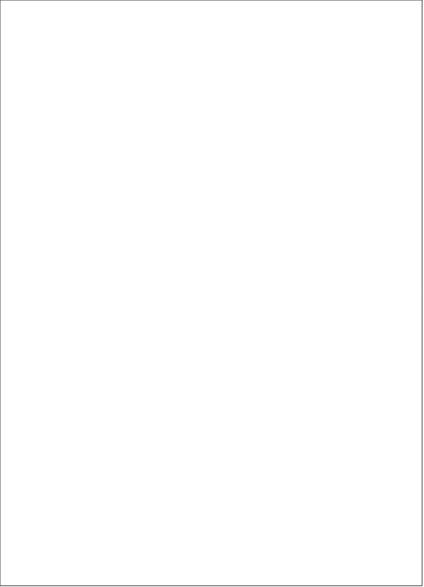
-- evalPoly x xs evaluates the polynomial represented by xs for the value of x

evalPoly :: Int -> [Int] -> Int

evalPoly x [] = 0

evalPoly x (a:as) = a + x \* evalPoly x as

LAB WEEK 3

import Data.List(nub)

-- delFirst x ys returns the list ys with the first occurrence of item x removed.

delFirst :: Eq a => a -> [a] -> [a]

delFirst x [] = []

delFirst x (y:ys)

| x==y = ys

| otherwise = y:delFirst x ys

-- delAll x ys deletes all instances of the value x from the list ys

delAll :: Eq a => a -> [a] -> [a]

delAll x [] = []

delAll x (y:ys)

| x==y = delAll x ys

| otherwise = y:delAll x ys

-- num x ys returns the number of times the value x occurs in the list ys

qqnum :: Eq a => a -> [a] -> Int

num y [] = 0

num y (x:xs)

| x==y = 1 + num y xs

| otherwise = num y xs

-- numSorted x ys returns the number of times the value x occurs in the sorted list ys

numSorted :: Ord a => a -> [a] -> Int

numSorted y [] = 0

numSorted y (x:xs)

| x==y = 1 + numSorted y xs

| x>y = 0

| otherwise = numSorted y xs

-- repFirst x y zs replaces the first occurrence of the value x in the list ys with the value z

repFirst :: Eq a => a -> a -> [a] -> [a]

repFirst x y [] = []

repFirst x y (z:zs)

| x==z = y:zs

| otherwise = z:repFirst x y zs

-- repAll x y zs replaces all occurrences of the value x in the list zs with the value y

repAll :: Eq a => a -> a -> [a] -> [a]

repAll x y [] = []

repAll x y (z:zs)

| x==z = y:repAll x y zs

| otherwise = z:repAll x y zs

-- elemSorted x ys determines whether the value x is contained in the sorted list ys

elemSorted :: Ord a => a -> [a] -> Bool

elemSorted x [] = False

elemSorted x (y:ys)

| x==y = True

| x>y = False

| otherwise = elemSorted x ys

-- nubSorted xs removes duplicates from the sorted list xs

nubSorted :: Eq a => [a] -> [a]

nubSorted [] = []

nubSorted [x] = [x]

nubSorted (x:y:ys)

| x==y = nubSorted (y:ys)

| otherwise = x:nubSorted (y:ys)

-- dupSorted xs determines if there are duplicate elements in the sorted list xs

dupSorted :: Eq a => [a] -> Bool

dupSorted [] = False

dupSorted [x] = False

dupSorted (x:y:ys)

| x==y = True

| otherwise = dupSorted (y:ys)

-- dup xs determines if there are duplicate elements in the unsorted list xs

dup :: Eq a => [a] -> Bool

dup [] = False

dup (x:xs) = (x `elem` xs) || (dup xs)

LAB WEEK 4

data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday

deriving (Enum,Show)

data Month = Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec

deriving (Enum,Read)

type Date = (Int,Month,Int)

-- leap y returns a boolean indicating whether or not year y is a leap year

leap :: Int -> Bool

leap y

| y `mod` 100 == 0 = y `mod` 400 == 0

| y `mod` 4 == 0 = True

| otherwise = False

-- mLengths y returns a list of the month lengths for year y

mLengths :: Int -> [Int]

mLengths y = [31,feb,31,30,31,30,31,31,30,31,30,31]

where

feb = if leap y then 29 else 28

-- numDays d returns the number of days since 31/12/1752 for date d

numDays :: Date -> Int

numDays (day,month,year)

= day -- days this month

+ sum (take (fromEnum month) (mLengths year)) -- days this year

+ (year-1753) \* 365 + length [yr | yr <- [1753..year-1], leap yr] -- days since adoption of Gregorian calendar

-- dayOfWeek d returns the name of the day for the given date d

dayOfWeek :: Date -> Day

dayOfWeek d = toEnum (((numDays d)-1) `mod` 7)

data Tree a = Null | Node a (Tree a) (Tree a)

deriving (Read,Show)

-- addNode n t adds node n to binary search tree t

addNode :: Ord a => a -> Tree a -> Tree a

addNode m Null = Node m Null Null

addNode m (Node n t1 t2)

| m<n = Node n (addNode m t1) t2

| otherwise = Node n t1 (addNode m t2)

-- makeTree ns makes a binary search tree from the list of nodes ns

makeTree :: Ord a => [a] -> Tree a

makeTree [] = Null

makeTree (x:xs) = addNode x (makeTree xs)

-- inOrder t returns the list of nodes in an inorder traversal of tree t

inOrder :: Tree a -> [a]

inOrder Null = []

inOrder (Node n t1 t2) = (inOrder t1) ++ [n] ++ (inOrder t2)

-- mpSort ns performs a monkey puzzle sort on the input list ns

mpSort :: Ord a => [a] -> [a]

mpSort = inOrder . makeTree

LAB WEEK 5

import Data.Char (toUpper,isUpper)

-- iter n f x applies function f n times with a starting value given by x

iter 0 f x = x

iter n f x = f (iter (n-1) f x)

-- pow x n raises x to the power of n

pow x n = iter n (\*x) 1

-- myzipWith implements the standard library function zipWith

myzipWith f (x:xs) (y:ys) = (f x y):(myzipWith f xs ys)

myzipWith \_ \_ \_ = []

-- functions defined using map

capitalise = map toUpper

squareall = map (^2)

prepend xs = map (xs++)

-- functions defined using filter

wc = filter (\c -> not (isUpper c))

we = filter (\n -> n `mod` 2 /= 0)

wv = filter (\x -> x `notElem` "aeiouAEIOU")

-- functions defined using foldr

sumsquaresr n = foldr (\x y -> x^2+y) 0 [1..n]

lengthr xs = foldr (\x y -> y+1) 0 xs

reverser xs = foldr (\x ys -> ys++[x]) [] xs

-- functions defined using foldl

sumsquaresl n = foldl (\x y -> y^2+x) 0 [1..n]

lengthl xs = foldl (\x y -> x+1) 0 xs

reversel xs = foldl (\ys x -> x:ys) [] xs

SAMPLE EXAM SOLUTIONS

myGCD :: Int -> Int -> Int

myGCD n 0 = n

myGCD m n = myGCD n (m `mod` n)

dectobin :: Int -> String

dectobin 0 = "0"

dectobin 1 = "1"

dectobin d

| d `mod` 2 == 0 = (dectobin (d `div` 2)) ++ "0"

| otherwise = (dectobin (d `div` 2)) ++ "1"

data Circuit = Component Float | Serial Circuit Circuit | Parallel Circuit Circuit

resistance :: Circuit -> Float

resistance (Component r) = r

resistance (Serial c1 c2) = resistance c1 + resistance c2

resistance (Parallel c1 c2) = 1 / (1/resistance c1 + 1/resistance c2)