```
{-
Notes on Haskell.
To make a PDF of this handout:
enscript -C -2r -M Letter -o hCS1.ps haskellCS1.hs
ps2pdf hCS1 haskellCS1.pdf

To compile this file:  ghc haskellCS1.hs
To load into ghci:
ghci
:load haskellCS1
:reload
:?   for help
:t   to see the type of an object
:q   to quit

References:
Programming in Haskell by Graham Hutton
Beginning Haskell by Alejandro Serrano Mena
-}

import qualified Data.Char as Char   -- some libraries that we need
import System.Random                 -- a library for random numbers
import Data.Ratio                    -- a library for rational numbers

-- It's good to have explicit function signatures
increment :: Int -> Int     -- but all functions have signatures
increment x = x+1           -- as well as definitions

-- verify that rational numbers work
ratioTest :: Integer -> Integer -> Rational
ratioTest x y = 1 % x + 1 % y

-- sum is builtin, but
sum1toN :: Integer -> Integer
sum1toN n = sum [1..n]

-- easy enough to implement
mySum :: [Integer] -> Integer
mySum [] = 0
mySum (x:xs) = x+mySum(xs)

-- last type given is the type of the answer
-- others are types of parameters
-- inspired by Cartesian product notion from set theory, and Currying
and1 :: Bool -> Bool -> Bool
and1 x y =
  if x==True && y==True then True else False

and2 :: Bool -> Bool -> Bool   -- two Bool input args
and2 True True = True          -- and pattern matching
and2 _ _       = False         -- with underscore as a wildcard

fact :: Integer -> Integer
fact 0 = 1
fact n = n*fact(n-1)

fact2 :: Integer -> Integer
fact2 0 = 1
fact2 n = product[1..n]  -- but using the builtin product function is faster

-- basic list functions from Hutton Chapter 2
listDemo = do
  let aList = [1,2,3,4,5]
  putStrLn ("aList is "++ show(aList))
  putStrLn ("head of aList is "++ show(head aList))
  putStrLn ("tail of aList is "++show(tail aList))
  putStrLn ("aList!!2 is "++show(aList !! 2))
  putStrLn ("take 3 aList is "++show(take 3 aList))
  putStrLn ("drop 3 aList is "++show(drop 3 aList))
  putStrLn ("[1,2,3]++[4,5] is "++show([1,2,3]++[4,5] ))
  putStrLn ("reverse aList is "++show(reverse aList ))
  putStrLn ("myInit aList is "++show(myInit aList))
  putStrLn ("myInit2 aList is "++show(myInit2 aList))
--  putStrLn (" "++show( ))   -- in case we want to add more

-- from end of Hutton Chapter 2 slides
--myInit:: [] -> []
myInit [] = []
myInit (x:xs) =
       if null xs then []
       else [x]++myInit xs

--myInit2:: [] -> []
myInit2 [] = []
myInit2 aList = reverse(tail(reverse(aList)))

-- polymorphic functions!
-- using old friend quicksort
qsortP :: Ord a => [a] -> [a]
qsortP [] = []
qsortP (x:xs) = qsortP lowerHalf ++ [x] ++ qsortP upperHalf
               where
                  lowerHalf = [a | a <- xs, a <= x]
                  upperHalf = [b | b <- xs, b > x]

-- quadratic formula
--roots :: Float -> Float -> Float -> (Float, Float)
roots a b c =
  if discrim<0 then (0,0)
  else (x1, x2) where
    discrim = b*b - 4*a*c
    e = -b/(2*a)
    x1 = e + sqrt discrim / (2*a)
    x2 = e - sqrt discrim / (2*a)

--- some list functions
listLen1 :: [a] -> Int
listLen1 []     = 0
listLen1 (x:xs) = 1 + listLen1(xs)

-- here's another (faster) way to do listLen
listLen2 :: [a] -> Int
listLen2 = sum . map (const 1)    -- . is explicit function composition

-- yet another way to do list length, using a list comprehension
listLen3 :: [a] -> Int
listLen3 aList = sum [1 | x <- aList]

listLenDemo = do
  putStrLn "listLenDemo"
  let aList = [1,3,2,4,7,5]
  putStrLn ("demo of listLen1 " ++ show(listLen1(aList)))
  putStrLn ("demo of listLen2 " ++ show(listLen2(aList)))
  putStrLn ("demo of listLen3 " ++ show(listLen3(aList)))

demo1 = do
  putStrLn "demo1"
  putStrLn ("demo of increment - should be 4:  " ++ show(increment(3)))
  putStrLn ("demo of logical constants, should be True:  " ++ show(0==0))
  putStrLn ("demo of logical constants, should be False:  " ++ show(0==1))
  putStrLn ("demo of and1 - should be True:  " ++ show(and1 True True))
  putStrLn ("demo of and1 - should be False:  " ++ show(and1 False True))
  putStrLn ("demo of and2 - should be True:  " ++ show(and2 True True))
  putStrLn ("demo of and2 - should be False:  " ++ show(and2 False True))
  putStrLn ("demo of sum1toN - should be 15:  " ++ show(sum1toN 5))
  putStrLn ("demo of fac - should be 720:  " ++ show(fact(6)))
```

```
  putStrLn "demo of polymorphic version, qsortP"
  putStrLn ("aList is " ++ show([3, 14, 15, 9, 26]))
  putStrLn ("qsortP aList is " ++ show(qsortP [3, 14, 15, 9, 26]))
  putStrLn ("bList is " ++ show(["Frodo","Bilbo","Smaug","Pippin","Gandalf"]))
  putStrLn ("qsortP aList is " ++
           show(qsortP ["Frodo","Bilbo","Smaug","Pippin","Gandalf"]))
  putStrLn "demo of roots"
  putStrLn (show(roots 2.0 1.0 1.0))   -- NaN
  putStrLn (show(roots 2.0 6.0 1.0))   -- normal output

-- ord ch is the ASCII code for any character ch
-- Haskell strings are lists of characters, so all the list functions work
-- including map
code    x = map Char.ord x     -- string -> [Int]
uncode ch = map Char.chr ch    -- [Int] -> string

demoAscii = do
  let aString = "foobar"
  putStrLn ("demo of code:  " ++ show(code(aString)))
  putStrLn ("demo of uncode:  " ++ show(uncode(code(aString))))

isVowel 'a' = True
isVowel 'e' = True
isVowel 'i' = True
isVowel 'o' = True
isVowel 'u' = True
isVowel x = False

-- using if/then/else
anyVowels [] = False
anyVowels (c:cs) = if isVowel(c) then True else anyVowels(cs)

-- using guards
anyVowels2 [] = False
anyVowels2 (c:cs)
  | isVowel(c) = True
  | otherwise  = anyVowels2(cs)

-- using map
anyVowels3 [] = False
anyVowels3 aString = or (map isVowel aString)

-- using filter
anyVowels4 [] = False
anyVowels4 aString = if vlen > 0 then True else False
  where vlen = length (filter isVowel aString)

-- if you don't want to use the built-in sum function :-)
sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList(xs)

sumList2 :: [Int] -> Int
sumList2 aList = foldr (+) 0 aList

-- an example of a high-order function and a lambda expression
squaresSequence :: Int -> [Int]
squaresSequence n = map (\x -> x^2) [1..n]

-- list comprehension examples inspired by Hutton Chapter 5
squaresSequence2 :: Int -> [Int]
squaresSequence2 n = [x^2 | x <- [1..n]]

somePairs = [(x,y) | x<-[1,2,3], y<-[4,5]]
somePairs2 = [(x,y) | y<-[4,5], x<-[1,2,3]]

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

isPrime n = factors n == [1,n]

zipDemo = print(zip [1,3..9] [0,2..8])

pairs  :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)

sorted :: Ord a => [a] -> Bool
sorted xs =
       and [x <= y | (x,y) <- pairs xs]

-- exercise 3 from end of Chapter 5 slides
dotProduct :: [Int] -> [Int] -> Int
dotProduct aList bList = sum [(a*b) | (a,b) <- zip aList bList]

demo2 = do
  let aList = [1,2,4,7,9]
  putStrLn ("length of aList, according to listLen1, is " ++ show(listLen1 aList))
  putStrLn ("length of aList, according to listLen2, is " ++ show(listLen2 aList))
  putStrLn ("sum of aList, according to sumList, is " ++ show(sumList aList))
  putStrLn ("sum of aList, according to sumList2, is " ++ show(sumList2 aList))
  let string1 = "great big cats"
--  let string1 = "grt bg cts"
  putStrLn ("anyVowels( "++string1++" ) is "++show(anyVowels string1))
  putStrLn ("anyVowels2( "++string1++" ) is "++show(anyVowels2 string1))
  putStrLn ("anyVowels3( "++string1++" ) is "++show(anyVowels3 string1))
  putStrLn ("anyVowels4( "++string1++" ) is "++show(anyVowels4 string1))
  zipDemo

main = do
  demo1
  listDemo
  listLenDemo
  demo2
  demoAscii
```