f x = if x > 3 then 'a' else 'b'

f x

| x > 3 = 'a'

| otherwise = 'b'

printSums[[3,4],[9],[],[3,1,2]] 7 9 0 6

printsums :: [[Int]] -> IO ()

printsums [] = putStrLn "Empty!"

printsums xss = mapM\_ printSum (zip [1..] xss)

where

printSum (i, xs) = putStrLn $ show i ++ ": " ++ show (sum xs)

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

largers = zipWith max

Explain why negate is considered polymorphic in Haskell. Which type class allows this?

It is considered polymorphic because “negate” can work with different numeric types.

The “Num” type class allow this.

Rewrite the following expression, removing unnecessary parentheses: f(x) + g(y) + (3\*y) f x + g y + 3 \* y

An **expression** is any code that evaluates to a value. It produces (returns) a result, and this value can be used, assigned to a variable, or passed as an argument. For example, x + 1 or len("Hello") are expressions because they compute and yield a value

A **statement** is an instruction that performs an action, but does not itself return a value. Statements are used to control the flow of a program, perform assignments, or invoke functions, but they do not yield a result that can be used as a value. For example, print("Hello") or a variable declaration like int x; are statements—they instruct the computer to do something, but do not themselves produce a value

.a = b = 3 (in C)

**Imperative programming**: Instructions change memory locations or registers. Branching instructions alter the flow of control

**procedural programming**: Programs are composed of bodies of code (procedures) that manipulate individual

data elements or structures. Procedures encapsulate complexity.

**Expression**: a sequence of symbols that can be evaluated to produce a value

side effect: something that happens in addition to the computation of an expression's value. It must be "observable". (hallmark of imperative programing)

**functional paradigm**: writing functions that are like pure mathematical functions

**Haskell** is a lazy and pure functional programming language/statically typed, with a very elaborate type system/ not object-oriented in any way

**Type class**: A collection of types that support a specified set of operations

use :set +t to direct ghci to automatically show types

use :: type to constrain a function's type:

Using a tuple lets type-checking ensure that an exact number of values is being aggregated, even if all values have the same type.

A **pattern** can be:

• A literal value such as 1, 'x', or True

• An identifier (bound to a value if there's a match)

• An underscore (the wildcard pattern)

• A tuple composed of patterns

• A list of patterns in square brackets (fixed size list)

• A list of patterns constructed with : operators

• Other things we haven't seen yet

A **higher-order function** is a function that (and/or)

• Has one or more arguments that are functions

• Returns a function

**map** is a Prelude function that applies a function to each element of a list, producing a new list

A **folding** function NEVER sees the list!

**map**: transforms a list of values length (input == length output)

**filter**: selects values from a list (0 <= length output <= length input)

**folding**: Input: An initial accumulator value and a list of values. Output: A value of any type and complexity

greet name = “ Hello, ” ++ name ++ “!”

negate 3 3 (negate is a polymorphic function. It handles values of many forms.)

even 5 False

pred 'C' ‘B’

signum 2 1

signum (negate 2) -1

negate (3 + 4) -7

signum (negate (3 + 4)) -1

import Data.Char

:type isLower isLower :: Char -> Bool

toUpper 'a’ “A”

ord 'A' 65

chr 66 ‘B’

:type negate negate :: Num a => a -> a

: type 3.4 3.4 :: a => a

truncate 7.999 7

double x = x \* 2

neg x = -x

toCelsius temp = (temp - 32) \* 5/9

neg x = -x :: Int neg :: Int -> Int

add x y = x + y :: Int add :: Int -> (Int -> Int)

min 6 2 2

min3 5 2 10 2

min3 a b c = min a (min b c) :: Int

plusThree = add 3 plusThree :: Int -> Int

s a t = 0.5 \* a \* t^2 Formula for displacement (s) of a falling object:

words "a test for words" ["a","test","for","words"]

map length it [1,4,3,5]

hwrap t s = "<" ++ t ++ ">" ++ s ++ "</" ++ t ++ ">"

hwrap "code" "print(3)" <code>print(3)</code>"

bold = hwrap "b"

bold "test" "<b>test</b>"

bold "Not" ++ " again, " ++ bold "<b>Not</b> again, <b><u>never!</u></b>"

uline = hwrap "u"

(+) 3 4 7

3 `add` 4 7

isCapital c = c >= 'A' && c <= 'Z' isCapital :: Char -> Bool

isPositive x = x > 0 isPositive :: (Num a, Ord a) => a -> Bool

smaller :: Ord a => a -> a -> a

smaller x y

| x <= y = x

| otherwise = y

weather :: Int -> String

weather temp

| temp >= 80 = "hot"

| temp >= 70 = "nice"

| otherwise = "cold"

factorial :: Integer -> Integer

factorial 0 = 1

factorial n = n \* factorial (n - 1)

factorial n

| n == 0 = 1

| otherwise = n \* factorial (n - 1)

length [3,4,5] 3 length :: [a] -> Int

head [3,4,5] 3 head :: [a] -> a

tail [3,4,5] [4,5] tail :: [a] -> [a]

[3,4] ++ [10,20,30] [3,4,10,20,30]

[1..20]

sum [1..10]

product [1..5]

drop 3 [1..10] [4,5,6,7,8,9,10]

take 5 [1.0,1.2..2] [1.0,1.2,1.4,1.6,1.8]

halves :: [a] -> [[a]]

halves lst = [first, second]

where

n = length lst `div` 2

first = take n lst

second = drop n lst

[10,20..100] !! 3 40

fromTo first last = [first..last]

fromTo :: Enum a => a -> a -> [a]

fromTo first last

| first > last = []

| otherwise = first : fromTo (succ first) last

show 10 "10"

putStr :: String -> IO ()

printN n = putStr (printN' n)

printN' n

| n == 0 = ""

| otherwise = printN' (n-1) ++ show n ++ "\n"

charbox :: Int -> Int -> Char -> IO ()

charbox width height ch = putStr (unlines (replicate height (replicate width ch)))

sumElems list

| list == [] = 0

| otherwise = head list + sumElems (tail list)

sumElems [] = 0

sumElems (h:t) = h + sumElems t

sumElems list =

if list == [] then 0

else head list + sumElems (tail list)

duphead :: [a] -> [a]

duphead [] = []

duphead (x:xs) = x : x : xs

len :: [a] -> Int

len [] = 0

len (\_:xs) = 1 + len xs

odds :: [Int] -> [Int]

odds [] = []

odds (x:xs)

| odd x = x : odds xs

| otherwise = odds xs\

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

isElem \_ [] = False

isElem x (y:ys)

| x == y = True

| otherwise = isElem x ys

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

maxVal [x] = x

maxVal (x:xs) = max x (maxVal xs)

middle (\_, m, \_) = m

swap ('a',False) (False,'a')

zip ["one","two","three"] [10,20,30] [("one",10),("two",20),("three",30)]

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

elemPos x = go 0

where

go \_ [] = -1

go i (y:ys)

| x == y = i

| otherwise = go (i + 1) ys

halves lst = [take (length lst `div` 2) lst, drop (length lst `div` 2) lst]

halves :: [a] -> ([a], [a])

halves lst = (take n lst, drop n lst)

where n = length lst `div` 2

countEO [3,4,5] (1,2)

countEO [] = (0,0)  
countEO (x:xs)  
 | odd x = (evens, odds+1)  
 | otherwise = (evens+1, odds)  
 where  
 (evens, odds) = countEO xs

twice f x = f (f x)

map (5+) [1,2,3] [6,7,8]

travel :: [Char] -> [Char]

travel s

| totalDisp == (0,0) = "Got home"

| otherwise = "Got lost"

where

disps = map mapMove s

totalDisp = (sum (map fst disps), sum (map snd disps))

flip :: (a -> b -> c) -> b -> a -> c

flip f x y = f y x  
splits :: [a] -> [([a], [a])]

splits xs = zip (inits xs) (tails xs)

where

inits [] = [[]]

inits (y:ys) = [[]] ++ map (y:) (inits ys)

tails [] = [[]]

tails ys@(\_:ys') = ys : tails ys'

f :: Char -> [String]

f c = [replicate n c | n <- [1..]]

foldl (+) 0 [1..10] 55

keepOdds list = foldr f [] list  
 where  
 f elem acm  
 | odd elem = elem : acm  
 | otherwise = acm

vowelPositions s = reverse result  
 where (result, \_) =  
 foldl (\acm@(vows, pos) letter ->  
 if letter `elem` "aeiou" then ((letter,pos):vows,pos+1)

else (vows,pos+1))  
 ([],0) s