**Homework 2 Solutions**

This homework covers the following topics from class:

* Functional Programming, part 1
  + Functional Programming Intro, Functions, Operators, Precedence, Control Flow, Bindings (Let and Where), Tuples, Lists (part 1)
* Functional Programming, part 2
  + Lists (part 2), List Comprehensions, Pattern Matching

## HASK1: Simple Functions (2 min)

(2 min.) Write a Haskell function named largest that takes in 2 String arguments and returns the longer of the two. If they are the same length, return the first argument.  
  
Example:  
largest “cat” “banana” should return “banana”.  
largest “Carey” “rocks” should return “Carey”.  
  
**Solution:**

| **largest** :: **String** -> **String** -> **String** **largest** first second =  **if** length first >= length second **then** first **else** second |
| --- |

## HASK2: Recursion, Precedence, Guards (4 min)

(4 min.) Barry Snatchenberg is an aspiring Haskell programmer. He wrote a function named reflect that takes in an Integer and returns that same Integer, but he wrote it in a very funny way:

| **reflect** :: **Integer** -> **Integer** **reflect** 0 = 0 **reflect** num  | num < 0 = (-1) + reflect num+1  | num > 0 = 1 + reflect num-1 |
| --- |

He finds that when he runs his code, it always causes a stack overflow (infinite recursion) for any non-zero argument! What is wrong with Barry’s code (i.e. can you fix it so that it works properly)?

**Solution:**

**Barry is missing parentheses around num+1 and num-1. In its current state, the program is functionally equivalent to:**

| **reflect** :: **Integer** -> **Integer** **reflect** 0 = 0 **reflect** num  | num < 0 = (-1) + (reflect num) + 1  | num > 0 = 1 + (reflect num) - 1 |
| --- |

**which causes an infinite loop.**  
  
**To fix the program, we need only to add some parentheses to fix the associativity:**

| **reflect** :: **Integer** -> **Integer** **reflect** 0 = 0 **reflect** num  | num < 0 = (-1) + reflect (num+1)  | num > 0 = 1 + reflect (num-1) |
| --- |

## HASK3: Recursion, Guards (4 min)

(4 min.) Write a pair of Haskell functions named is\_odd and is\_even that each take in 1 Integer argument and return a Bool indicating whether the integer is odd or even respectively. You may assume that the argument is always positive.   
  
**You may not use any builtin arithmetic, bitwise or comparison operators (including mod, rem and div). You may only use the addition and subtraction operators (+ and -) and the equality operator (==).**

**You must implement THREE versions of these functions: (1) with regular if statements, (2) using** [**guards**](http://learnyouahaskell.com/syntax-in-functions#guards-guards)**, and (3) using** [**pattern matching**](http://learnyouahaskell.com/syntax-in-functions)**.**  
  
Example:  
is\_even 8 should return True.  
is\_odd 8 should return False.  
  
**Hint:** The functions can call one another in their implementations. (This is called [mutual recursion](https://en.wikipedia.org/wiki/Mutual_recursion)).  
  
**Solutions:**

| -- with if statements **is\_odd** :: **Integer** -> **Bool** **is\_odd** x =  **if** x == 0 **then** **False** **else** is\_even (x-1)  **is\_even** :: **Integer** -> **Bool** **is\_even** x =  **if** x == 0 **then** **True** **else** is\_odd (x-1) |
| --- |

| -- with guards **is\_odd** :: **Integer** -> **Bool** **is\_odd** x   | x == 0 = **False**  | otherwise = is\_even (x-1)  **is\_even** :: **Integer** -> **Bool** **is\_even** x  | x == 0 = **True**  | otherwise = is\_odd (x-1) |
| --- |

## HASK4: Where, Tuples (5 min)

(5 min.) Write a Haskell function called *quad* that finds the roots of a quadratic equation with coefficients a, b and c. Your solution must take three Double parameters and return a tuple containing the positive and negative roots. If the coefficient a is zero (resulting in division by zero) or the roots would be imaginary then the function must return the tuple (0, 0). **Your solution must have a type signature and use the *where* clause.**

Here's how it would be used:

\*Main> quad 1 (-5) 6

(3.0,2.0)

\*Main> quad 10 1 2

(0.0, 0.0)

\*Main> quad 1 10 0

(0.0, -10.0)

**Solution:**



| **quad** :: **Double** -> **Double** -> **Double** -> (**Double**, **Double**) **quad** a b c  | a == 0 = (0, 0) -- Avoid divide by zero  | discriminant < 0 = (0, 0) -- Imaginary roots case  | otherwise = (root1, root2)  **where**  discriminant = b^2 - 4 \* a \* c  x = sqrt(discriminant)  root1 = ((-b) + x) / (2 \* a)  root2 = ((-b) - x) / (2 \* a) |
| --- |

## HASK5: Recursion, Where, Helper Functions (5 min)

(5 min.) In this problem you're going to write a function called *sum\_is\_divisible* that determines whether the sum of integers in the range [a,b] inclusive is evenly divisible by a third integer c, and returns True if so and False otherwise. Your function may assume that a <= b. You must use recursion in a useful way. Here's how it might be used:

Main\*> *sum\_is\_divisible* 3 5 6

True

Main\*> *sum\_is\_divisible* 1 3 5

False

Your top-level function must have a type signature.

Hints:

1. You will need to use a nested function within a let or where clause to solve this problem!

2. Haskell's modulo function is called `mod`. The infix version looks like this: *6 `mod` 3*. The prefix version look like this: *mod 6 3*

**Solution:**

| **sum\_is\_divisible :: Int -> Int -> Int -> Bool sum\_is\_divisible a b c =  summer a b `mod` c == 0  where  summer a b  | a == b = a  | otherwise = a + (summer (a+1) b)** |
| --- |

## HASK6: Simple List Processing, Recursion (5 min)

(5 min.) In this problem we'll learn how to process items in a list, using Haskell's length, head and tail functions, and a little bit of recursion. Don't worry if we haven't covered Haskell lists in class yet… they're just like python's lists!

Here's how we define a simple list of integers in Haskell:

| **nums** = [3, -2, 1, 4] |
| --- |

Given this, you are to write a function, called *find\_min,* that finds the minimum value in a list of integers. Your function must have the following type signature:

| **find\_min** :: [Int] -> Int |
| --- |

which indicates that this function's first parameter is a *list of integers*, and it returns an *integer* result.

You will find the following Haskell functions useful:

**length** x: Returns the number of values in list x, so *length nums* would return 4  
**head** x: Returns the first value in the list x, so *head nums* would return 3

**tail** x: returns a new list containing all but the first value in a list x, so *tail nums* would return [-2, 1, 4]

Write the *find\_min* function.

**Solution:**

| **find\_min** :: [**Int**] -> **Int** **find\_min** lst  | length lst == 1 = head lst -- Base case: if there's only one element, return it  | otherwise = min first restMin -- Recursive case: compare head with the min of the tail  **where**  first = head lst -- The first element of the list  restMin = find\_min (tail lst) -- The minimum of the rest of the list |
| --- |

**Note: this is not the idiomatic way to process lists in Haskell. To do that we would need to use pattern matching, which we'll learn in our next lecture.**

## HASK7: List Comprehensions (5 min)

Part A: (2 min.) Write a Haskell function named all\_factors that takes in an Integer argument and returns a list containing, in ascending order, all factors of that integer. You may assume that the argument is always positive. **Your function’s implementation should be a single, one-line list comprehension.**Example:

x

all\_factors 1 should return [1].  
all\_factors 42 should return [1, 2, 3, 6, 7, 14, 21, 42].

**Solution:**

| **all\_factors** :: **Integer** -> [**Integer**] **all\_factors** num =  [x | x <- [1..num], num `mod` x == 0] |
| --- |

Part B: (3 min.) A [perfect number](https://www.britannica.com/science/perfect-number) is defined as a positive integer that is equal to the sum of its proper divisors (where “proper divisors” refers to all of its positive whole number factors, excluding itself). For example, 6 is a perfect number because its proper divisors are 1, 2 and 3 and 1 + 2 + 3 = 6.  
  
Using the all\_factors function, write a Haskell expression named perfect\_numbers whose value is a **list comprehension** that generates an infinite list of all perfect numbers (even though it has not been proved yet whether there are infinitely many perfect numbers 😉).

Example:  
take 4 perfect\_numbers should return [6, 28, 496, 8128].  
  
**Hint:** You may find the [init](https://hackage.haskell.org/package/base-4.17.0.0/docs/Data-List.html) and [sum](https://hackage.haskell.org/package/base-4.17.0.0/docs/Data-List.html) functions useful.

**Solution:**

| **perfect\_numbers** :: [**Integer**] **perfect\_numbers** =   [x | x <- [1..], sum (init (all\_factors x)) == x] |
| --- |

## HASK8: Recursion, Pattern Matching (10 min)

(10 min.) Write a function named count\_occurrences that returns the number of ways that all elements of list a1 appear in list a2 in the same order (though a1’s items need not necessarily be consecutive in a2). The empty sequence appears in another sequence of length n in 1 way, even if n is 0. Make sure to use pattern matching in your solution.

Examples:

count\_occurrences [10, 20, 40] [10, 50, 40, 20, 50, 40, 30] should return 1.

count\_occurrences [10, 40, 30] [10, 50, 40, 20, 50, 40, 30] should return 2.

count\_occurrences [20, 10, 40] [10, 50, 40, 20, 50, 40, 30] should return 0.

count\_occurrences [50, 40, 30] [10, 50, 40, 20, 50, 40, 30] should return 3.

count\_occurrences [] [10, 50, 40, 20, 50, 40, 30]   
should return 1.

count\_occurrences [] [] should return 1.

count\_occurrences [5] [] should return 0.

**Solution:**

| **count\_occurrences** :: [**Integer**] -> [**Integer**] -> **Integer count\_occurrences** [] \_ = 1 **count\_occurrences** \_ [] = 0 **count\_occurrences** (x:xs) (y:ys)| x == y = count\_occurrences xs ys + other\_occurrences  | otherwise = other\_occurrences **where** other\_occurrences = count\_occurrences (x:xs) ys |
| --- |

## HASK9: Recursion, Lists (20 min)

Part A: (10 min.) Write a Haskell function named fibonacci that takes in an Int argument n. It should return the first n numbers of the [Fibonacci sequence](https://en.wikipedia.org/wiki/Fibonacci_number) (for this problem, we’ll say that the first two numbers of the sequence are 1, 1).  
  
Examples:  
fibonacci 10 should return [1,1,2,3,5,8,13,21,34,55].  
fibonacci -1 should return [].  
  
**Hint:** You may find it easier to build the list in reverse in a right-to-left manner, then use the [reverse](https://hackage.haskell.org/package/base-4.17.0.0/docs/Data-List.html#v:reverse) function.  
  
**Solutions:**

| -- solution where we build the list in reverse **fibonacci** :: **Int** -> [**Integer**] **fibonacci** n =  **let** fib\_rev1 = [1] fib\_rev2 = [1, 1]fib\_revn =  **let** prev\_fib\_rev = fib\_rev (n-1)first = head prev\_fib\_rev second = head (tail prev\_fib\_rev) **in** (first + second) : prev\_fib\_rev  **in** reverse (fib\_rev n) |
| --- |

| -- less-efficient solution building the list forward  -- second\_last is O(n) **fibonacci** 1 = [1] **fibonacci** 2 = [1, 1] **fibonacci** n =  **let** second\_last xs  | length xs == 2 = head xs  | otherwise = second\_last (tail xs)  prev\_fib = fibonacci (n-1)  **in** prev\_fib ++ [last prev\_fib + second\_last prev\_fib] |
| --- |

| -- fancy solution using list comprehension that generates an  -- infinite list.  -- this works because of Haskell’s lazy evaluation. **fibonacci** n =  **let** fib = 1 : 1 : [a+b | (a,b) <- zip fib (tail fib)]  **in** take n fib |
| --- |

Part B: (10 min.) In this problem, you'll be writing a function in Haskell that run-length-encodes a List of integers. Run-length encoding (RLE) is a data compression technique that encodes consecutive repeated values in a sequence by replacing them with a single value and a count of its repetitions. For example, given the sequence 1, 1, 1, 5, 5, 3, 1, 1, RLE would compress it by generating a list of tuples of the form (value, count). The first three 1s are encoded as (1, 3), followed by the two 5s as (5, 2), and so on. This results in a more compact representation of the original data.

For example:

rle [1, 1, 1, 5, 5, 3, 1, 1] should return [(1, 3), (5, 2), (3, 1), (1, 2)].

rle [4, 4, 4, 4, 4] should return [(4, 5)].

rle [7, 8, 9]` should return [(7, 1), (8, 1), (9, 1)].

rle [] should return [].

Requirements:

1. The function must be implemented recursively and avoid using imported functions like `group` or `span`.

2. The function must handle cases where the input list is empty and return an empty list in such scenarios.

3. The function must correctly count and group consecutive occurrences only.

**Solution:**

| **rle :: [Int] -> [(Int, Int)] rle [] = [] rle (x:xs) = count x 1 xs  where  count current n [] = [(current, n)]  count current n (y:ys)  | current == y = count current (n + 1) ys  | otherwise = (current, n) : rle (y:ys)** |
| --- |