Problem Set

1 Programming with Lists

1. Which of the following expressions are true and which are false?

```
(a) [[]] ++ xs == xs
(b) [[]] ++ xs == [xs]
(c) [[]] ++ xs == [[], xs]
(d) [[]] ++ [xs] == [[],xs]
(e) [xs] ++ [] == [xs]
(f) [xs] ++ [xs] == [xs, xs]
```

2. A *well balanced parantheses string* (WBPS) is defined by the grammar

$$S \to (S)\ S \mid \epsilon$$

As examples, the strings (), (()) and (())() are WBPS and the strings)(and (() are not.

A string which is not WBPS can be converted to a WBPS by removing some characters from it. e.g. ((() gets converted to (). Of course, by removing all characters from ((() we can also get a WBPS,

Your job is to define a function `errno`, which when given a parenthesised string, would return minimum number of characters, removal of which would give a WBPS. As examples, `errno ")("` returns `2`, `errno ())()))"` returns `3`, and `errno (())()` is, of course `0`.

3. Using list comprehension, write a Haskell function `powers` which when given an argument `x` returns a list containing the powers of `x`.

example: `powers 2 = [1,2,4,8,16,...]`

Note that any recursion must only involve `powers`.

4. (a) In terms of the lengths of the input lists, what is the work done by the function `(++)` shown below?

```
[] ++ l = l
(x:xs) ++ l = x : (xs ++ l)
```

(b) What is the work done to reverse a list using the following definition?

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

(c) Rewrite `reverse` so that it becomes efficient by more than a constant factor.

5. Consider the function `hof` (higher order function) given by:

```
hof a b c [] = b
hof a b c (x:xs) = c ( a x ) (hof a b c xs)
```

Use this function to define the functions:

- `map`
- `foldr`
- `reverse`
- `takewhile`

You may use auxilliary functions if you wish, but define them before you do so. To judge whether you have used `hof` in the right sense, ensure that all recursion is through `hof` only.

6. Write a function `lcs :: (Eq a) => [a] -> [a] -> [a]` in Haskell to find a longest common subsequence of two lists. For example the following call to `lcs`:

   `lcs ['A','B','C','B','D','A'] ['B','D','C','A','B','A']`
   returns `['B','C','B','A']`.

7. Write a Haskell program to find all Hamiltonian cycles in a given graph represented as a list of edges, using the suggestion outlined below.

   Assume the following type synonyms.

```
type Graph = [(Int,Int)]
type Node = Int
type Path = [Int]
```

   Let `makepath :: Node -> Graph -> [Path]` be a function which returns all simple paths (paths not containing cycles) starting from a node in a graph. Define `makepath`.

   Let `is_hc :: Path -> Graph -> Boolean` be a function which takes a simple path and a graph, and returns true if the path can be completed to a Hamiltonian cycle in the graph. Define `is_hc`.

   Now define `hcycles :: Graph -> [Path]`, which returns all Hamiltonian cycles in a given graph.

8. The function `cprod` takes a list of lists and computes its Cartesian product:

```
cprod [[1, 2, 3], [4], [5, 6]] =
[[1, 4, 5], [1, 4, 6], [2, 4, 5], [2, 4, 6], [3, 4, 5], [3, 4, 6]]
```

9. Consider a 2-D map with a horizontal river passing through its center. There are n cities on the southern bank in the order $[a_1, \ldots, a_n]$ and similarly there are n cities on the northern bank in the order $[b_1, \ldots, b_n]$. You want to connect as many north-south pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, you can only connect city a_i to b_j, if $a_i = b_j$.

   As an example, if southern bank cities are in the order [2,3,1,6,4,5,7] and northern bank cities are in the order [6,4,2,7,1,3,5], then one of the solutions is to connect cities [2,3,5]. A 4-bridge solution does not exist. Your function should be called `connect` and should return the list of cities which should be connected.

10. Pascal's triangle is a sequence of rows of numbers that go like this:

```
        1
      1   1
    1   2   1
  1   3   3   1
_  _   _   _   _
```

An entry in a row is obtained by adding the numbers to its left and right in the previous row, padding with zeroes if necessary.

Suppose that Pascal's triangle were represented as a list of lists. Using list comprehension, give a recursive definition of Pascal's triangle.

`pascal = _________pascal__________`

11. Here is a puzzle that I am sure most of you are aware of: Two numbers `a` and `b` in the range `[2,99]` are picked. S is given their sum `a+b` and P their product `a*b`. They do not know each others numbers. The following dialog takes place:

    P: I don't know the numbers
    S: I knew you didn't know. I don't know either.
    P: Now I know the numbers.
    S: Now I know them too.

Let us compute the numbers `a` and `b` making use of the conversation. Make heavy use of list comprehension.

```
good_nums = [2..99]::[Int]
```

First define a function to find all the good co-factors of a number `p`. Include only one of `(x,y)` and `(y,x)`.

```
good_factors p = [(a,b) | a<-good_nums, b<-good_nums, ...]
```

Similarly define a function `good_summands`. Also define a function called `singleton`, which returns `True` if its argument is a singleton list.

Now let's encode the first fact: P doesn't know the pair of numbers `(a,b)`. P would have known the numbers if the product `a*b` had a unique good factorization.

```
fact1 (a,b) = ...
```

Encode the second fact: S doesn't know the numbers

```
fact2 (a,b) = ...
```

Encode the third fact: S knows that P doesn't know the numbers. In other words, for all possible summands that make `a+b`, P cannot be certain of the numbers

`fact3 (a,b) = ...` Encode the fourth fact: P *now* knows that `fact3` is true and is able to find the numbers. That is, of all factorizations of `a*b`, there exists only one that makes the third fact true.

```
fact4 (a,b) = ...
```

The fifth fact is that S *now* knows what the numbers are. Figure out why this happens and express this in the form of `fact5`.

```
fact5 (a,b) = ...
```

All that remains is to compute the list of all numbers such that `fact1` ... `fact5` hold:

```
result = [(a,b) | a<-good_nums, b<-good_nums,  ... ]
```

12. Given a list `xs = [`x_0`, `x_1`, ..., `x_n`]` of numbers, the sequence of successive maxima, `ssm xs`, is the longest subsequence `[`x_{j_0}`, `x_{j_1}`, ...`x_{j_m}`]` such that $j_0 = 0$ and $x_{j_i} < x_{j_k}$ for $j_i < j_k$. For example, the sequence of successive maxima of `[3,1,3,4,9,2,10,7]` is `[3,4,9,10]`. Define `ssm`.

13. Imagine the following game: You are given a path that consists of white and black squares. You start on the leftmost square (which we'll call square 1), and your goal is to move off the right end of the path in the least number of moves. However, the rules stipulate that:

    - If you are on a white square, you can move either 1 or 2 squares to the right.
    - If you are on a black square, you can move either 1 or 4 squares to the right.

    Write a function `fewest_moves :: [Int] -> Int` that takes a path representated as a list of 0's and 1's (1 is white and 0 is black) and computes the minimum number of moves. As an example:

    `fewest-moves [1,0,1,1,1,0,1,1,0,1,0,1,1,0,0,1,1,1]` returns 6, and is obtained by stepping on the squares at positions 1, 2, 6, 10, 11 and 15.

14. Write a function called `summands` in Haskell which takes as input a positive integer n, and produces a list containing all ways of writing n as a sum of positive integers. Example:

```
summands 1  => [[1]]
summands 2  => [[1,1],[2]]
summands 3  => [[1,1,1],[1,2],[2,1],[3]]
summands 4  => [[1,1,1,1],[1,1,2],[1,2,1],[1,3],[2,1,1],[2,2],
                [3,1],[4]]
summands 5  => [[1,1,1,1,1],[1,1,1,2],[1,1,2,1],[1,1,3],[1,2,1,1],
                [1,2,2],[1,3,1],[1,4],[2,1,1,1],[2,1,2],[2,2,1],
                [2,3],[3,1,1],[3,2],[4,1],[5]]
```

15. All of you probably know the crypt-arithmetic puzzle:

```
   s e n d
+  m o r e
----------
 m o n e y
```

    Here each of the alphabets above stand for a distinct digit such that the resulting addition is correct. Write a Haskell expression using list comprehension which will solve the puzzle and give the following output:

    `[('s',9),('e',5),('n',6),('d',7),('m',1),('o',0),('r',8),('y',2)]`.

2 Algebra of Programs

16. Consider the following laws:

Law 1 $(foldr\ (\oplus)\ a) \circ (map\ f) = foldr\ (\otimes)\ a$ where $x \otimes y = f\ x \oplus y$

Law 2 $foldl\ (\oplus)\ a\ xs = foldr\ (\ominus)\ a\ (reverse\ xs)$, where $x \oplus y = y \ominus x$

Law 3 $(map\ f) \circ reverse = reverse \circ (map\ f)$

Here $\oplus$, $\ominus$ and $\otimes$ are arbitrary operators. From these laws can you prove a law for $foldl$ which says:

Law 4 $(foldl\ (\oplus)\ a) \circ (map\ f) = foldl\ (\otimes)\ a$ where $x \otimes y = x \oplus f\ y$

You might find pictures helpful in guiding your proof.

17. Using the universal law for foldr, give proof of:

$$sum\ x + sum\ y = foldr\ (+)\ (sum\ x)\ y$$

18. Prove the following identities using fusion law. As you manipulate one expression to another, mention the rule that you use.

    (a) `map f (x ++ y) = map f x ++ map f y`

    (b) `map f.  map g = map (f.g)`

    (c) `map f.concat = concat.  map(map f)`

19. Write definitions for the following functions which will execute in O(n) time.

    (a) **foldr1 -** It takes the last two items of the list and applies the function, then it takes the third item from the end and the result, and so on. See scanr1 for intermediate results
    Input: foldr1 (+) [1,2,3,4]
    Output: 10

    (b) **scanl -** It takes the second argument and the first item of the list and applies the function to them, then feeds the function with this result and the second argument and so on. It returns the list of intermediate and final results.
    Input: scanl (/) 64 [4,2,4]
    Output: [64.0,16.0,8.0,2.0]

    (c) **scanr -** It takes the second argument and the last item of the list and applies the function, then it takes the penultimate item from the end and the result, and so on. It returns the list of intermediate and final results.
    Input: scanr (+) 5 [1,2,3,4]
    Output: [15,14,12,9,5]

20. Using the properties of fold, prove the following laws

    (a) **Bookkeeping Law -** `foldr f a .  concat = foldr f a .  map (foldr f a)`

    (b) **Generalized Horner's rule**