CS 357: Declarative Programming
Homework 5

1. Define a function *myTakeWhile* which takes a predicate and a list as arguments and returns the prefix of the list satisfying the predicate. For example,

```
*Main> myTakeWhile (/= ' ') "This is practice."
"This"
```

2. Define a function *mySpan* which takes a predicate and a list as arguments and returns a pair of lists where the first element of the pair is the portion of the list which the function *myTakeWhile* would return and the second element is the remainder of the list. For example,

```
*Main> mySpan (/= ' ') "This is practice."
("This"," is practice.")
```

3. The function *combinations3* takes a list as its argument and returns a list of length three lists representing all possible subsets of size three. For example,

```
*Main> :t combinations3
:t combinations3
combinations3 :: (Ord a) => [a] -> [[a]]
*Main>   combinations3 "ABCDE"
combinations3 "ABCDE"
["ABC","ABD","ABE","ACD","ACE","ADE","BCD","BCE","BDE","CDE"]
```

Write *combinations3* using a list-comprehension. You may assume that the input list contains no duplicates.

4. The function *runLengthEncode* takes a list of values as its argument and returns a list of pairs of values and run lengths. See

```
http://en.wikipedia.org/wiki/Run_length_encoding
```

For example,

```
*Main> runLengthEncode [4,2,2,1,1,1,1,4,4,4,4]
[(4,1),(2,2),(1,4),(4,4)]
*Main> runLengthEncode "foo"
[('f',1),('o',2)]
```

Write *runLengthEncode*. Hint: Divide and conquer. Ask yourself: What helper functions would make this problem trivial and then write those. Make use of higher-order functions when appropriate. This is the key to modular design and you will complete your homework faster as a bonus.

5. The function *runLengthDecode* takes a list of pairs of values and run lengths and returns a list of values. For example,

```
*Main> runLengthDecode [(4,1),(2,2),(1,4),(4,4)]
[4,2,2,1,1,1,1,4,4,4,4]
```

   Write *runLengthDecode.*

6. Define a function *splitText* which takes a string of text and a predicate and returns a list of subtstrings comprised of contiguous characters for which the predicate is satisfied. For example,

```
*Main> splitText (/= ' ') "This is practice."
["This","is","practice."]
```

7. Use a list comprehension to define a function *encipher* which takes two lists of equal length and a third list. It uses the first two lists to define a substitution cipher which it uses to encipher the third list. For example,

```
*Main> encipher ['A'..'Z'] ['a'..'z'] "THIS"
"this"
```

8. The Goldbach conjecture states that any even number greater than two can be written as the sum of two prime numbers. Using list comprehensions, write a function *goldbach*, which when given an even number *n*, returns a list of all pairs of primes which sum to *n*. Note: You will have to write a function which tests an integer for primality and this should be written as a list comprehension also. For example,

```
*Main> goldbach 6
[(3,3)]
*Main> :t goldbach
Int -> [(Int,Int)]
```

9. The function *increasing* takes a list of enumerable elements as its argument and returns *True* if the list is sorted in increasing order and *False* otherwise.

```
*Main> increasing "ABBD"
True
*Main> increasing [100,99..1]
False
```

   Write *increasing* using the function *and* and a list comprehension.

10. The function *select* takes a predicate and two lists as arguments and returns a list composed of elements from the second list in those positions where the predicate, when applied to the element in the corresponding positions of the first list, returns *True*.

```
*Main> :t select
select :: (t -> Bool) -> [t] -> [a] -> [a]
*Main> select even [1..26]  "abcdefghijklmnopqrstuvwxyz"
"bdfhjlnprtvxz"
*Main> select  (<= 'g') "abcdefghijklmnopqrstuvwxyz" [1..26]
[1,2,3,4,5,6,7]
```

Write *select* using list comprehensions.

11. The function *combinations* takes an integer k and a list of elements of typeclass *Ord* as its arguments and returns a list of length k lists representing all possible subsets of size k. For example,

```
*Main> :t combinations
combinations :: (Ord a) => Int -> [a] -> [[a]]
*Main>   combinations 3 "ABCDE"
["ABC","ABD","ABE","ACD","ACE","ADE","BCD","BCE","BDE","CDE"]
```

Write *combinations*. Hint: Don't use list-comprehensions. Do use *increasing*. Write *combinations1*. Use *combinations1* and *map* to write *combinations2*. Now use *combinations2* and *map* to write *combinations3*. Abstract the pattern.

12. Addition and multiplication of complex numbers are defined as follows:

$$\begin{aligned}(x+iy)+(u+iv) &= (x+u)+(y+v)i \\ (x+iy)\times(u+iv) &= (xu-yv)+(xv+yu)i\end{aligned}$$

A *complex integer* is a complex number with integer real and imaginary parts. Define a data type for complex integers called *ComplexInteger* with selector functions *real* and *imaginary* which return the real and imaginary parts. Give minimum instance declarations for the *Eq*, *Show*, and *Num* type classes. For example,

```
*Main> real (ComplexInteger 1 2)
1
*Main> imaginary (ComplexInteger 2 3)
3
*Main> (ComplexInteger 1 2) == (ComplexInteger 3 4)
False
*Main> (ComplexInteger 1 2)
1+2i
*Main> (ComplexInteger 1 2) * (ComplexInteger 3 4)
-5+10i
```