THE UNIVERSITY OF BRITISH COLUMBIA
CPSC 312: MIDTERM EXAM
OCTOBER 8, 2009

Instructor: Kurt Eiselt

Name: _______________________ Student #: _________________

Signature: _____________________

Notes about this examination

1. You have 80 minutes to write this examination. When you have completed the exam, please bring your exam to your instructor.
2. No notes, books, or any type of electronic equipment is allowed including cell phones, laptop computers, and calculators.
3. Good luck!

Question	Mark	Max
1		10
2		5
3		5
4		5
5		10
6		10
Total		**45**

Important Exam Notes: For any of the following questions that asks you to write, create, or construct a function to perform some task, you might want to write additional functions to help. That's not only acceptable, it's encouraged. You don't have to ask for permission to create additional functions to answer the question.

Question 1: [10 marks]

A geometric progression is one in which there is a constant ratio between each element and the one preceding it. The sum of the first n elements of a geometric series is defined as:

$S = ar^0 + ar^1 + ar^2 + ... \_ ar^{n-1}$

Where S is the sum, a is the first element, r is the ratio, and n is the number of elements. For example, if a = 7, r = 2, and n = 5, we get:

$$\begin{aligned} S &= 7*2^0 + 7*2^1 + 7*2^2 + 7*2^3 + 7*2^4 \\ &= 7*1 + 7*2 + 7*4 + 7*8 + 7*16 \\ &= 7 + 14 + 28 + 56 + 112 \\ &= 217 \end{aligned}$$

Using Haskell, the information provided above, and the `**` function for raising some number to a power (described below), use the next page to construct two versions of the function `geometric` which takes three arguments corresponding to a, r, and n above, and returns the sum of the first n terms of the geometric series. To keep things simple, you may assume that a and r and n are always integers greater than 0. Here are some examples:

```
> geometric 7 2 5
217.0
> geometric 2 7 5
5602.0
> geometric 1 1 1
1.0
> geometric 1 1 7
7.0
> 2 ** 5
32.0
> 5 ** 2
25.0
> 5 ** 0
1.0
```

Question 1 continued:

In the space below, write two versions of the function described previously. The first version should not be tail recursive. Call this version `geometric`. The second version should be tail recursive. Call this version `geometric_tr`.

Question 2: [5 marks]

You may not believe it, but some students aren't really expending much effort at using meaningful, helpful names, comments, or abstraction when constructing the Haskell functions that we have to mark. Now it's time for us to turn the tables. Take a good look at the following four functions:

```
lista list1 = listb list1 [] [] []

listb list1 list2 list3 list4
  | null list1 = list2++list3++list4
  | otherwise  = listc (tail list1) (list2++((head list1):[])) list3 list4

listc list1 list2 list3 list4
  | null list1 = list2++list3++list4
  | otherwise  = listd (tail list1) list2 (list3++((head list1):[])) list4

listd list1 list2 list3 list4
  | null list1 = list2++list3++list4
  | otherwise  = listb (tail list1) list2 list3 (list4++((head list1):[]))
```

Now tell us what will be returned when the following call to `lista` is evaluated (and think of what your marker goes through when you turn in code that looks like this):

```
Hugs> lista "sglphiaovcseets"
```

Write your answer here:

Question 3: [5 marks]

The following two operations give exactly the same result:

```
Hugs> 1:[1,1,1,1]
[1,1,1,1,1]
Hugs> [1,1,1,1]++(1:[])
[1,1,1,1,1]
```

Which of these two operations is less computationally expensive? Explain why.

Question 4: [5 marks]

Below is part of the state space for a ripping good game of hexapawn. At the top is the current state of the game, which shows that white has made an opening move by pushing a pawn forward to the center of the board. The next level below that shows all three of black's possible responses, and below that you see the moves that white could make in response:

```
                                            W W -
                                            - - W     current state (after white's opening move)
                                            B B B
                                           /  |  \
                                      /       |       \
                                 /            |            \
                            /                 |                 \
                       /                      |                      \
                  /                           |                           \
             /                                |                                \
        W W -                               W W -                               W W -   black's three
        B - W                               - B W                               - - B   possible moves
        - B B                               B - B                               B - B
        / | \                               /   \                               / | \
      /   |   \                           /       \                           /   |   \
    /     |     \                       /           \                       /     |     \
  /       |       \                   /               \                   /       |       \
W - -   W - -   W W -               - W -           - W -               - W -   W - -   W - -   white's
W - W   B W W   B - -               W B W           - W W               W - B   - W B   - - W   possible
- B B   - B B   - W B               B - B           B - B               B - B   B - B   B - B   replies



_____   _____   _____               _____           _____               _____   _____   _____
```

Pretend you're a minimaxing, hexapawn-playing Haskell program that is controlling the black pawns, and your static board evaluation function is as follows:

```
The function returns a +10 if the board is such that black wins.  It returns
a -10 if white wins.  If neither side has won, the function returns the
result of counting the number of black pawns on the board and subtracting the
number of white pawns.
```

Part a: Apply your static evaluation function to each of the bottom-level boards and write the value that your function would assign to each board below that board in the spaces provided in that diagram up there. (There are 8 values to compute.)

Part b: On that same diagram, show which values would be propagated upward by your minimax component. Then clearly indicate which move should be made by black, according to the minimax strategy.

Question 5: [10 marks]

One of the questions in the second homework assignment asked you to write the function `myreplaceall` with the following behaviour:

```
myreplaceall 3 7 [7,0,7,1,7,2,7] => [3,0,3,1,3,2,3]
myreplaceall 'x' 'a' "" => ""
myreplaceall 'x' 'a' "abacad" => "xbxcxd"
```

It was pointed out that this description was incomplete. The description did not say, for example what was to happen in this case:

```
myreplaceall 3 7 [7,3,7,1,7,3,7]
```

Should the result be `[3,3,3,1,3,3,3]` or should it be `[3,7,3,1,3,7,3]?` Our response was that the first result is the correct result. That is, the first parameter replaces all occurrences of the second parameter in the list. But now let's say that the second result is the correct result. Not only does the first parameter replace all occurrences of the second parameter, but the second parameter replaces all occurrences of the first parameter. In the space below and/or on the next page, write this new version of `myreplaceall`. Actually, write it twice -- once without pattern matching, and once with pattern matching. Call the pattern matching version `myreplaceall_pm`.

Question 6: [10 marks]

You will now use the awesome power of Haskell to construct a program to decode a secret coded message. The program expects two parameters: a secret message given as a string of alphanumeric characters, and of course the key that allows your program to decode the message. Your program will then return the decoded message as a string. (Crypto freaks: yes, it's really a cipher, not a code. Forgive me.)

The key or replacement cipher is a list of two-element tuples that can be formed using the `zip` function:

```
key    = zip ['a','b' .. 'z'] ['z','y' .. 'a']
```

and in case you don't remember the `zip` function, the expression above is the same as:

```
key    = [('a','z'),('b','y'),('c','x'),('d','w'),('e','v'),
          ('f','u'),('g','t'),('h','s'),('i','r'),('j','q'),
          ('k','p'),('l','o'),('m','n'),('n','m'),('o','l'),
          ('p','k'),('q','j'),('r','i'),('s','h'),('t','g'),
          ('u','f'),('v','e'),('w','d'),('x','c'),('y','b'),
          ('z','a')]
```

Your program will consist of the following two functions (plus any helper functions that you need). The specifications for the functions are below. Write your functions on the following page.

6a) [5] Write a Haskell function called `substituteChar` that takes two arguments. The first argument is a list of character substitutions, where each character substitution is a tuple as shown in the `key` example above. The second argument is a single character. Given a list of substitutions and a single character as arguments, your `substituteChar` function should find the corresponding substitution character among the tuples and return that substitution character. In other words, the list of tuples named `key` above can be interpreted like this: "If given an 'a', return a 'z'; if given a 'b', return a 'y', and so on...". Your function should return the character '?' if it can't find a substitute for the character passed as the second argument. Here are some examples:

```
Main> substituteChar key 'a'
'z'
Main> substituteChar key 'z'
'a'
Main> substituteChar key '$'
'?'
```

6b) [5] Let's assume that you now have a working version of the `substituteChar` function. Now use `substituteChar` and recursion to write a function named `crypto1` that expects two arguments, a list of tuples representing character substitutions and a string representing an encoded message, and returns a decoded message.

```
Main> crypto1 key "rolevszhpvoo"
"ilovehaskell"
Main> crypto1 key "kovzhvwlmlguzronv"
"pleasedonotfailme"
```