Spring, 2013 Name: _____

(Please *don't* write your id number!)

COP 4020 — Programming Languages I

Makeup Test on Higher-Order Functional Programming in Erlang

Special Directions for this Test

This test has 6 questions and pages numbered 1 through 7.
This test is open book and notes, but no electronics.
If you need more space, use the back of a page. Note when you do that on the front.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
Clarity is important; if your programs are sloppy and hard to read, you may lose some points. Correct syntax also makes a difference for programming questions. We will take some points off for duplicated code, code with extra unnecessary cases, or code that is excessively hard to follow.
You will lose points if you do not "follow the grammar" when writing programs. You should always assume that the inputs given will follow the grammar for the types specified, and so your code should not have extra cases for inputs that do not follow the grammar.
When you write Erlang code on this test, you may use anything that is built-in to Erlang and the module `lists`.
You are encouraged to define functions not specifically asked for if they are useful to your programming; however, if they are not built-in to Erlang/OTP, then you must write them into your test. (Note that you can use built-in functions such as `lists:map/2`, `lists:foldr/3`, `lists:filter/2`, `lists:member/2`, etc.)

Hints

If you use functions like `lists:filter`, `lists:map`, and `lists:foldr` whenever possible, then you will have to write less code on the test, which will mean fewer chances for making mistakes and will leave you more time to be careful.

For Grading

Question:	1	2	3	4	5	6	Total
Points:	10	10	10	10	40	20	100
Score:							

1. [Concepts] This is a question about free and bound variable identifiers. Consider the following Erlang expression.

```
fun(Q,R,S) -> if Q -> fun(M) -> G(M+X) end;
                 true -> fun() -> R end
                 end
end
```

   (a) (5 points) Write, in set brackets ({ and }), the entire set of variable identifiers that occur free in the above Erlang expression.

   (b) (5 points) Write, in set brackets ({ and }), the entire set of variable identifiers that occur bound in the above Erlang expression.

2. (10 points) [UseModels] In Erlang, write a function `censor/2`, whose type is given by the following.

```
-spec censor([[atom()]], [atom()]) -> [[atom()]].
```

   The function `censor` takes a list of lists of atoms, `Document`, and a list of atoms, `BadWords`, and returns a list of lists of atoms that is just like `Document` except that the result does not contain any words that are in `BadWords`. The following are examples written using the homework's testing module.

```
-module(censor_tests).
-import(censor,[censor/2]).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0]).
main() -> dotests("censor_tests $Revision: 1.1 $", tests()).
tests() ->
    [eqTest(censor([],[sword, fword]), "==", []),
     eqTest(censor([[fword, it]], [sword, fword]), "==", [[it]]),
     eqTest(censor([[this, is, sword, see], [fword, it]], [sword, fword]),
            "==", [[this, is, see], [it]]),
     eqTest(censor([[inceptis, grauibus, plerumque, et, magna, professis],
                    [purpureus, late, qui, splendeat], [unus, et, alter],
                    [adsuitur, pannus, cum, lucus, et, ara, dianae]],
                   [purpureus, et, inceptis, dianae, unus, alter]),
            "==", [[grauibus,plerumque,magna,professis],
                   [late,qui,splendeat], [], [adsuitur,pannus,cum,lucus,ara]]),
     eqTest(censor([['@#*!+', '@#*!+', '@#*!+'], ['*^$@!', '*^$@!'],
                    [flowers, birds, trees], ['@#*!+', '*^$@!']],
                   ['@#*!+', '*^$@!']),
            "==", [[], [], [flowers, birds, trees], []])  ].
```

3. (10 points) [Concepts] [UseModels] In Erlang, write a function `compose/2`, whose type is given by the following.

```
-spec compose(fun((B) -> C), fun((A) -> B)) -> fun((A) -> C).
```

This function takes two functions as arguments and returns a function that is their composition (in the usual mathematical order). The following are examples written using the testing module of the homework.

```
% $Id: compose_tests.erl,v 1.1 2013/04/21 16:12:50 leavens Exp leavens $
-module(compose_tests).
-import(compose,[compose/2]).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0]).
main() -> dotests("compose_tests $Revision: 1.1 $", tests()).
tests() ->
    [eqTest((compose(fun(X) -> X+1 end, fun(Y) -> Y*3 end))(7),"==",22),
     eqTest((compose(fun(X) -> X==21 end, fun(Y) -> Y*3 end))(7),"==",true),
     eqTest((compose(fun(Y) -> Y*3 end, fun(X) -> X+1 end))(7),"==",24),
     eqTest((compose(fun(B) -> not B end, fun(Ls) -> Ls == [] end))([]),"==",false),
     eqTest((compose(fun(N) -> {N*2} end, fun(N) -> N+5 end))(6),"==",{22}),
     eqTest((compose(fun(X) -> {X,2} end, fun(Y) -> [Y,1] end))(6),"==",{[6,1],2})
    ].
```

4. (10 points) [UseModels] In Erlang, using `lists:foldr`, write the function `graph/2`, whose type is given by the following.

```
-spec graph(fun((A) -> B), [A]) -> [{A,B}].
```

This function takes a function and a list and returns a list of pairs. In the list of pairs that is the result, the first element of each pair is an element of the argument list, and the second element of the pair is the result of applying the function argument to that element. The following are tests.

```
% $Id: graph_tests.erl,v 1.1 2013/04/21 16:12:50 leavens Exp leavens $
-module(graph_tests).
-import(graph,[graph/2]).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0]).
main() -> dotests("graph_tests $Revision: 1.1 $", tests()).
tests() ->
    [eqTest(graph(fun(X) -> X*42 end, []),"==",[]),
     eqTest(graph(fun(X) -> X+1 end, [1,2,3]),"==",[{1,2}, {2,3}, {3,4}]),
     eqTest(graph(fun(X) -> (3*X)+1 end, [7,9,10,27]),
            "==", [{7,22}, {9,28}, {10,31}, {27,82}]),
     eqTest(graph(fun length/1, [[], "abc", "Erlang"]),
            "==", [{[],0}, {"abc", 3}, {"Erlang", 6}])
    ].
```

Your code must be written using `lists:foldr`, and without using any list comprehension or explicit recursion. So your solution must just pass arguments to `lists:foldr`. Do this by filling in the arguments to `lists:foldr` below.

```
graph(F, Xs) ->
    lists:foldr(
```

5. (40 points) [Concepts] [UseModels] In this problem you will implement an ADT of infinite sequences, in Erlang. You will represent sequences as pairs consisting of an atom, `seq`, and a function from non-negative-integers to floats.

```
-type sequence() :: {seq, fun((non_neg_integer()) -> float())}.
```

Using this type, your task is to implement the following four functions.

```
-spec fromRule(fun((non_neg_integer()) -> float())) -> sequence().
-spec add(sequence(), sequence()) -> sequence().
-spec scaleBy(sequence(), float()) -> sequence().
-spec nth(sequence(), non_neg_integer()) -> float().
```

These functions behave as described below:

- `fromRule(F)` returns a sequence whose `N`th element is `F(N)`.
- `add(S1, S2)` returns a sequence whose `N`th element is the sum of `N`th element of `S1` and the `N`th element of `S2(N)`.
- `scaleBy(S, X)` returns a sequence whose `N`th element is `X` times the `N`th element of `S`.
- `nth(S, N)` returns the `N`th element of the sequence `S`.

The following are examples, written using the `testing` module from the homework.

```
% $Id: sequence_tests.erl,v 1.1 2013/04/21 16:12:50 leavens Exp leavens $
-module(sequence_tests).
-import(sequence,[fromRule/1, add/2, scaleBy/2, nth/2]).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0]).
main() -> dotests("sequence_tests $Revision: 1.1 $", tests()).
tests() ->
    Ones = fromRule(fun(_N) -> 1.0 end),
    Squares = fromRule(fun(N) -> N*N end),
    Sp1 = add(Ones, Squares),
    [eqTest(nth(Ones, 0), "==", 1.0),
     eqTest(nth(Ones, 63323), "==", 1.0),
     eqTest(nth(Ones, 999999999999999999999999999999999999999), "==", 1.0),
     eqTest(nth(Squares, 0), "==", 0.0),
     eqTest(nth(Squares, 1), "==", 1.0),
     eqTest(nth(Squares, 2), "==", 4.0),
     eqTest(nth(Squares, 10), "==", 100.0),
     eqTest(nth(Squares, 1000), "==", 1000000.0),
     eqTest(nth(Sp1, 0), "==", 1.0),
     eqTest(nth(Sp1, 1), "==", 2.0),
     eqTest(nth(Sp1, 2), "==", 5.0),
     eqTest(nth(Sp1, 1000), "==", 1000001.0),
     eqTest(nth(scaleBy(Squares,3.0), 1), "==", 3.0),
     eqTest(nth(scaleBy(Squares,3.0), 5), "==", 75.0),
     eqTest(nth(scaleBy(Squares,3.0), 4020), "==", 4020.0*4020.0*3.0),
     eqTest(nth(add(fromRule(fun(N) -> 3.0*N*N + 2.0*N + 1.0 end),
                    fromRule(fun(N) -> 40.20*N*N*N + 6.0*N*N end)),
                10),
            "==", 41121.0),
     eqTest(nth(add(fromRule(fun(N) -> 3.0*N*N + 2.0*N + 1.0 end),
                    fromRule(fun(N) -> 40.20*N*N*N + 6.0*N*N end)),
                100),
            "==", 40290201.00000001) % some imprecision in this result
    ].
```

Fill in your answer by completing the `sequence` module on the next page.

For your answer, complete the following module.

```
-module(sequence).
-export([fromRule/1,add/2,scaleBy/2,nth/2]).
-export_type([sequence/0]).

% sequences are represented by tuples whose second element is a function
-type sequence() :: {seq, fun((non_neg_integer()) -> float())}.

% You are to implement the following functions
-spec fromRule(fun((non_neg_integer()) -> float())) -> sequence().
-spec add(sequence(), sequence()) -> sequence().
-spec scaleBy(sequence(), float()) -> sequence().
-spec nth(sequence(), non_neg_integer()) -> float().
```

6. (20 points) [Concepts] [UseModels] Your task in this problem is to write a functional abstraction of code that works on two lists. As examples to generalize from, consider the following. (Note: you are not supposed to write these functions, but generalize them.)

```
-spec sumtwo([number()], [number()]) -> number().
sumtwo([], _Bs) -> 0;
sumtwo(_As, []) -> 0;
sumtwo([A|As], [B|Bs]) -> A + B + sumtwo(As, Bs).
-spec prodtwo([number()], [number()]) -> number().
prodtwo([], _Bs) -> 1;
prodtwo(_As, []) -> 1;
prodtwo([A|As], [B|Bs]) -> A * B * prodtwo(As, Bs).
-spec zip([A], [B]) -> [{A,B}].
zip([], _Bs) -> [];
zip(_As, []) -> [];
zip([A|As], [B|Bs]) -> [{A,B} | zip(As, Bs)].
```

Your task in this problem is to write a functional abstraction of the above examples, which will be the function `twofold/4` that has the following type specification.

```
-spec twofold(fun((A,B,R) -> R), R, [A], [B]) -> R.
```

The function `twofold` takes a function `F`, a base case value `Z`, and two lists `As` and `Bs`. The function `F` is used in the recursive case. The base case value `Z` is used in the base cases.

Your task is to write `twofold` so that it can be used as in the examples below.

```
-module(twofold_tests).
-import(testing,[dotests/2,eqTest/3]).
-import(twofold,[twofold/4]).
-export([main/0]).
main() -> dotests("twofold_tests $Revision: 1.1 $", tests()).
% 3 functions defined just for testing purposes, not for you to implement
sumtwo(As, Bs) -> twofold(fun(A,B,R) -> A+B+R end, 0, As, Bs).
prodtwo(As, Bs) -> twofold(fun(A,B,R) -> A*B*R end, 1, As, Bs).
zip(As, Bs) -> twofold(fun(A,B,R) -> [{A,B}|R] end, [], As, Bs).
tests() ->
    [eqTest(sumtwo([],[]),"==",0),
     eqTest(sumtwo([5,7,10],[]),"==",0),
     eqTest(sumtwo([],[5,7,10,11]),"==",0),
     eqTest(sumtwo([1,2,3],[10,20,30,40]),"==",11+22+33),
     eqTest(prodtwo([1,2,3],[10,20,30,40]),"==",10*2*20*3*30),
     eqTest(zip([2.73,3.14],[]),"==",[]),
     eqTest(zip([],[1,2,3]),"==",[]),
     eqTest(zip([5,7,4],[1,2,3]),"==",[{5,1}, {7,2}, {4,3}]),
     eqTest(zip([a,b,c,d],[1,2,3,4]),"==",[{a,1}, {b,2}, {c,3}, {d,4}])  ].
```

Do *not* use any Erlang library functions or list comprehensions in your answer.