Spring, 2015 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 7 questions and pages numbered 1 through 10.
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`, unless the problem specifically prohibits it.
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 functions or from the `lists` module, 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`, `lists:reverse/1`, `lists:sublist/2`, `lists:sum/1`, etc.)

Hints

If you use functions like `lists:filter/2`, `lists:map/2`, and `lists:foldr/3` 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	7	Total
Points:	10	10	15	15	15	10	25	100
Score:								

1. (10 points) [Concepts] [UseModels] In Erlang, using `lists:foldr/3`, write a function `lengths/1`, whose type is given by the following.

```
-spec lengths(LL :: [[any()]]) -> [non_neg_integer()].
```

This function takes a list of lists, `LL`, as an argument and returns a list of the lengths of each of the sublists of `LL`, in order. The following are examples written using the testing module.

```
% $Id: lengths_tests.erl,v 1.1 2015/04/16 11:25:27 leavens Exp leavens $
-module(lengths_tests).
-import(lengths,[lengths/1]).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0]).
main() -> compile:file(lengths),
    dotests("lengths_tests $Revision: 1.1 $", tests()).
tests() ->
    [eqTest(lengths([]),"==",[]),
     eqTest(lengths([[]]),"==",[0]),
     eqTest(lengths([[4020]]),"==",[1]),
     eqTest(lengths([[], [7], [9,2], [5,1,5], [9,6,3,2]]),"==",[0,1,2,3,4]),
     eqTest(lengths([[9,8,7,6,4,5,3,2,1],[6,6,7,8,3]]),"==",[9,5]),
     eqTest(lengths([[2,4,6,8],[who,do,we,apprec,i,ate],[erlang]]),"==",[4,6,1]),
     eqTest(lengths([[today,all,'of',us,who,know],[go,to,the],
                     [store,for,shellfish],[its,good]]),"==",[6,3,3,2])    ].
```

Your solution must use `lists:foldr/3` in an essential way, so you must write it by filling in the remainder of the following. You must not use explicit recursion or a list comprehension in your solution. However, you may use helping functions and built-in functions.

```
lengths(LL) ->
    lists:foldr(
```

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

```
-spec count(What :: atom(), LOA :: [atom()]) -> pos_integer().
```

This function takes an atom, `What`, and a list of atoms, `LOA`, and returns the number of times that `What` occurs within `LOA`. The following are examples written using the testing module.

```
% $Id: count_tests.erl,v 1.1 2015/04/16 11:25:27 leavens Exp leavens $
-module(count_tests).
-import(count,[count/2]).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0]).
main() -> compile:file(count),
    dotests("count_tests $Revision: 1.1 $", tests()).
tests() ->
    [eqTest(count(maple, []),"==",0),
     eqTest(count(maple, [the,maple,tree,makes,maple,syrup]),"==",2),
     eqTest(count(a, [a,ah,the,ah,a,fine,thing,to,do,is,to,get,an,a,eh,a]),"==",4),
     eqTest(count(secret, [secret,house,secret,storm,secret,secret,secret]),"==",5),
     eqTest(count(ucf, [ucf,second,largest,hitt,pres,ucf,at,ucf,ok]),"==",3)  ].
```

Your solution must use `lists:foldr/3` in an essential way, so you must write it by filling in the remainder of the following. You must not use explicit recursion or a list comprehension in your solution. However, you may use helping functions and built-in functions.

```
count(What, LOA) ->
    lists:foldr(
```

Text Data Type Problems

The next few problems are work with the type `text()` and the type `formatted()` exported by the module `text` found in Figure 1 below.

```
-module(text).
-export_type([face/0, formatted/0, text/0]).
-export([fs/1, ts/1]).
% A face is the way a charater looks
-type face() :: roman | bold | italic.
% A formatted is a pair of a face and a string (= list of characters)
-type formatted() :: {face(), string()}.
% A text can be ...
-type text() :: {fs, [formatted()]}     % a list of formatted strings
              | {ts, [text()]}.         % or a list of texts.

% constructor functions that you can use if you wish
-spec fs([formatted()]) -> text().
fs(LS) -> {fs, LS}.
-spec ts([text()]) -> text().
ts(LS) -> {ts, LS}.
```

Figure 1: The module `text`, used in the next few problems.

The type `formatted` is intended to model formatted text, that can appear in a roman, **bold**, or *italic* typeface. The type `text` can either be a list of formatted strings or a list of texts.

It may be helpful to recall that a string in Erlang is a list of characters.

3. (15 points) [UseModels] This is a problem about the `text:text()` type in Figure 1 on the previous page. In Erlang, write a function, `chars`, whose type is given by the following:

```
-spec chars(Txt :: text:text()) -> string().
```

This function takes a text, `Txt`, as an input and returns the string of all the characters that appear in the strings of `Txt`, in the left to right order in which they appear in the argument. Be sure to follow the grammar given by the types in Figure 1 on the preceding page! The following are tests.

```
-module(chars_tests).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0, tests/1]).
main() -> compile:file(chars),
    dotests("chars_tests $Revision: 1.1 $", tests(fun chars:chars/1)).
-spec tests(CFun :: fun((text:text()) -> string()))
          -> testing:testCase([string()]).
tests(CFun) ->
    [eqTest(CFun({fs, []}), "==", ""),
     eqTest(CFun({ts, []}), "==", ""),
     eqTest(CFun({fs, [{bold,"are changin'"}]}), "==", "are changin'"),
     eqTest(CFun({fs, [{roman, "the times"},{bold," are changin'"}]}), "==",
            "the times are changin'"),
     eqTest(CFun({ts, [{fs, [{italic, "the times"}]},
                       {fs, [{bold, " try "}, {italic, "our souls"}]}]}),
            "==", "the times try our souls"),
     eqTest(CFun({ts, [{ts, [{fs, [{italic, "the times"}]},
                             {fs, [{bold, " try "}, {italic, "our souls"}]}]},
                       {ts, [{fs, [{roman, " and these tests"}]},
                             {fs, [{italic, " try "}, {roman, "our patience"}]}]}]}),
            "==", "the times try our souls and these tests try our patience"),
     eqTest(CFun({ts, [{ts, [{ts, [{ts, [{ts, [{fs, [{italic, "deep!"}]}]}]}]}]}]}),
            "==", "deep!")     ].
```

4. (15 points) [UseModels] This is a problem about the `text:text()` type in Figure 1 on page 4. In Erlang, write a function, `embolden`, whose type is given by the following:

```
-spec embolden(T :: text:text()) -> text:text().
```

This function takes a text, `T`, as an input and returns a text that is just like `T` but in which each formatted string occurring in it is made bold. Be sure to follow the grammar given by the types in Figure 1 on page 4! The following are tests.

```
-module(embolden_tests).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0, tests/1]).
main() -> compile:file(embolden),
    dotests("embolden_tests $Revision: 1.1 $", tests(fun embolden:embolden/1)).
-spec tests(EBFun :: fun((text:text()) -> text:text()))
          -> testing:testCase([text:text()]).
tests(EBFun) ->
    [eqTest(EBFun({fs, []}), "==", {fs, []}),
     eqTest(EBFun({ts, []}), "==", {ts, []}),
     eqTest(EBFun({fs, [{italic,"are changin'"}]}),
            "==", {fs, [{bold, "are changin'"}]}),
     eqTest(EBFun({fs, [{roman, "the times"},{italic," are changin'"}]}),
            "==", {fs, [{bold, "the times"}, {bold," are changin'"}]}),
     eqTest(EBFun({ts, [{fs, [{italic, "the times"}]},
                        {fs, [{bold, " try "}, {italic, "our souls"}]}]}),
            "==", {ts, [{fs, [{bold, "the times"}]},
                        {fs, [{bold, " try "}, {bold, "our souls"}]}]}),
     eqTest(EBFun({ts, [{ts, [{fs, [{italic, "the times"}]},
                              {fs, [{bold, " try "}, {italic, "our souls"}]}]},
                        {ts, [{fs, [{roman, " and these tests"}]},
                              {fs, [{italic, " try "}, {roman, "our patience"}]}]}]}),
            "==", {ts, [{ts, [{fs, [{bold, "the times"}]},
                              {fs, [{bold, " try "}, {bold, "our souls"}]}]},
                        {ts, [{fs, [{bold, " and these tests"}]},
                              {fs, [{bold, " try "}, {bold, "our patience"}]}]}]}),
     eqTest(EBFun({ts, [{ts, [{ts, [{ts, [{ts, [{fs, [{italic, "deep!"}]}]}]}]}]}]}),
            "==", {ts, [{ts, [{ts, [{ts, [{ts, [{fs, [{bold, "deep!"}]}]}]}]}]}]}) ].
```

5. (15 points) [UseModels] This is a problem about the `text:text()` type in Figure 1 on page 4. In Erlang, write a function, `maptext/2`, whose type is given by the following.

```
-spec maptext(F::fun((text:formatted()) -> text:formatted()), Txt::text:text()) -> text:text().
```

This function takes a 1-argument function, `F`, and a text, `Txt`, and returns a text that is just like `T`, except that every formatted text `FT` that occurs inside `Txt` is replaced by `F(FT)`. Be sure to follow the grammar given by the types in Figure 1 on page 4! The following are tests.

```
-module(maptext_tests).
-import(testing,[dotests/2,eqTest/3]).
-export([main/0, tests/1]).
main() -> compile:file(maptext),
    dotests("maptext_tests $Revision: 1.1 $", tests(fun maptext:maptext/2)).
-spec tests(MTFun :: fun((fun((text:formatted()) -> text:formatted()),
                          text:text()) -> text:text())) -> testing:testCase([text:text()]).
tests(MTFun) ->
    [eqTest(MTFun(fun id/1, {fs, []}), "==", {fs, []}),
     eqTest(MTFun(mk(bold), {ts, []}), "==", {ts, []}),
     eqTest(MTFun(mk(bold), {fs, [{italic,"are changin'"}]}), "==", {fs, [{bold, "are changin'"}]}),
     eqTest(MTFun(fun rev/1, {fs, [{roman, "the times"},{italic," are changin'"}]}),
            "==", {fs, [{roman, "semit eht"},{italic,"'nignahc era "}]}),
     eqTest(MTFun(mk(italic), {ts, [{fs, [{italic, "the times"}]},
                                    {fs, [{bold, " try "}, {italic, "our souls"}]}]}),
            "==", {ts, [{fs, [{italic, "the times"}]},
                        {fs, [{italic, " try "}, {italic, "our souls"}]}]}),
     eqTest(MTFun(fun bang/1,
                  {ts, [{ts, [{fs, [{italic, "the times"}]},
                              {fs, [{bold, " try "}, {italic, "our souls"}]}]},
                        {ts, [{fs, [{roman, " and these tests"}]},
                              {fs, [{italic, " try "}, {roman, "our patience"}]}]}]}),
            "==", {ts, [{ts, [{fs, [{italic, "the times!"}]},
                              {fs, [{bold, " try !"}, {italic, "our souls!"}]}]},
                        {ts, [{fs, [{roman, " and these tests!"}]},
                             {fs, [{italic, " try !"}, {roman, "our patience!"}]}]}]}),
     eqTest(MTFun(fun rev/1,
                  {ts, [{ts, [{ts, [{ts, [{ts, [{fs, [{italic, "deep!"}]}]}]}]}]}]}),
            "==", {ts, [{ts, [{ts, [{ts, [{ts, [{fs, [{italic, "!peed"}]}]}]}]}]}]}) ].
% Some functions for testing purposes, NOT FOR YOU TO IMPLEMENT!
id(FT) -> FT.
mk(FACE) -> fun({_Face,STR}) -> {FACE, STR} end.
rev({Face,STR}) -> {Face, lists:reverse(STR)}.
bang({Face,STR}) -> {Face, STR++"!"}.
```

6. (10 points) [Concepts] [UseModels] Suppose we want to generalize the previous three problems involving the text type from Figure 1 on page 4. That is, suppose we want to have a function `foldtext/4` whose type is given by the following:

```
-spec foldtext(FL :: fun(([R]) -> R), FF :: fun((text:formatted()) -> R),
               TF :: fun(([R]) -> R), T :: text:text()) -> R.
```

This function should be such that, for any desired result type `R`, it takes three function arguments, `FL`, of type `fun(([R]->R)`, `FF`, of type `fun((text:formatted()) ->R)`, and `TF`, of type `fun(([R]) ->R)`, and a text, `T`, and returns a value of type `R`. This function is an abstraction of the pattern of recursion over values of type `text:text()`. It should apply `FF` to all the formatted values in the `fs` case and apply `FL` to the list of those results, and should recurse over the list of texts in the `ts` case and apply `TF` to the list of those results. The following are test cases, written using the test cases in the previous 3 problems.

```
% Functions that exercise foldtests, NOT FOR YOU TO IMPLEMENT
chars(Txt) -> foldtext(fun lists:concat/1, fun({_Face,STR}) -> STR end, fun lists:concat/1, Txt).
embolden(Txt) -> foldtext(fun text:fs/1, fun({_Face,STR}) -> {bold, STR} end, fun text:ts/1, Txt).
maptext(F,Txt) -> foldtext(fun text:fs/1, fun({Face,STR}) -> F({Face, STR}) end, fun text:ts/1, Txt).
main() -> compile:file(foldtext), io:format("foldtext_tests $Revision: 1.1 $~n",[]),
    dotests("chars_tests", chars_tests:tests(fun chars/1)),
    dotests("embolden_tests", embolden_tests:tests(fun embolden/1)),
    dotests("maptext_tests", maptext_tests:tests(fun maptext/2)).
```

Your task in this problem is to choose which one of the following, if any, is a declaration that correctly implements `foldtext`. The correct implementation should have the type and behavior described above and satisfy the test cases given above. (So don't ask us why some choice has a type error or is incorrect during the test — it's because it is the wrong answer!) Circle the letter of the correct choice.

A.
```
foldtext(FL, FF, TF, LS) -> TF(FL(lists:map(FF, LS))).
```

B.
```
foldtext(FL, FF, _TF, {fs, LOF}) -> text:fs(FL(lists:map(FF, LOF)));
foldtext(FL, FF, TF, {ts, LOT}) ->
       text:ts(TF(lists:map(fun(T) -> foldtext(FL, FF, TF, T) end, LOT))).
```

C.
```
foldtext(FL, FF, _TF, {fs, LOF}) -> text:fs(lists:map(FF, LOF));
foldtext(FL, FF, TF, {ts, LOT}) ->
       text:ts(lists:map(fun(T) -> foldtext(FL, FF, TF, T) end, LOT)).
```

D.
```
foldtext(FL, FF, TF, {fs, LOF}) -> foldtext(FL, FF, TF, {fs, LOF});
foldtext(FL, FF, TF, {ts, LOT}) -> foldtext(FL, FF, TF, {ts, LOT}).
```

E.
```
foldtext(_FL, FF, _TF, {fs, LOF}) -> lists:map(FF, LOF);
foldtext(FL, FF, TF, {ts, LOT}) ->
       lists:map(fun(T) -> foldtext(FL, FF, TF, T) end, LOT).
```

F.
```
foldtext(FL, FF, _TF, {fs, LOF}) -> FL(lists:map(FF, LOF));
foldtext(FL, FF, TF, {ts, LOT}) ->
       TF(lists:map(fun(T) -> foldtext(FL, FF, TF, T) end, LOT)).
```

G.
```
foldtext(FL, FF, _TF, {fs, LOF}) -> FL(lists:map(FF,LOF));
foldtext(FL, FF, TF, {ts, LOT}) -> foldtext(FL, FF, TF, LOT).
```

H.
```
foldtext(FL, FF, _TF, {fs, LOF}) -> FL(FF(LOF));
foldtext(FL, FF, TF, {ts, LOT}) -> foldtext(FL, FF, TF, LOT).
```

I. None of the above purported solutions are correct.

Infinite Bag Data Type Problem

7. [Concepts] [UseModels] In this problem you will implement three functions on infinite bags in Erlang. An infinite bag, or multiset, can hold any number of items; for example 6 Cokes and 2 Pepsis. Infinite bags of type T are to be represented as pairs containing the atom bag and a function that takes a T value and returns a non-negative integer.

```
-type infbag(T) :: {bag, fun((T) -> non_neg_integer())}.
```

The three functions you will program are described below.

(a) (9 points) The function fromRule/1 takes a function, F, and returns an infinite bag (using the representation above).

```
-spec fromRule(F::fun((T) -> non_neg_integer())) -> infbag(T).
```

The resulting infinite bag contains a value X exactly F(X) times.

(b) (6 points) The function how_many/2 takes an infinite bag, B, and a value What, and returns the number of times that What is contained in B.

```
-spec how_many(B::infbag(T), What::T) -> non_neg_integer().
```

(c) (10 points) The function union/2 takes two infinite bags, B1 and B2, and returns an infinite bag that contains every value X as the number of times that X is contained in B1 plus the number of times that X is contained in B2.

```
-spec union(B1::infbag(T), B2::infbag(T)) -> infbag(T).
```

The following are tests.

```
-module(infbag_tests).
-import(testing,[dotests/2, eqTest/3]).
-import(infbag,[fromRule/1,how_many/2,union/2]).
-export([main/0, tests/0]).
main() -> compile:file(infbag),
    dotests("infbag_tests $Revision: 1.2 $", tests()).
-spec tests() -> [testing:testCase(non_neg_integer())].
tests() ->
    EverythingOnce = fromRule(fun(_) -> 1 end),
    SixCokeTwoPepsi = fromRule(fun(What) -> case What of
                                                coke -> 6;
                                                pepsi -> 2;
                                                _ -> 0
                                            end
                               end),
    EO6C2P = union(EverythingOnce, SixCokeTwoPepsi),
    [eqTest(how_many(SixCokeTwoPepsi, coke), "==", 6),
     eqTest(how_many(SixCokeTwoPepsi, pepsi), "==", 2),
     eqTest(how_many(EverythingOnce, bmw), "==", 1),
     eqTest(how_many(EverythingOnce, ford), "==", 1),
     eqTest(how_many(EverythingOnce, coke), "==", 1),
     eqTest(how_many(EO6C2P, coke), "==", 7),
     eqTest(how_many(EO6C2P, pepsi), "==", 3),
     eqTest(how_many(EO6C2P, chevy), "==", 1),
     eqTest(how_many(EO6C2P, knitting), "==", 1),
     eqTest(how_many(EO6C2P, warblers), "==", 1),
     eqTest(how_many(fromRule(fun(_) -> 42 end), hhgg), "==", 42)  ].
```

There is space for your answer on the next page.

For your answer, complete the following module.

```
-module(infbag).
-export([fromRule/1, how_many/2, union/2]).
-export_type([infbag/1]).
-type infbag(T) :: {bag, fun((T) -> non_neg_integer())}.

-spec fromRule(F::fun((T) -> non_neg_integer())) -> infbag(T).
-spec how_many(B::infbag(T), What::T) -> non_neg_integer().
-spec union(B1::infbag(T), B2::infbag(T)) -> infbag(T).
```