Fall, 2010 Name: _____

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

COP 4020 — Programming Languages 1

Test on Declarative Programming Techniques

Special Directions for this Test

This test has 8 questions and pages numbered 1 through 11.
This test is open book and notes.
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.
When you write Oz code on this test, you may use anything in the declarative model (as in chapters 2–3 of our textbook). So you must *not* use not use cells or the library functions `IsDet` and `IsFree`. But please use all linguistic abstractions and syntactic sugars that are helpful.
You are encouraged to define functions or procedures not specifically asked for if they are useful to your programming; however, if they are not in the Oz base environment, then you must write them into your test. You can use the built-in functions in the Oz base environment such as `Append`, `Not`, `Reverse`, `Length`, `Filter`, `Map`, and `FoldR`.

For Grading

Question:	1	2	3	4	5	6	7	8	Total
Points:	10	10	15	10	10	10	15	20	100
Score:									

1. (10 points) [UseModels] Write an iterative function

   ```
   CountGreater: <fun {$ <List Int> <Int>}: <Int> >
   ```

   that takes a list `Ints` of integers and an integer `Num`, and returns a count of the number of elements of `Ints` that are strictly greater than `Num`.

   Your solution must have iterative behavior, and must be written using tail recursion. Don't use any higher-order functions, and don't use the Oz **for** loop syntax in your solution! (You are supposed to know what these directions mean.)

   The following are examples, that use the `Test` procedure from the homework.

   ```
   \insert 'CountGreater.oz'
   {Test {CountGreater nil 7} '==' 0}
   {Test {CountGreater [2] 7} '==' 0}
   {Test {CountGreater [7 2] 6} '==' 1}
   {Test {CountGreater [9 8 1 2 7 6 6 6 6 1 9 2 10] 6} '==' 5}
   {Test {CountGreater [9 8 1 2 7 6 6 6 6 1 9 2 10] 5} '==' 9}
   {Test {CountGreater [9 8 1 2 7 6 6 6 6 1 9 2 10] 0} '==' 13}
   {Test {CountGreater [8 1 2 7 6 6 6 6 1 9 2 10] 0} '==' 12}
   ```

2. (10 points) [UseModels] Write a function

   ```
   Xerox: <fun {$ <List T>}: <List T>
   ```

   that takes a list of some type T, `Lst`, and returns a list that is just like `Lst` except that each element of `Lst` is duplicated.

   The following are examples.

   ```
   \insert 'Xerox.oz'
   {Test {Xerox nil} '==' nil}
   {Test {Xerox [bar]} '==' [bar bar]}
   {Test {Xerox [foo bar]} '==' [foo foo bar bar]}
   {Test {Xerox [1 2 3 4 5 0]} '==' [1 1 2 2 3 3 4 4 5 5 0 0]}
   {Test {Xerox [l i s t]} '==' [l l i i s s t t]}
   ```

3. (15 points) [UseModels] Using Oz's built-in `FoldR` function, write the function

   ```
   Count: <fun {$ <List T> <T>}: <Int> >
   ```

   that, for some type `T`, takes two arguments: `Lst`, which is a list of values of type `T`, and `Elem`, which is a value of type `T`. The function you are to write, `Count`, returns an integer that is equal to the number of times that an element equal to `Elem` is found in `Lst`. Use the `==` operator to tell whether an element of `Lst` is equal to `Elem`. The following are examples.

   ```
   \insert 'Count.oz'
   {Test {Count nil 7} '==' 0}
   {Test {Count [7 2 1 7 3] 7} '==' 2}
   {Test {Count [2 1 7 3] 7} '==' 1}
   {Test {Count [a g o o d t i m e] o} '==' 2}
   {Test {Count [a g o o d t i m e] e} '==' 1}
   {Test {Count [e e e k s a i d m i n e e] e} '==' 5}
   ```

   Your solution must use Oz's built-in `FoldR` function (but you can also write additional helping functions if you wish)! So you must fill in your answer by completing the code outline below.

   ```
   declare
   fun {Count Lst Elem}
       {FoldR
   
   
   
   
   
   
   
   
   
   
   
       }
   end
   ```

4. (10 points) [Concepts] [UseModels] Write a curried version of the function `Count`, from question 3 on the previous page. The function you are to write should be called `CurriedCount`. That is, write

   ```
   CurriedCount: <fun {$ <List T>}: <fun {$ <T>}: <Int>>>
   ```

   that, for some type `T`, `CurriedCount` takes an argument, `Lst`, which is a list of values of type `T`, and returns a function that takes as an argument, `Elem` of type `T`, and which returns the number of times that an element equal to `Elem` occurs in `Lst`. For a call such as `{{CurriedCount Lst} Elem}` the result is an integer that tells how many times an element equal to `Elem` occurs in `Lst`. The following are examples.

   The following are examples.

   ```
   \insert 'CurriedCount.oz'
   {Test {{CurriedCount nil} 7} '==' 0}
   {Test {{CurriedCount [7 2 1 7 3]} 7} '==' 2}
   {Test {{CurriedCount [2 1 7 3]} 7} '==' 1}
   {Test {{CurriedCount [a g o o d t i m e]} o} '==' 2}
   {Test {{CurriedCount [a g o o d t i m e]} e} '==' 1}
   {Test {{CurriedCount [e e e k s a i d m i n e e]} e} '==' 5}
   ```

   Please write your answer below. To save time, you can call `Count` in your answer, instead of writing out the body of `Count` again.

   ```
   \insert 'Count.oz' % so you can use Count in your answer
   ```

5. (10 points) [UseModels] Using Oz's built-in Map function, write the function

```
ApplyList: <fun {$ <List <fun {$ T}: S>> <T>}: <List S>>
```

that, for some types T and S takes a list, Funs, of functions of type <**fun** {$ T}: S> and a value, X, of type T, and returns a list which contains the results of applying each function in Funs to X. The resulting list preserves the order of the functions in Funs. The following are examples.

```
\insert 'ApplyList.oz'
{Test {ApplyList nil 33} '==' nil}
{Test {ApplyList [fun {$ X} X+1 end] 4020} '==' [4021]}
{Test {ApplyList [fun {$ X} X+1 end fun {$ X} X+2 end] 4020} '==' [4021 4022]}
{Test {ApplyList
       local AddC = fun {$ Y} fun {$ X} X+Y end end in
          {Map [1 2 3 4 5 2 27 999] AddC}
       end
       1000}
 '==' [1001 1002 1003 1004 1005 1002 1027 1999]}
{Test {ApplyList [fun {$ X} bread#X#bread end fun {$ X} pita#X#pita end]
      turkey}
 '==' [bread#turkey#bread pita#turkey#pita]}
```

Write your answer by filling in the blanks in the following solution.

```
declare
fun {ApplyList Funs X}
   {Map









   }
end
```

6. (10 points) [UseModels] Write a function,
`Classify: <fun {$ <List T> <List <fun {$ T}: Bool>>}: <Pair T Bool>>` which for some type `T` takes a list `Lst` of elements of type `T` and a list of predicates, `Preds`, which is a list of functions of type `<fun {$ T}: Bool>` and returns a list of #-pairs containing an element from `Lst` and the result of applying the list of predicates in `Preds` to that element (in the sense of the `ApplyList` function in problem 5).

   The following are examples using the `Test` function from the homework.

```
\insert 'Classify.oz'
{Test {Classify nil nil} '==' nil}
{Test {Classify nil [IsEven IsOdd]} '==' nil}
{Test {Classify [4 0 7] [IsEven IsOdd]} '==' [4#[true false] 0#[true false] 7#[false true]]}
{Test {Classify [3 4] nil} '==' [3#nil 4#nil]}
{Test {Classify [~2 ~1 0 1 2 3 4 5] [fun {$ X} X > 0 end fun {$ X} X > 2 end]}
 '==' [~2#[false false] ~1#[false false] 0#[false false] 1#[true false]
       2#[true false] 3#[true true] 4#[true true] 5#[true true]]}
{Test {Classify "a_list_2" [fun {$ Ch} Ch == &_ end Char.isAlpha Char.isDigit]}
 '==' [&a#[false true false] &_#[true false false]
       &l#[false true false] &i#[false true false] &s#[false true false]
       &t#[false true false] &_#[true false false]
       &2#[false false true]]}
local Digits = [0 1 2 3 4 5 6 7 8 9]
      fun {NFalse N} if N =< 0 then nil else false|{NFalse N-1} end end
   in
   {Test {Classify Digits {Map Digits fun {$ D} fun {$ I} I == D end end}}
    '=='
    {Map Digits fun {$ I} I#{Append {NFalse I} true|{NFalse 10-I-1}} end}}
end
```

   In your solution you can use the `ApplyList` function from problem 5. Please write your answer below

```
\insert 'ApplyList.oz' % So you can use this in your solution if you wish
```

7. (15 points) [UseModels] This problem works with the type ⟨Music⟩, as defined by the following grammar. (Note that all the ⟨Int⟩s that occur in a ⟨Music⟩ are guaranteed to be non-negative.)

   ⟨Music⟩ ::= `pitch(`⟨Int⟩`)` | `chord(`⟨List Music⟩`)` | `sequence(`⟨List Music⟩`)`

   Write a function

   `Invert: <fun {$ <Music> <Int>} : <Music> >`

   that takes a ⟨Music⟩, `Song`, and a non-negative integer `Around`, and returns a ⟨Music⟩ that is like `Song` except that every integer `Note` occurring anywhere in `Song` is replaced by `Around - (Note - Around)`. (This formula makes notes that are lower than `Around` turn into notes above it and vice versa, hence the name.) You can assume that `Around` is such that no pitch integer in the result will be negative. Be sure to follow the grammar!

   See the following examples.

```
\insert 'Invert.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'Invert'}
{Test {Invert pitch(3) 5} '==' pitch(7)}
{Test {Invert pitch(3) 2} '==' pitch(1)}
{Test {Invert chord([pitch(5) pitch(3) pitch(1)]) 8}
 '==' chord([pitch(11) pitch(13) pitch(15)])}
{Test {Invert chord([sequence([pitch(3) pitch(5)]) sequence([pitch(7) pitch(1)])]) 9}
 '==' chord([sequence([pitch(15) pitch(13)]) sequence([pitch(11) pitch(17)])])}
{Test {Invert sequence([pitch(2) pitch(8) pitch(3) pitch(9)]) 10}
 '==' sequence([pitch(18) pitch(12) pitch(17) pitch(11)])}
{Test {Invert sequence([sequence([pitch(3) pitch(5)]) pitch(3)
                        chord([pitch(1) pitch(3) pitch(5)])
                        sequence([pitch(7) pitch(1)])]) 16}
 '==' sequence([sequence([pitch(29) pitch(27)]) pitch(29)
                chord([pitch(31) pitch(29) pitch(27)])
                sequence([pitch(25) pitch(31)])])}
{Test {Invert sequence([sequence([pitch(1) pitch(2) pitch(3)])
                        sequence([chord([pitch(5) pitch(9)])
                                  chord([pitch(3) pitch(5)])])]) 5}
 '==' sequence([sequence([pitch(9) pitch(8) pitch(7)])
                        sequence([chord([pitch(5) pitch(1)])
                                  chord([pitch(7) pitch(5)])])])}
{Test {Invert sequence([sequence([sequence([pitch(3) pitch(5)])
                                  pitch(3)
                                  chord([pitch(1) pitch(3) pitch(5)])
                                  sequence([pitch(7) pitch(1)])])
                        sequence([sequence([pitch(1) pitch(2) pitch(3)])
                                  sequence([chord([pitch(5) pitch(9)])
                                            chord([pitch(3) pitch(5)])])])
                     pitch(5)]) 8}
 '==' sequence([sequence([sequence([pitch(13) pitch(11)])
                                  pitch(13)
                                  chord([pitch(15) pitch(13) pitch(11)])
                                  sequence([pitch(9) pitch(15)])])
                        sequence([sequence([pitch(15) pitch(14) pitch(13)])
                                  sequence([chord([pitch(11) pitch(7)])
                                            chord([pitch(13) pitch(11)])])])
                     pitch(11)])}
```

   There is room for your answer on the next page.

Please put your answer to the `Invert` problem below.

8. (20 points) [UseModels] This problem works with the type "Entry," as defined by the following grammar.

⟨Entry⟩ ::= `file(name:` ⟨String⟩ `bytes:` ⟨String⟩`)`
| `directory(name:` ⟨String⟩ `entries:` ⟨List Entry⟩`)`

Write a function

`Rename: <fun {$ <Entry> <String> <String>} : <Entry> >`

that takes a `<Entry>`, `Ent`, and two strings, `Old` and `New`, and returns a new `<Entry>`, with the value of each name field that is equal to `Old` changed to `New`.

The following are examples.

```
\insert 'TestingNoStop.oz'
\insert 'Rename.oz'
{Test {Rename file(name: "Cassius" bytes: "The Greatest by Cassius Clay...")
             "Cassius" "Mohammed"}
 '==' file(name: "Mohammed" bytes: "The Greatest by Cassius Clay...")}
{Test {Rename directory(name: "MyDir"
                       entries: [file(name: "foo" bytes: "fooish stuff...")])
       "foo" "bar"}
 '==' directory(name: "MyDir"
                entries: [file(name: "bar" bytes: "fooish stuff...")])}
{Test {Rename directory(name: "MyStuff"
                       entries: [file(name: "secrets" bytes: "attack at dawn!")
                                 file(name: "fred" bytes: "fred")
                                 file(name: "fred" bytes: "rubble")])
       "fred" "barney"}
 '==' directory(name: "MyStuff"
                       entries: [file(name: "secrets" bytes: "attack at dawn!")
                                 file(name: "barney" bytes: "fred")
                                 file(name: "barney" bytes: "rubble")])}
{Test {Rename directory(name: "ISU"
                       entries:
                          [directory(name: "MyStuff"
                                     entries: [file(name: "secrets" bytes: "attack at dawn!")
                                               file(name: "fred" bytes: "fred")
                                               file(name: "fred" bytes: "rubble")])
                           directory(name: "YourStuff"
                                     entries: [file(name: "ISU" bytes: "Iowa State")
                                               file(name: "Cyclones" bytes: "football!")])])
       "ISU" "UCF"}
 '==' directory(name: "UCF"
                       entries:
                          [directory(name: "MyStuff"
                                     entries: [file(name: "secrets" bytes: "attack at dawn!")
                                               file(name: "fred" bytes: "fred")
                                               file(name: "fred" bytes: "rubble")])
                           directory(name: "YourStuff"
                                     entries: [file(name: "UCF" bytes: "Iowa State")
                                               file(name: "Cyclones" bytes: "football!")])])}
```

There is space for your answer on the next page.

Please put your answer to the `Rename` problem below.