Fall, 2008 Name: ______________________

COP 4020 — Programming Languages 1

Test on Declarative Programming Techniques

Special Directions for this Test

This test has 5 questions and pages numbered 1 through 7.

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 imperative features (such as cells and assignment) or the library functions `IsDet` and `IsFree`.

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 like `Append`, `Filter`, `Map`, and `FoldR`.

For Grading

Problem	Points	Score
1	15	
2	15	
3	20	
4	15	
5	35	

1. (15 points) [UseModels] Write an iterative function

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

   that takes a list `Elems` of values of some type T and a value `Sought` of type T, and returns the first index (counting from 1) in `Elems` whose value is equal to `Sought` (using ==), if it exists. If there is no element in `Elems` that is == to `Sought`, then the result is `~1`.

   Your solution must have iterative behavior, and must be written using tail recursion. Don't use any higher-order functions or 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.

   ```
   {Test {Find [a b c d] a} '==' 1}
   {Test {Find [a b c d] e} '==' ~1}
   {Test {Find [a b c d e f g] c} '==' 3}
   {Test {Find [[99] [86] [12 22] nil [3]] [3]} '==' 5}
   {Test {Find [nil [3]] [3]} '==' 2}
   {Test {Find nil 72} '==' ~1}
   {Test {Find [penn ohio flor iowa] vote} '==' ~1}
   ```

2. (15 points) [UseModels] Write a function

```
RoboCall: <fun {$ <List Atom> <List Atom> <List Atom>}: <List <List Atom>>>
```

that takes three lists of atoms, `Voters`, `Text1`, and `Text2`, and produces a list of lists of atoms. The resulting list of lists contains, for each atom V in `Voters`, a list that contains, in order, the elements of `Text1`, V, and then the elements of `Text2`. Thus each V appears spliced in between the lists `Text1` and `Text2`. The following are examples.

```
{Test {RoboCall nil [hi] [give me your vote]} '==' nil}
{Test {RoboCall [jack jane jill joe] [hello there] [please vote 'for' me]}
 '==' [[hello there jack please vote 'for' me]
       [hello there jane please vote 'for' me]
       [hello there jill please vote 'for' me]
       [hello there joe please vote 'for' me]]}
{Test {RoboCall [fanny freddie wamu] [hi] [please give me your vote tuesday]}
 '==' [[hi fanny please give me your vote tuesday]
       [hi freddie please give me your vote tuesday]
       [hi wamu please give me your vote tuesday]]}
{Test {RoboCall [barak joe john sarah] [hey] [please stop calling my phone]}
 '==' [[hey barak please stop calling my phone]
       [hey joe please stop calling my phone]
       [hey john please stop calling my phone]
       [hey sarah please stop calling my phone]]}
```

3. (20 points) [UseModels] Write a higher-order function

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

   that takes two integers, `Low` and `High`, and returns a function that takes a list of integers `Temps` and produces a list that contains the elements of `Temps` that are between `Low` and `High` (inclusive), in their original order. The following are examples.

   ```
   local FLTemps = {StateTemps 60 90} in
      {Test {FLTemps nil} '==' nil}
      {Test {FLTemps [60 90 32 72]} '==' [60 90 72]}
      {Test {FLTemps [59 91]} '==' nil}
      {Test {FLTemps [100 95 90 85 80 75 70 65 60 55 50 45 40]} '==' [90 85 80 75 70 65 60]}
      {Test {FLTemps [~2 10 5 91 77 3 77 5 89 89 77]} '==' [77 77 89 89 77]}
   end
   local IATemps = {StateTemps ~30 100} in
      {Test {IATemps [60 90 0 ~4 ~30 ~33 32 172]} '==' [60 90 0 ~4 ~30 32]}
      {Test {IATemps [~20 10 5 101 100 3 100 5 89 89 100]}
       '==' [~20 10 5 100 3 100 5 89 89 100]}
   end
   {Test {{StateTemps ~50 65} [~51 ~50 60 ~50]} '==' [~50 60 ~50]}
   ```

4. (15 points) [UseModels] Using `FoldR`, write the function

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

   that for some types `T` and `S`, takes a list of functions `Funs` (which has type `<List <fun {$ T}: S>>`), and a value `X` of type `T`, and returns a list that results from applying each element of `Funs` to `X`, and returning a list of the results (preserving the order of the `Funs` list). That is, for functions F_1, F_2, ..., F_n, and value X, we have:

   `{ApplyList [`F_1 F_2 `...` F_n`]` X`} == [{`F_1 X`} {`F_2 X`} ... {`F_n X`}]`

   The following are examples.

```
{Test {ApplyList [Not Not] true} '==' [false false]}
{Test {ApplyList [fun {$ X} bread#X#bread end fun {$ X} pita#X#pita end]
       turkey}
 '==' [bread#turkey#bread pita#turkey#pita]}
{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]}
```

   Since you must use `FoldR`, write your answer by filling in the following outline (you can also write helping functions if you wish).

```
fun {ApplyList Funs X}
   {FoldR




   }
end
```

5. (35 points) [UseModels] This problem is about the ‘statement and expression” grammar:

   ⟨Statement⟩ ::= `expStmt(`⟨Expression⟩`)`
   | `assignStmt(`⟨Atom⟩ ⟨Expression⟩`)` | `ifStmt(`⟨Expression⟩ ⟨Statement⟩`)`
   ⟨Expression⟩ ::= `varExp(`⟨Atom⟩`)` | `numExp(`⟨Number⟩`)`
   | `equalsExp(`⟨Expression⟩ ⟨Expression⟩`)` | `beginExp(`⟨List Statement⟩ ⟨Expression⟩`)`

   Write a function `OptimizeIfs: <`**`fun`**` {$ <Statement>}: <Statement>>` that optimizes a statement, `Stmt`, by returning a statement that is just like `Stmt` except that each statement of the form `ifStmt(`E S`)`, where E is necessarily true, is replaced by an optimized version of S. Expression E is *necessarily true* if it is an `equalsExp` with both arguments the same variable or number. In your solution, you can use function on the following page, `NecessarilyTrue`, that computes this. The following are examples using the `Test` function from the homework. Note that the process occurs recursively for all subparts of `Stmt`, even the expression part of an `ifStmt` and within replaced parts of statements.

```
{Test {OptimizeIfs ifStmt(varExp(b) assignStmt(x numExp(3)))}
 '==' ifStmt(varExp(b) assignStmt(x numExp(3)))}
{Test {OptimizeIfs ifStmt(equalsExp(varExp(b) varExp(b))
                          assignStmt(q numExp(53)))}
 '==' assignStmt(q numExp(53))}
{Test {OptimizeIfs ifStmt(equalsExp(numExp(402) numExp(402))
                          ifStmt(equalsExp(varExp(b) varExp(b))
                                 assignStmt(i numExp(22))))}
 '==' assignStmt(i numExp(22))}
{Test {OptimizeIfs
       expStmt(beginExp([ifStmt(equalsExp(varExp(b) varExp(b))
                                assignStmt(x numExp(3)))
                         assignStmt(y numExp(4))]
                        varExp(y)))}
 '==' expStmt(beginExp([assignStmt(x numExp(3)) assignStmt(y numExp(4))]
                       varExp(y)))}
{Test {OptimizeIfs expStmt(beginExp([ifStmt(equalsExp(numExp(402) numExp(3))
                                            ifStmt(equalsExp(varExp(b) varExp(b))
                                                   assignStmt(i numExp(22))))]
                                    varExp(i)))}
 '==' expStmt(beginExp([ifStmt(equalsExp(numExp(402) numExp(3))
                               assignStmt(i numExp(22)))]
                       varExp(i)))}
{Test {OptimizeIfs
       ifStmt(beginExp([ifStmt(equalsExp(numExp(4020) numExp(4020))
                               ifStmt(equalsExp(varExp(b) varExp(b))
                                      assignStmt(q numExp(53))))
                        assignStmt(y numExp(4))]
                       equalsExp(varExp(y) varExp(q)))
              assignStmt(z numExp(55)))}
 '==' ifStmt(beginExp([assignStmt(q numExp(53)) assignStmt(y numExp(4))]
                      equalsExp(varExp(y) varExp(q)))
             assignStmt(z numExp(55)))}
% The following illustrate base cases and other recursive cases
{Test {OptimizeIfs expStmt(numExp(3))} '==' expStmt(numExp(3))}
{Test {OptimizeIfs expStmt(varExp(y))} '==' expStmt(varExp(y))}
{Test {OptimizeIfs expStmt(equalsExp(varExp(y) varExp(z)))}
 '==' expStmt(equalsExp(varExp(y) varExp(z)))}
{Test {OptimizeIfs expStmt(equalsExp(varExp(y) varExp(y)))}
 '==' expStmt(equalsExp(varExp(y) varExp(y)))}
{Test {OptimizeIfs assignStmt(x numExp(3))} '==' assignStmt(x numExp(3))}
{Test {OptimizeIfs expStmt(beginExp(nil numExp(3)))} '==' expStmt(beginExp(nil numExp(3)))}
```

   There is space for your answer on the next page.

Put your answer to the `OptimizeIfs` problem below.

```
%% Assume this function in your solution
%% NecessarilyTrue : <fun {$ <expression>}: Bool>
fun {NecessarilyTrue Exp}
   case Exp of
      equalsExp(varExp(V1) varExp(V2)) then V1 == V2
   [] equalsExp(numExp(N1) numExp(N2)) then N1 == N2
   else false
   end
end
```