Name: _____

COP 4020 — Programming Languages I (Spring 2011)

Test on Declarative Programming Techniques

Directions for this test

Please read these instructions carefully before starting.

This test has 6 questions and pages numbered 1 through 7. This test is open book and notes, but no electronics.

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 we have seen in chapters 1–3 of our textbook. But unless specifically directed, you should not use imperative features (such as cells) or the library functions `IsDet` and `IsFree`.

You are encouraged to use and/or 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 their definition on your test. You can use the built-in functions in the Oz base environment such as `Append`, `Not`, `Reverse`, `Length`, `Filter`, `Map`, `FoldL`, `Member` and `FoldR`.

For Grading

Question:	1	2	3	4	5	6	Total
Points:	10	15	15	15	20	25	100
Score:							

1. (10 points) [UseModels] Write a function

   ```
   Retlif: <fun {$ <List T> <fun {$ T}: Boolean>}: <List T>>
   ```

   that takes a list `Lst` and a boolean-valued unary function `P` as arguments, and returns a list consisting of only those elements of `Lst` that **do not** satisfy the predicate `P`. (An element `Element` satisfies `P` if `{P Element}` returns **`true`**.) The following are examples:

   ```
   % $Id: RetlifTests.oz,v 1.2 2011/04/04 03:23:33 fhussain Exp fhussain $

   \insert 'TestingNoStop.oz'
   \insert 'Retlif.oz'

   {StartTesting 'Retlif $Revision: 1.2 $'}
   {Test {Retlif [2 4 10 8 1] fun {$ X} {IsEven X} end} '==' [1]}
   {Test {Retlif [10 7 8 3 80 5 4 2] fun {$ X} X>9 end} '==' [7 8 3 5 4 2]}
   {Test {Retlif [4 9 10 7 100] fun {$ X} X>=4 end} '==' nil}
   {Test {Retlif nil fun {$ _} true end} '==' nil}
   {EndTesting 'done'}
   ```

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

   ```
   Filter2: <fun {$ <List T> <List T> <fun {$ T T}: Boolean>}: <List <Pair T>>>
   ```

   that takes two lists, `Lst1` and `Lst2`, and a boolean-valued, binary function, `Pred`, and returns a list that is the same length as the shorter of `Lst1` and `Lst2`. The resultant list is constructed by applying `Pred` to the i^{th} elements of `Lst1` and `Lst2` (in that order), where `i` ranges from 1 to the length of the shorter list (list indexes in Oz begin with 1); if `Pred` is satisfied, the corresponding two elements are added to the resultant list as a Pair, where ⟨Pair T⟩ ::=pair(T T). The following are examples:

   ```
   % $Id: Filter2Test.oz,v 1.3 2011/04/03 19:36:29 fhussain Exp fhussain $

   \insert 'Filter2.oz'
   \insert 'TestingNoStop.oz'

   {StartTesting '$Revision: 1.3 $'}
   {Test {Filter2 [2 3 10 8 9] [20 5 0 1 100 10 20 80] fun {$ X Y} {IsEven X+Y} end}
    '==' [pair(2 20) pair(3 5) pair(10 0)]}
   {Test {Filter2 [10 90 8 1 ~5 2.345] [3 2] fun {$ X Y} (X*Y)>50 end} '==' [pair(90 2)]}
   {Test {Filter2 [2 3 10 8 9] nil fun {$ X Y} {IsEven X+Y} end} '==' nil}
   {Test {Filter2 nil [2 3 10 8 9]  fun {$ X Y} {IsEven X+Y} end} '==' nil}
   {Test {Filter2 nil nil  fun {$ X Y} {IsEven X+Y} end} '==' nil}
   {EndTesting 'done'}
   ```

3. (15 points) [UseModels] Write an **iterative** function

```
MyMapInd: <fun {$ <List T> <fun {$ <Integer> <T>}: S>}: <List S>>
```

that takes a list `Lst` and a function `F` as arguments and returns a list. `F` is a binary function that is applied to each element in `Lst` with the index of the list element taken as the first argument (list indexes in Oz begin with 1) and the element itself taken as the second argument. For this problem, you are prohibited from using the Oz base environment's `mapInd` function. The following are examples:

```
% $Id: MyMapIndTest.oz,v 1.3 2011/04/03 20:06:56 fhussain Exp fhussain $

\insert 'TestingNoStop.oz'
\insert 'MyMapInd.oz'

{StartTesting 'MyMapInd $Revision: 1.3 $'}
{Test {MyMapInd [d a e] fun {$ I A} I#A end} '==' [1#d 2#a 3#e]}
{Test {MyMapInd [2 3 10 80 5 11] fun {$ I X} I#((X mod 2) == 0) end} '=='
 [1#true 2#false 3#true 4#true 5#false 6#false]}
{Test {MyMapInd nil fun {$ _ X} X end} '==' nil}
{Test {MyMapInd ['sizeA' 'sizeB' 'sizeC' 'sizeD']
       fun {$ I X} if {IsEven I} then X else 'default' end end} '=='
 ['default' 'sizeB'  'default' 'sizeD']}
{EndTesting 'done'}
```

4. (15 points) [UseModels] Write a function

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

that takes two lists, `Lst1` and `Lst2` as arguments and returns a list, `ResLst` containing those elements of `Lst1` that also occur in `Lst2`. Elements appear in `RestLst` in the same order that they appear in `Lst1`. The following are examples:

```
% $Id: InBothTest.oz,v 1.6 2011/04/03 20:22:53 fhussain Exp $

\insert 'TestingNoStop.oz'
\insert 'InBoth.oz'

{StartTesting 'InBoth $Revision: 1.6 $'}
{Test {InBoth nil nil} '==' nil}
{Test {InBoth nil [3 5 7 10]} '==' nil}
{Test {InBoth [2 5 10] nil} '==' nil}
{Test {InBoth [3 2 5 10 5]  [2 5]} '==' [2 5 5]}
{Test {InBoth [2 3 1 1 1] [1 2 3]} '==' [2 3 1 1 1]}
{Test {InBoth ['hello' 'this' 'world' 'is' 'wonderful' 'life'] ['random' 'hello' 'life' 'delete']}
 '==' ['hello' 'life']}
{EndTesting 'done'}
```

5. (20 points) [UseModels] Using **FoldL**, write a function

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

   that takes a list, Ls as argument and returns a **list**, ResLs, consisting of the set of elements in Ls. Scanning the input list from left to right, only the first occurrence of an element is reported in the result; subsequent occurrences, if any, are ignored. Therefore, elements appear in ResLs in the same order that they appear in Ls. The following are examples:

   ```
   % $Id: UniqueTests.oz,v 1.2 2011/04/03 18:42:10 fhussain Exp fhussain $

   \insert 'TestingNoStop.oz'
   \insert 'Unique.oz'

   {StartTesting 'Unique $Revision: 1.2 $'}
   {Test {Unique [3 7 7 8 10 0 8]} '==' [3 7 8 10 0]}
   {Test {Unique nil} '==' nil}
   {Test {Unique [b a]} '==' [b a]}
   {Test {Unique [b b a]} '==' [b a]}
   {Test {Unique [a b b a]} '==' [a b]}
   {Test {Unique [b d a c a b b a]} '==' [b d a c]}
   {Test {Unique [3 9 10 10 0 0 9 10 3]} '==' [3 9 10 0]}
   {EndTesting 'done'}
   ```

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

   ```
   fun {Unique Lst}
   { FoldL









   }
   end
   ```

6. (25 points) [UseModels] Consider the grammar for "rules" shown below:

⟨Rule T S⟩ ::=
`singlerule(<fun {$ T}: S>)`
| `ruleset(<List ⟨Rule T S⟩> <fun {$ <List S>}: S>)`

Write a function:

`UseRule: <fun {$ <Rule T S> T}: S>`

that takes a ⟨Rule T S⟩, `Rule`, and a value of type T, `Element`, and returns a value of type S, by using the functionality of `Rule` described below.

Using a rule of the form `singlerule(F)` on some element `Element` generates the value returned by the function application `{F Element}`. Using a rule of the form `ruleset(Ls Selector)` on some element `Element` causes each of the functions in `Ls` to be applied to `Element`, resulting in another list, say `ResLst`; the final result is arrived at by applying the `Selector` on `ResLst` i.e. by the application `{Selector ResLst}`. The following are examples:

```
% $Id: UseRuleTests.oz,v 1.2 2011/04/04 03:37:55 fhussain Exp $

\insert 'TestingNoStop.oz'
\insert 'UseRule.oz'

{StartTesting 'UseRule $Revision: 1.2 $'}
declare R0 R1 R2 R3 in
{Test {UseRule singlerule(fun{$ X} X*X end) 10} '==' 100}

{Test {UseRule ruleset([singlerule(fun {$ X} X+1 end) singlerule(fun {$X} X+2 end)] fun {$ L} L.2.1 end) 100} '==' 102}

{Test {UseRule ruleset([singlerule(fun {$ K} K*2 end) singlerule(fun {$ K} K+1 end)
                        singlerule(fun {$ _} 0 end)] fun {$ L} {Reverse L}.1 end)
       50} '==' 0}

{Test {UseRule ruleset([singlerule(fun {$ K} K*2 end)
              ruleset([singlerule(fun {$ K} K+1 end)
                       singlerule(fun {$K} K*K end)] fun {$ L} {Reverse L}.1 end)]
                       fun {$ L} {Reverse L}.1 end)
       50} '==' 2500}


{EndTesting 'done'}
```