Spring, 2012 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 9.
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. 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 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`, `Nth`, `Reverse`, `Length`, `Filter`, `Map`, and `FoldR`.

For Grading

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

1. (10 points) [UseModels] Write, in Oz, a tail-recursive function:

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

   that takes a value, `V`, of some type `T`, and an integer `N`, and returns a list of length `N` in which each element is `V`. Your code should assume that `N` is non-negative.

   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.

   ```
   % $Id: DuplicateTest.oz,v 1.1 2012/03/18 01:58:38 leavens Exp leavens $
   \insert 'Duplicate.oz'
   \insert 'TestingNoStop.oz'
   {StartTesting 'DuplicateTest.oz $Revision: 1.1 $'}
   {Test {Duplicate x 0} '==' nil}
   {Test {Duplicate b 3} '==' b|b|b|nil}
   {Test {Duplicate a 7} '==' a|a|a|a|a|a|a|nil}
   {Test {Duplicate 5 10} '==' [5 5 5 5 5 5 5 5 5 5]}
   {Test {Duplicate nil 2} '==' [nil nil]}
   {Test {Duplicate easy 1} '==' [easy]}
   {Test {Duplicate easy 5} '==' [easy easy easy easy easy]}
   {Test {Duplicate [o k] 3} '==' [[o k] [o k] [o k]]}
   {DoneTesting}
   ```

2. (10 points) [UseModels] Without using a **for**-loop, FoldR, or Map, write in Oz the function

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

   that takes a list of integers, LoI, and an integer, Factor, and returns a list integers in which each element is the result of multiplying Factor by the corresponding element in the original list. The following are examples.

   ```
   % $Id: ScaleTest.oz,v 1.1 2012/03/18 01:58:38 leavens Exp leavens $
   \insert 'Scale.oz'
   \insert 'TestingNoStop.oz'
   {StartTesting 'ScaleTest.oz $Revision: 1.1 $'}
   {Test {Scale nil 100} '==' nil}
   {Test {Scale 5|nil 100} '==' 500|nil}
   {Test {Scale 6|5|nil 100} '==' 600|500|nil}
   {Test {Scale [3 4 3 2] 10} '==' [30 40 30 20]}
   {Test {Scale [3 ~4 7 ~2] 0} '==' [0 0 0 0]}
   {Test {Scale [7 9 2 8 7 2 1 0 ~4 60] ~18} '==' [~126 ~162 ~36 ~144 ~126 ~36 ~18 0 72 ~1080]}
   {DoneTesting}
   ```

3. (10 points) [UseModels] Using Oz's built-in FoldR function, write the function

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

   from the previous problem.

   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 {Scale LoI Factor}
       {FoldR
   
   
   
   
   
   
   
       }
   end
   ```

4. (10 points) [Concepts] [UseModels] Write a curried version of the function `Compose`, shown below.

```
declare
fun {Compose F G X}
   {F {G X}}
end
```

The function you are to write should be called `CurriedCompose`. That is, write

```
CurriedCompose: <fun {$ <fun {$ <S>}: <U>>}:
                                  <fun {$ <fun {$ <T>}: <S>>}:
                                        <fun {$ <T>}: <U>>>>
```

such that, `CurriedCompose` takes a function, `F`, and returns a function that takes as an argument a function, `G`, and which returns a function that itself takes an argument, `X`, and returns the value of `{F {G X}}`.

The following are examples.

```
% $Id: CurriedComposeTest.oz,v 1.1 2012/03/18 01:58:38 leavens Exp leavens $
\insert 'CurriedCompose.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'CurriedComposeTest.oz $Revision: 1.1 $'}
{Test {{{CurriedCompose Not} IsList} 3} '==' {Not {IsList 3}}}
{Test {{{CurriedCompose IsInt} Length} nil} '==' {IsInt {Length nil}}}
{Test {{{CurriedCompose IsInt} fun {$ X} X+3 end} 7} '==' {IsInt 10}}
{Test {{{CurriedCompose fun {$ Y} Y*5 end} fun {$ X} X+3 end} 7} '==' 50}
{Test {{{CurriedCompose fun {$ Y} Y*5 end} fun {$ X} X+10 end} 3} '==' 65}
{DoneTesting}
```

Please write your answer below.

5. (10 points) [UseModels] In Oz, write a function,
ReplaceIn: <**fun** {$ <List <List <T>>> <T> <T>}: <List <List <T>>>> which takes a list of lists of elements of some type T, LL, and two elements of type T, Old and New, and returns a list of lists that is just like LL, except that all occurrences of Old in the inner lists are replace by New. The following are examples.

```
% $Id: ReplaceInTest.oz,v 1.2 2012/03/18 01:58:38 leavens Exp leavens $
\insert 'ReplaceIn.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'ReplaceInTest.oz $Revision: 1.2 $'}
{Test {ReplaceIn nil old new} '==' nil}
{Test {ReplaceIn [[foo bar old stuff]] old new} '==' [[foo bar new stuff]]}
{Test {ReplaceIn [[old old] nil [foo bar old stuff]] old new} '==' [[new new] nil [foo bar new stuff]]}
{Test {ReplaceIn [[7 6 5] [3 2] [9 3 1 2] [7 8]] 2 22} '==' [[7 6 5] [3 22] [9 3 1 22] [7 8]]}
{Test {ReplaceIn [["a" "bad" "" "scene"] ["was" "" "troubling"]] "" "fng"}
 '==' [["a" "bad" "fng" "scene"] ["was" "fng" "troubling"]]}
{DoneTesting}
```

6. (15 points) [UseModels] Consider the following grammar

   ⟨Route⟩ ::= `route(`⟨List ⟨Place⟩⟩`)`
   ⟨Place⟩ ::= `place(where:` ⟨Atom⟩ `features:` ⟨List ⟨Atom⟩⟩`)`

   In Oz, write a function, `AddFeature: <fun {$ <Route> <Atom> <Atom>}: <Route>>` which takes a ⟨Route⟩, `R`, and two atoms `Where` and `What`, and returns a ⟨Route⟩ that is just like `R`, except that in each ⟨Place⟩ whose `where` field is (== to) `Where`, `What` is added to the front of the list in that Place's `features` field. Here are examples:

```
\insert 'AddFeature.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'AddFeatureTest.oz $Revision: 1.1 $'}
{Test {AddFeature route(nil) dennys pancakes} '==' route(nil)}
{Test {AddFeature route(place(where: dennys features: cokes|icecream|nil)|nil) dennys pancakes}
 '==' route(place(where: dennys features: pancakes|cokes|icecream|nil)|nil)}
{Test {AddFeature route([place(where: iceland features: [cold]) place(where: iceland features: [ice])])
       iceland herring}
 '==' route([place(where: iceland features: [herring cold]) place(where: iceland features: [herring ice])])}
{Test {AddFeature route([place(where: atlanta features: [grits baseball]) place(where: paris features: [wine])
                         place(where: berlin features: [donuts]) place(where: berlin features: [museums])
                         place(where: paris features: [art]) place(where: atlanta features: [basketball])])
                 paris moonlight}
 '==' route([place(where: atlanta features: [grits baseball]) place(where: paris features: [moonlight wine])
             place(where: berlin features: [donuts]) place(where: berlin features: [museums])
             place(where: paris features: [moonlight art]) place(where: atlanta features: [basketball])])}
{DoneTesting}
```

7. (15 points) [UseModels] This problem uses the same grammar for ⟨Route⟩ as the previous problem.

   ⟨Route⟩ ::= `route(`⟨List ⟨Place⟩⟩`)`
   ⟨Place⟩ ::= `place(where:` ⟨Atom⟩ `features:` ⟨List ⟨Atom⟩⟩`)`

   In Oz, write the function `PlacesMap: <fun {$ <Route> <fun {$ <Place>}: <Place>>}: <Route>>`, which takes a route, `R`, and a function `PF`, (where `PF` takes a ⟨Place⟩ and returns a ⟨Place⟩), and returns a route that has the same shape as `R`, except that each ⟨Place⟩, `P`, in the route is replaced by the result of applying `PF` to `P`. Here are examples:

```
\insert 'PlacesMap.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'PlacesMapTest.oz $Revision: 1.1 $'}
{Test {PlacesMap route(nil) fun {$ P} place(where: nowhere features: nil) end}
 '==' route(nil)}
{Test {PlacesMap route([place(where: paris features: [wine cheese]) place(where: delhi features: [chai curry])])
       fun {$ P} place(where: nowhere features: nil) end}
 '==' route([place(where: nowhere features: nil) place(where: nowhere features: nil)])}
{Test {PlacesMap route([place(where: orlando features: [disney universal])
                        place(where: paris features: [wine cheese])
                        place(where: delhi features: [chai curry])])
       fun {$ P} case P of
                    place(where: paris features: _) then place(where: paris features: [degalle roissy])
                 [] place(where: delhi features: _) then place(where: delhi features: [gandhi])
                 else P
       end       end}
 '==' route([place(where: orlando features: [disney universal])
             place(where: paris features: [degalle roissy])
             place(where: delhi features: [gandhi])])}
{DoneTesting}
```

8. (20 points) [UseModels] This problem works with the type ⟨Order⟩, as defined by the following grammar (where a ⟨String⟩ is, as usual, a ⟨List Char⟩).

   ⟨Order⟩ ::= `order(`⟨List Item⟩`)`
   ⟨Item⟩ ::= `drink(what:` ⟨Atom⟩ `price:` ⟨Int⟩`)`
   | `food(what:` ⟨Atom⟩ `price:` ⟨Int⟩`)`
   | `combo(`⟨List Item⟩`)`

   In Oz, write a function

   `OrderCost: <fun {$ <Order>}: <Int>>`

   that takes an ⟨Order⟩, `Ord`, and return the sum of the cost of the items in the Order's list. The *cost* of an ⟨Item⟩ is the ⟨Int⟩ in the `price` field in a `drink` or `food` record, and the sum of the costs of the items in the list of items in a `combo` record.

```
\insert 'OrderCost.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'OrderCostTest.oz $Revision: 1.1 $'}
{Test {OrderCost order(nil)} '==' 0}
{Test {OrderCost order([drink(what: beer price: 3)])} '==' 3}
{Test {OrderCost order([drink(what: beer price: 3)
                        drink(what: coke price: 2)])} '==' 5}
{Test {OrderCost order([drink(what: pepsi price: 2)
                        food(what: burger price: 8)])} '==' 10}
{Test {OrderCost order([drink(what: water price: 2)
                        drink(what: tea price: 2)
                        drink(what: sprite price: 3)
                        food(what: coupon price: ~3)
                        food(what: salad price: 6)
                        food(what: steak price: 15)
                        combo([food(what: hsalad price: 3)
                               food(what: chop price: 10)])])}
 '==' 38}
{Test {OrderCost order([combo([combo([drink(what: water price: 0)
                                      drink(what: coffee price: 1)])
                               combo([food(what: blt price: 5)
                                      food(what: cuban price: 8)
                                      food(what: chips price: 0)])])])}
 '==' 14}
{Test {OrderCost order([drink(what: water price: 2)
                        combo([drink(what: wine75 price: 15)
                               food(what: foisgras price: 30)
                               drink(what: wine79 price: 14)
                               food(what: duck price: 35)
                               combo([food(what: wine24 price: 12)
                                      food(what: chocolatemoosse price: 10)])])
                        drink(what: coffee price: 3)])}
 '==' 121}
{DoneTesting}
```

   There is room for your answer on the next page.

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