Fall, 2011 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.
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 an iterative function in Oz

   `FindIndex: <`**`fun`**` {$ <List T> <T>}: <Int> >`

   that takes a list, `Ls`, of elements of some type `T`, and an element of type `T`, `Sought`, and returns the least integer, `I`, such that the `I`th element of `Ls` is equal to (using ==) `Sought`. If no element of `Ls` is equal to `Sought`, then `FindIndex` returns `~1`.

   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.)

   `FindIndex` counts indexes starting at 1, as you can see in the following examples.

   ```
   \insert 'FindIndex.oz'
   {StartTesting 'FindIndexTest'}
   {Test {FindIndex nil 7} '==' ~1}
   {Test {FindIndex [2] 7} '==' ~1}
   {Test {FindIndex [7] 7} '==' 1}
   {Test {FindIndex [7 2] 6} '==' ~1}
   {Test {FindIndex [9 8 1 2 7 6 6 6 6 1 9 2 10] 6} '==' 6}
   {Test {FindIndex [9 8 1 2 7 6 6 6 6 1 9 2 10] 9} '==' 1}
   {Test {FindIndex [9 8 1 2 7 6 6 6 6 1 9 2 10] 10} '==' 13}
   {Test {FindIndex [a c b x y z m q p f] x} '==' 4}
   {Test {FindIndex [b b c d e m] b} '==' 1}
   {DoneTesting}
   ```

2. (10 points) [UseModels] Without using `FoldR` or `Map`, write a function in Oz

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

   that takes a list of atoms, `Ls`, and an atom, `Title`, and returns a list of two-element lists, each of which contains the given `Title` followed by an atom from `Ls`. The order in the result corresponds to the order of `Ls`.

   The following are examples.

   ```
   \insert 'GiveTitles.oz'
   {StartTesting 'GiveTitleTest'}
   {Test {GiveTitles nil sir} '==' nil}
   {Test {GiveTitles [elton david henry] sir} '==' [[sir elton] [sir david] [sir henry]]}
   {Test {GiveTitles [aretha gaga di godiva] lady} '==' [[lady aretha] [lady gaga] [lady di] [lady godiva]]}
   {Test {GiveTitles [philips detroit evil wu] dr} '==' [[dr philips] [dr detroit] [dr evil] [dr wu]]}
   {Test {GiveTitles [a b c d e f g h i j k a] ms}
    '==' [[ms a] [ms b] [ms c] [ms d] [ms e] [ms f] [ms g] [ms h] [ms i] [ms j] [ms k] [ms a]]}
   {DoneTesting}
   ```

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

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

   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 {GiveTitles Ls Title}
       {FoldR









       }
   end
   ```

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

```
declare
fun {WeightedAverage W X Y}
   % Requires: 0.0 =< W and W =< 1.0
   (W*X) + ((1.0-W)*Y)
end
```

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

```
WeightedAverageCurried: <fun {$ <Float>}:
                                   <fun {$ <Float>}:
                                         <fun {$ <Float>}: <Float>>>
```

such that, `WeightedAvergeCurried` takes a Float `W`, and returns a function that takes as an argument, `X`, of type Float, and which returns a function that itself takes an argument, `Y`, of type Float, and returns `(W*X) + ((1.0-W)*Y)`.

The following are examples.

```
\insert 'WeightedAverageCurried.oz'
{StartTesting 'WeightedAverageCurriedTest $Revision: 1.1 $'}
{Test {{{WeightedAverageCurried 0.5} 4.0} 4.0} '==' 4.0}
{Test {{{WeightedAverageCurried 0.25} 10.0} 20.0} '==' 17.5}
{Test {{{WeightedAverageCurried 0.35} 0.9} 0.8} '==' 0.835}
{Test {{{WeightedAverageCurried 0.0} 0.9} 0.8} '==' 0.8}
{Test {{{WeightedAverageCurried 1.0} 0.9} 0.8} '==' 0.9}
{Test {{{WeightedAverageCurried 0.65} 0.9} 0.8} '==' 0.865}
{DoneTesting}
```

Please write your answer below.

5. (10 points) [UseModels] Consider the following grammar (where Lists and Atoms are standard).

   $\langle$Text$\rangle$ ::= $\langle$List Paragraph$\rangle$
   $\langle$Paragraph$\rangle$ ::= `para(`$\langle$List Atom$\rangle$`)`

   In Oz, write a function, `NumWords: <`**`fun`**` {$ <Text>}: <Int>>` which takes a Text, `Txt`, and returns the total number of atoms in it. The following are examples.

```
\insert 'NumWords.oz'
{StartTesting 'NumWords'}
{Test {NumWords nil} '==' 0}
{Test {NumWords [para([and along came java]) para([cpp was miffed with java])]}  '==' 9}
{Test {NumWords [para([dum dum dum dum]) para([dum bum]) para([rum scum])]} '==' 8}
{Test {NumWords [para([it was the best 'of' times it was the worst 'of' times])]} '==' 12}
{DoneTesting}
```

6. (15 points) [UseModels] Using the same grammar as in the previous question,

   ⟨Text⟩ ::= ⟨List Paragraph⟩<br>
   ⟨Paragraph⟩ ::= `para(`⟨List Atom⟩`)`

   in Oz, write a function, `SubstAll: <`**`fun`**` {$ <Text> <Atom> <Atom>}: <Text>>` which takes a Text, `Txt`, and two atoms `Old` and `New`, and returns a Text that is just like `Txt`, except that each occurrence of the atom `Old` is replaced by the value of `New`. The following are examples.

```
\insert 'SubstAll.oz'
{StartTesting 'SubstAll'}
{Test {SubstAll nil java csharp} '==' nil}
{Test {SubstAll [para([and along came java]) para([cpp was miffed with java])]
                java csharp}
  '==' [para([and along came csharp]) para([cpp was miffed with csharp])]}
{Test {SubstAll [para([dum dum dum dum]) para([dum bum]) para([rum scum])]
                dum ah}
 '==' [para([ah ah ah ah]) para([ah bum]) para([rum scum])]}
{Test {SubstAll [para([it was the best 'of' times it was the worst 'of' times])]
                times tests}
 '==' [para([it was the best 'of' tests it was the worst 'of' tests])]}
{DoneTesting}
```

7. (15 points) [UseModels] In Oz, write the function

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

that, for some types `T` and `S`, takes a list, `Funs`, of functions (each of type `<fun {$ T}: S>`) and a list, `Args`, of values (each of type `T`), which has the same length as `Funs`, and returns a list that contains the results of applying the $i$th function in `Funs` to the $i$th argument in `Args`. The resulting list preserves the order of the original lists. You should assume that the lists `Funs` and `Args` have equal lengths. The following are examples.

```
\insert 'ListApply.oz'
{StartTesting 'ListApply'}
{Test {ListApply nil nil} '==' nil}
{Test {ListApply [fun {$ X} X+1001 end] [4020]} '==' [5021]}
{Test {ListApply [fun {$ X} X+1 end fun {$ X} X+2 end] [10 20]} '==' [11 22]}
{Test {ListApply
       local AddC = fun {$ Y} fun {$ X} X+Y end end in
          {Map [1 2 3 4 5 2 27 999 6] AddC}
       end
       [10 20 30 40 50 60 70 1000 10]}
 '==' [11 22 33 44 55 62 97 1999 16]}
{Test {ListApply [fun {$ X} bread#X#bread end fun {$ X} pita#X#pita end]
      [turkey humus]}
 '==' [bread#turkey#bread pita#humus#pita]}
{DoneTesting}
```

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

   ⟨Entry⟩ ::= directory(⟨AList⟩) | file(⟨String⟩)
   ⟨AList⟩ ::= ⟨List ⟨#-Pair Atom Entry⟩⟩
   ⟨#-Pair Atom Entry⟩ ::= ⟨Atom⟩#⟨Entry⟩

   In Oz, write a function

   `Retrieve: <`**`fun`**` {$ <Entry> <List Atom>}: <Entry>>`

   that takes an ⟨Entry⟩, `Ent`, and a list of atoms, `Path`, and returns the ⟨Entry⟩ that corresponds to the given `Path` in `Ent`. An empty `Path` argument names `Ent` itself. If `Ent` is a `file` record, then the `Path` can only be `nil`, otherwise `Retrieve` throws an exception. If `Ent` is a `directory` record, then each element of `Path` names the ⟨Entry⟩ associated to it by the association list (AList). When `Ent` is a directory, the first element of `Path` names an ⟨Entry⟩ in the ⟨AList⟩ inside `Ent`. Subsequent elements in `Path` similarly are associated to ⟨Entry⟩ values by association lists in directories within the ⟨Entry⟩ value associated to the previous elements in `Path`. For example, in the test of `{Retrieve Users [mydir mine]}` below, the atom `mydir` in the `Path` argument `[mydir mine]` names the directory (`MyD`) that holds the file record named `mine`.

```
\insert 'Retrieve.oz'
{StartTesting 'RetrieveTest'}
{Test {Retrieve file("only needs to work on a file when the path is nil") nil}
 '==' file("only needs to work on a file when the path is nil")}
{Test try _ = {Retrieve file("otherwise throw an exception") [some path]}
         false
      catch _ then true end % returns true if expected exception is caught
 '==' true}
local MyD = directory([mine#file("my file 1") metoo#file("my file 2")]) in
   {Test {Retrieve MyD nil} '==' MyD}
   {Test {Retrieve MyD [mine]} '==' file("my file 1")}
   {Test {Retrieve MyD [metoo]} '==' file("my file 2")}
   {Test try _ = {Retrieve MyD [notfound]}
            false
         catch _ then true end  % returns true if expected exception caught
    '==' true}
   local Users = directory([mydir#MyD
                            yourdir#directory([yours#file("your file 1")
                                               youtoo#file("your file 2")])]) in
      {Test {Retrieve Users [mydir]} '==' MyD}
      {Test {Retrieve Users [mydir mine]} '==' file("my file 1")}   % This test is described above
      {Test {Retrieve Users [mydir metoo]} '==' file("my file 2")}
      {Test {Retrieve Users [yourdir yours]} '==' file("your file 1")}
      {Test {Retrieve Users [yourdir youtoo]} '==' file("your file 2")}
      local Home = directory([users#Users
                              lib#directory([cpp#file("STL")
                                             c#file("stdio")])]) in
         {Test {Retrieve Home [users mydir mine]} '==' file("my file 1")}
         {Test {Retrieve Home [users yourdir yours]} '==' file("your file 1")}
         {Test {Retrieve Home [lib cpp]} '==' file("STL")}
         {Test {Retrieve Home [users]} '==' Users}
      end
   end
end
{DoneTesting}
```

   There is room for your answer on the next page.

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