Spring, 2009 Name: _____

COP 4020 — Programming Languages 1

Test on Declarative Programming Techniques

Special Directions for this Test

This test has 6 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 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 like `Append`, `Member`, `Max`, `Filter`, `Map`, and `FoldR`.

For Grading

Question:	1	2	3	4	5	6	Total
Points:	10	10	15	15	30	20	100
Score:							

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

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

   that takes a list `Lst` of values of some type T and a value `Sought` of type T, and returns number of values in `Lst` that are equal to `Sought` (using `==`).

   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.

   ```
   \insert 'Count.oz'
   {Test {Count nil anatom} '==' 0}
   {Test {Count [a b b a a c] a} '==' 3}
   {Test {Count [a b b a a c] b} '==' 2}
   {Test {Count [a b b a a c] c} '==' 1}
   {Test {Count [u c f rules] c} '==' 1}
   {Test {Count [u c f rules] u} '==' 1}
   {Test {Count [u c f r u l e s] u} '==' 2}
   {Test {Count [a fine mess have you a knife] a} '==' 2}
   {Test {Count [4020 541 342 342 4020 641] 4020} '==' 2}
   {Test {Count [4020 541 342 342 4020 641] 1} '==' 0}
   {Test {Count [nil [3]] [3]} '==' 1}
   ```

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

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

that takes a list of some type T, `Elems`, and produces a list of singleton lists containing each element of `Elems`. That is, the result has each element of `Elems` in their original order, but each is put in a 1-element list. The following are examples.

```
\insert 'Down.oz'
{Test {Down nil} '==' nil}
{Test {Down [9 8 7 6 5 4 3 2 1 0]}
      '==' [[9] [8] [7] [6] [5] [4] [3] [2] [1] [0]]}
{Test {Down [shuttle launched tonight]} '==' [[shuttle] [launched] [tonight]]}
{Test {Down [da yes si oui yes ok ja yes]}
      '==' [[da] [yes] [si] [oui] [yes] [ok] [ja] [yes]]}
{Test {Down [1 2 3 4 2]} '==' [[1] [2] [3] [4] [2]]}
{Test {Down {Down [1 2 3 4 2]}} '==' [[[1]] [[2]] [[3]] [[4]] [[2]]]}
```

3. (15 points) [UseModels] Write a higher-order function

```
Map2: <fun {$ <List T> <List T> <fun {$ T T}: S>} : <List S>>
```

that takes two lists, `Ls1` and `Ls2`, and a function `F`, and returns a list that is the same length as the shortest of `Ls1` and `Ls2`, and in which the i^{th} element is the result of applying `F` to the i^{th} elements of `Ls1` and `Ls2` (in that order). The following are examples.

```
\insert 'Map2.oz'
{Test {Map2 [7 4 3] [1 2 3] fun {$ M N} M-N end} '==' [6 2 0]}
local
   fun {Radix10 X Y} X*10+Y end
in
   {Test {Map2 nil nil Radix10} '==' nil}
   {Test {Map2 [1 2] [1 2] Radix10} '==' [11 22]}
   {Test {Map2 [4 0 2 0] [4 7 5 8] Radix10} '==' [44 7 25 8]}
   {Test {Map2 [4] [4 7 5 8] Radix10} '==' [44]}
   {Test {Map2 [6 4 0 2 0] [7 5 8] Radix10} '==' [67 45 8]}
end
{Test {Map2 [a good egg] [is over easy] fun {$ A B} A#B end}
 '==' [a#is good#over egg#easy]}
{Test {Map2 [richard cindy al freddie margo] [uk us de uk fr]
       fun {$ Name Country} [Name Country] end}
 '==' [[richard uk] [cindy us] [al de] [freddie uk] [margo fr]]}
```

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

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

   that for some type `T` takes a list of values of type `T`, `Lst`, and returns a list that is just like `Lst` except that it has no duplicates. That is, each element only appears in the result if there are no elements that are == to it farther along the list (towards the right, or end of the list). The following are examples.

   ```
   \insert 'NoDuplicates.oz'
   {Test {NoDuplicates nil} '==' nil}
   {Test {NoDuplicates [b a]} '==' [b a]}
   {Test {NoDuplicates [b b a]} '==' [b a]}
   {Test {NoDuplicates [a b b a]} '==' [b a]}
   {Test {NoDuplicates [b d a c a b b a]} '==' [d c b a]}
   {Test {NoDuplicates [um no no no no yes no yes no]} '==' [um yes no]}
   {Test {NoDuplicates [4 0 7 5 5 5 1 2 1 2]} '==' [4 0 7 5 1 2]}
   ```

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

   ```
   fun {NoDuplicates Lst}
      {FoldR




      }
   end
   ```

5. (30 points) [UseModels] This problem is about the following grammar for "sales data":

   ⟨SalesData⟩ ::=
   `store(address:` ⟨String⟩ `amounts:` ⟨List ⟨Int⟩⟩`)`
   | `group(name:` ⟨String⟩ `members:` ⟨List ⟨SalesData⟩⟩`)`

   Write a function `MaxAmounts: <fun {$ <SalesData>}: <SalesData>>` that takes a ⟨SalesData⟩, `Data`, and returns a ⟨SalesData⟩ that is just like `Data`, except that in each `store` record, the list of amounts (in the field `amounts`) is replaced by its maximum. Assume that each store record in `Data` has a list of amounts that is non-nil and that each amount is positive. The following are examples using the `Test` function from the homework.

```
\insert 'MaxAmounts.oz'
{Test {MaxAmounts store(address: "The Bad One" amounts: [1])}
 '==' store(address: "The Bad One" amounts: [1])}
{Test {MaxAmounts store(address: "Waterford Lakes" amounts: [5 18 999 13 5])}
 '==' store(address: "Waterford Lakes" amounts: [999])}
{Test {MaxAmounts store(address: "Oviedo" amounts: [7 3 7 5 19 8 19 1])}
 '==' store(address: "Oviedo" amounts: [19])}
{Test {MaxAmounts
       group(name: "Central Florida"
             members: [store(address: "The Bad One" amounts: [1])
                       store(address: "Waterford Lakes" amounts: [5 18 999 13 5])
                       store(address: "Oviedo" amounts: [7 3 7 5 19 8 19 1])])}
 '==' group(name: "Central Florida"
            members: [store(address: "The Bad One" amounts: [1])
                      store(address: "Waterford Lakes" amounts: [999])
                      store(address: "Oviedo" amounts: [19])])}
```

6. (20 points) [UseModels] This problem is about the following grammar for "rules":

⟨Rule T S⟩ ::=
`baseRule(<fun {$ T}: S>)`
| `extension(arg: T value: S otw:` ⟨Rule T S⟩`)`

Write a function `ApplyRule: <fun {$ <Rule T S> T}: S>` that takes a ⟨Rule T S⟩, `Rule`, and a value of type T, `Val`, and returns value of type S, by using the functionality of `Rule`. Applying a rule of the form `baseRule(F)` to some value `Val` yields the value returned by the application `{F Val}`; that is the rule given by a base rule is the algorithm computed by the function found in the `baseRule` record. Applying a rule of the form `extension(arg:X value:Y otw:SubRule)` to some value `Val` yields `Y` if `X == Val`, and otherwise is the value of applying `SubRule` to `Val`. The following are examples using the `Test` function from the homework.

```
\insert 'ApplyRule.oz'
{Test {ApplyRule baseRule(fun {$ T} T+1 end) 4019 } '==' 4020}
{Test {ApplyRule baseRule(fun {$ X} X*10+5 end) 6 } '==' 65}
{Test {ApplyRule baseRule(fun {$ X} [X on average] end) good} '==' [good on average]}
{Test {ApplyRule extension(arg: good value: super otw: baseRule(fun {$ X} [X on average] end)) good}
 '==' super}
{Test {ApplyRule
       extension(arg: good value: great
                 otw: extension(arg: good value: super otw: baseRule(fun {$ X} [X on average] end)))
       good}
 '==' great}
{Test {ApplyRule
       extension(arg: good value: great
                 otw: extension(arg: good value: super otw: baseRule(fun {$ X} [X on average] end)))
       poor}
 '==' [poor on average]}
```