Fall, 2007 Name: ______________________________

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 imperative features (such as cells and assignment).

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.

For Grading

Problem	Points	Score
1	5	
2	5	
3	10	
4	10	
5	10	
6	10	
7	25	
8	25	

1. (5 points) In the following statement, written in the kernel language of the declarative model, circle each free variable identifier occurrence (and only the free variable identifier occurrences).

```
MyFun = proc {$ Work Elems Ret}
            local Z in
               local MyFun in
                 MyFun = proc {$ X R} Y={Times X X} end
                 {FoldR Elems MyFun Z}
                 {Apply Work Z Ret}
               end
            end
         end
```

2. (5 points) The following is another copy of the same kernel code as above. On the following copy, circle each bound variable identifier occurrence (and only the bound variable identifier occurrences).

```
MyFun = proc {$ Work Elems Ret}
            local Z in
               local MyFun in
                 MyFun = proc {$ X R} Y={Times X X} end
                 {FoldR Elems MyFun Z}
                 {Apply Work Z Ret}
               end
            end
         end
```

3. (10 points) Desugar the following Oz code into kernel syntax by expanding all syntactic sugars. (Assume that `Plus` and `Times` are functions declared elsewhere.)

```
fun {Next A I}
   J = {Plus I 4}
in
   {A {Times J J}}
end
```

4. (10 points) Write a function

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

that takes a list of atoms `Teams` and produces a list of lists of atoms. Each of the inner lists in the result is a two-element list with the atom `beat` preceding the corresponding element of the argument list. The following are examples, that use the `Test` method from the homework.

```
{Test {TeamSpirit nil} '==' nil}
{Test {TeamSpirit [ncstate]} '==' [[beat ncstate]]}
{Test {TeamSpirit [texas ncstate]} '==' [[beat texas] [beat ncstate]]}
{Test {TeamSpirit [louisiana texas ncstate]}
      '==' [[beat louisiana] [beat texas] [beat ncstate]]}
{Test {TeamSpirit [smiss tulsa usf ecarolina louisiana texas ncstate]}
 '==' [[beat smiss] [beat tulsa] [beat usf] [beat ecarolina]
       [beat louisiana] [beat texas] [beat ncstate]]}
```

5. (10 points) Write a function

   ```
   MinElement: <fun {$ <List Number>}: <Number>
   ```

   that takes a list of numbers and returns the smallest number in the argument list.

   Your solution must have iterative behavior, and must be written using tail recursion. Don't use any higher-order functions in your solution. (You are supposed to know what these directions mean.)

   Assume that the argument list has at least one element, since otherwise the minimum element is not defined.

   The following are examples, that use the `Test` method from the homework.

   ```
   {Test {MinElement [3]} '==' 3}
   {Test {MinElement [4020]} '==' 4020}
   {Test {MinElement [7 19]} '==' 7}
   {Test {MinElement [3 7 19]} '==' 3}
   {Test {MinElement [54 3 7 19]} '==' 3}
   {Test {MinElement [78 7 654 7 5 54 3 7 19]} '==' 3}
   {Test {MinElement [~6 78 7 654 7 5 54 3 7 19]} '==' ~6}
   ```

   Hint: you can use Oz's built-in `Min` function, which returns the minimum of its two arguments.

6. (10 points) Write a definition of `MinElement` that solves the problem given above, but using `FoldR`. Do this by filling in the blanks in the following code outline. Note that these blanks may be larger in size than you need.

   (Again, assume that the argument list has at least one element. And you can again can use Oz's `Min` function.)

   ```
   fun {MinElement ____________________}
     {FoldR __________________________________________________________
            __________________________________________________________
            __________________________________________________________
     }
   end
   ```

7. (25 points) This problem is about "window layouts." Window layouts are defined by the following grammar, which you have seen previously in the "Following the Grammar" handout and the homework.

```
⟨WindowLayout⟩ ::=
     window(name: ⟨Atom⟩ width: ⟨Number⟩ height: ⟨Number⟩)
   | horizontal(⟨List WindowLayout⟩)
   | vertical(⟨List WindowLayout⟩)
```

Write a function

`ChangeChannel: <`**`fun`**` {$ <WindowLayout> <Atom> <Atom>}: <WindowLayout>`

that takes a window layout `WL`, two atoms `New` and `Old`, and returns a window layout that is just like `WL` except that all windows whose name field's value is (== to) `Old` in the argument `WL` are changed to `New` in the result.

The following are examples using the `Test` function from the homework.

```
{Test {ChangeChannel vertical(nil) cnn simpsons} '==' vertical(nil)}
{Test {ChangeChannel horizontal(nil) cnn simpsons} '==' horizontal(nil)}
{Test {ChangeChannel window(name: simpsons width: 30 height: 40)
      cnn simpsons}
                '==' window(name: cnn width: 30 height: 40)}
{Test {ChangeChannel
       horizontal([window(name: simpsons width: 30 height: 40)])
       simpsons snl}
  '==' horizontal([window(name: simpsons width: 30 height: 40)])}
{Test {ChangeChannel
       vertical([window(name: snl width: 90 height: 50)
                 window(name: snl width: 180 height: 120)])
       futurama snl}
 '=='  vertical([window(name: futurama width: 90 height: 50)
                window(name: futurama width: 180 height: 120)])}
{Test {ChangeChannel
       horizontal([window(name: cbs width: 30 height: 15)
                   vertical([window(name: cnn width: 89 height: 55)
                             window(name: cbs width: 101
                                    height: 45)])
                   horizontal([window(name: cbs width: 92
                                      height: 150)])])
       dailyshow cbs}
  '==' horizontal([window(name: dailyshow width: 30 height: 15)
                   vertical([window(name: cnn width: 89 height: 55)
                             window(name: dailyshow width: 101
                                    height: 45)])
                   horizontal([window(name: dailyshow width: 92
                                      height: 150)])
                   ])}
 {Test {ChangeChannel
        vertical(
           [vertical([window(name: simpsons width: 30 height: 40)])
            horizontal([horizontal([window(name: news width: 5
                                           height: 5)])])
            horizontal([window(name: simpsons width: 30 height: 15)
```

There is space for your answer on the next page.

Put your answer to the `ChangeChannel` problem here.

8. (25 points) This problem is about “expressions” defined by the following grammar.

```
⟨Expression⟩ ::=
      varIdExp(⟨String⟩)
    | numExp(⟨Number⟩)
    | plusExp(⟨Expression⟩ ⟨Expression⟩)
    | timesExp(⟨Expression⟩ ⟨Expression⟩)
```

Write a function

```
EvalExp: <fun {$ <Expression> <fun {$ <String>}: <Number>>}: <Number>
```

that takes an expression `Exp` and a function from Strings to Numbers, `Env`. Assume that `Env` is defined on each String that occurs in a ⟨varIdExp⟩ inside `Exp`. The function `Eval` evaluates the expression `Exp`, using `Env` to determine the values of all subexpressions of the form `varIdExp("Name")`.

The following are examples using the `Test` function from the homework.

```
declare
fun {StdEnv Str}
   case Str of
      "One" then 1
   [] "Two" then 2
   [] "Ten" then 10
   [] "Z" then 0
   else raise stdEnvIsUndefinedOn(Str) end
   end
end

{Test {EvalExp varIdExp("Two") StdEnv} '==' 2}
{Test {EvalExp varIdExp("One") StdEnv} '==' 1}
{Test {EvalExp varIdExp("One") fun {$ N} 7 end} '==' 7}
{Test {EvalExp numExp(20) StdEnv} '==' 20}
{Test {EvalExp plusExp(varIdExp("Two") numExp(20)) StdEnv} '==' 22}
{Test {EvalExp timesExp(varIdExp("Two") numExp(20)) StdEnv} '==' 40}

{Test {EvalExp timesExp(numExp(7) plusExp(varIdExp("Ten") numExp(1)))
       StdEnv}
 '==' 77}

{Test
 {EvalExp
  plusExp(timesExp(varIdExp("Two") plusExp(varIdExp("Ten") varIdExp("Z")))
          plusExp(varIdExp("Two") varIdExp("One")))
  StdEnv}
  '==' 23}

{Test
 {EvalExp
  plusExp(
     plusExp(timesExp(varIdExp("Two") plusExp(varIdExp("Ten") varIdExp("Z")))
             plusExp(varIdExp("Two") varIdExp("One")))
     timesExp(numExp(7) plusExp(varIdExp("Ten") numExp(1))))
  StdEnv}
 '==' 100}
```

There is space for your answer on the next page.

Put your answer to the `EvalExp` problem here.