Spring, 2008 Name: ______________________

COP 4020 — Programming Languages 1

Test on Declarative Programming Techniques

Special Directions for this Test

This test has 5 questions and pages numbered 1 through 8.

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) or the library functions `IsDet` and `IsFree`.

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	20	
2	10	
3	20	
4	25	
5	25	

1. (20 points) [UseModels] Write a function

   ```
   SumValues: <fun {$ <fun {$ Int}: Int> Int Int}: Int>
   ```

   that takes a function `F`, and two integers `LB` and `UB`, such that `LB < UB`, and which returns the sum of `{F I}` for all `I` between `LB` and `UB` (inclusive).

   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` method from the homework.

   ```
   local
      fun {MyF I} 10*I end
   in
      {Test {SumValues MyF 1 3} '==' 60}
   end
   {Test {SumValues fun {$ I} 3 end 1 10} '==' 30}
   {Test {SumValues fun {$ I} I end 1 10} '==' 55}
   {Test {SumValues fun {$ I} I end 2 10} '==' 54}
   {Test {SumValues fun {$ I} vals(3 7 9 2).I end 1 4} '==' 21}
   {Test {SumValues fun {$ I} I*I+1 end 2 10} '==' 393}
   ```

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

   ```
   VoteFor: <fun {$ <List Atom>}: <List <Pair Atom Atom>>
   ```

   that takes a list of atoms, `Candidates`, and produces a list of pairs of atoms. Each of the pairs has the atom `vote` as its first element and the corresponding element of the argument list as its second element. The following are examples, that use the `Test` method from the homework.

   ```
   {Test {VoteFor nil} '==' nil}
   {Test {VoteFor [clinton obama mccain]}
    '==' [vote#clinton vote#obama vote#mccain]}
   {Test {VoteFor [john hillary barak ralph]}
    '==' [vote#john vote#hillary vote#barak vote#ralph]}
   {Test {VoteFor [uptown downtown midtown motown funkytown]}
    '==' [vote#uptown vote#downtown vote#midtown vote#motown vote#funkytown]}
   ```

3. (20 points) [UseModels] Write a function

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

   that takes a list of integers `Nums` and produces a list that contains just the strictly positive elements of `Nums`, in their original order. The following are examples, that use the `Test` method from the homework. (Note that `~3` is the Oz way of writing negative numbers, such as -3.)

   ```
   {Test {Positive nil} '==' nil}
   {Test {Positive [~1]} '==' nil}
   {Test {Positive [3 7 ~1]} '==' [3 7]}
   {Test {Positive [~3 3 7 ~1]} '==' [3 7]}
   {Test {Positive [~2 0 5 ~3 3 7 ~1]} '==' [5 3 7]}
   {Test {Positive [0 5 ~3 3 7 ~1]} '==' [5 3 7]}
   ```

4. (25 points) [UseModels] This problem is about the following "statement and expression" grammar, which you have seen previously in the "Following the Grammar" handout and the homework.

```
⟨Statement⟩ ::=
      expStmt(⟨Expression⟩)
    | assignStmt(⟨Atom⟩ ⟨Expression⟩)
    | ifStmt(⟨Expression⟩ ⟨Statement⟩)
⟨expression⟩ ::=
      varExp(⟨Atom⟩)
    | numExp(⟨Number⟩)
    | equalsExp(⟨Expression⟩ ⟨Expression⟩)
    | beginExp(⟨List Statement⟩ ⟨Expression⟩)
```

Write a function

```
NegateIfs: <fun {$ <Statement>}: <Statement>
```

that takes a statement `Stmt`, and returns a statement that is just like `Stmt` except that all `ifStmt` statements of the form `ifStmt(`E S`)` that occur anywhere within `Stmt` are replaced by `ifStmt(equalsExp(`E `varExp(false))` S`)`. This process occurs recursively for all subparts of `Stmt`, even within E and S. The following are examples using the `Test` function from the homework.

```
{Test {NegateIfs expStmt(numExp(3))} '==' expStmt(numExp(3))}
{Test {NegateIfs expStmt(varExp(y))} '==' expStmt(varExp(y))}
{Test {NegateIfs expStmt(equalsExp(varExp(y) varExp(z)))}
 '==' expStmt(equalsExp(varExp(y) varExp(z)))}
{Test {NegateIfs assignStmt(x numExp(3))} '==' assignStmt(x numExp(3))}
{Test {NegateIfs ifStmt(varExp(true)
                        assignStmt(x numExp(3)))}
 '==' ifStmt(equalsExp(varExp(true) varExp(false))
             assignStmt(x numExp(3)))}
{Test {NegateIfs expStmt(beginExp(nil numExp(3)))}
 '==' expStmt(beginExp(nil numExp(3)))}
{Test {NegateIfs
       expStmt(beginExp([ifStmt(varExp(true)
                                assignStmt(x numExp(3)))
                         assignStmt(y numExp(4))]
                        varExp(y)))}
 '==' expStmt(beginExp([ifStmt(equalsExp(varExp(true) varExp(false))
                                      assignStmt(x numExp(3)))
                               assignStmt(y numExp(4))]
                              varExp(y)))}
{Test {NegateIfs
       ifStmt(beginExp([ifStmt(varExp(true)
                               assignStmt(x numExp(3)))
                        assignStmt(y numExp(4))]
                       varExp(y))
              assignStmt(q beginExp([ifStmt(varExp(m)
                                           expStmt(numExp(7)))]
                                    varExp(m))))}
 '==' ifStmt(equalsExp(beginExp([ifStmt(equalsExp(varExp(true) varExp(false))
                                               assignStmt(x numExp(3)))
                                        assignStmt(y numExp(4))]
                                       varExp(y))
                      varExp(false))
             assignStmt(q beginExp([ifStmt(equalsExp(varExp(m) varExp(false))
                                                  expStmt(numExp(7)))]
                                    varExp(m))))}
```

There is space for your answer on the next page.

Put your answer to the `NegateIfs` problem here.

5. (25 points) [UseModels] This problem is about "music" defined by the following grammar.

```
⟨Music⟩ ::=
     pitch(⟨Number⟩)
   | chord(⟨List Music⟩)
   | sequence(⟨List Music⟩)
```

Write a function

```
Transpose: <fun {$ <Music> <Number>}: <Music>
```

that takes a music value, `Song`, and a number, `Delta`, and produces a music value that is just like `Song`, but in which each number has been replaced by that number plus `Delta`. (This is what musicians call transposition, hence the name.) The following are examples using the `Test` function from the homework.

```
{Test {Transpose pitch(3) 7} '==' pitch(10)}
{Test {Transpose pitch(10) 5} '==' pitch(15)}
{Test {Transpose chord(nil) ~3} '==' chord(nil)}
{Test {Transpose chord([pitch(1) pitch(5) pitch(8)]) 2}
 '==' chord([pitch(3) pitch(7) pitch(10)])}
{Test {Transpose sequence(nil) ~1} '==' sequence(nil)}
{Test {Transpose sequence([pitch(1) pitch(5) pitch(8)]) 2}
 '==' sequence([pitch(3) pitch(7) pitch(10)])}
{Test {Transpose
       sequence([chord([pitch(1) pitch(5) pitch(8)])
                 chord([pitch(3) pitch(7) pitch(0)])
                 chord([pitch(7) pitch(5) pitch(9)])])
       1}
 '==' sequence([chord([pitch(2) pitch(6) pitch(9)])
                 chord([pitch(4) pitch(8) pitch(1)])
                 chord([pitch(8) pitch(6) pitch(10)])])}
{Test {Transpose
       chord([sequence([chord([pitch(1) pitch(5) pitch(8)])
                        chord([pitch(3) pitch(7) pitch(0)])
                        chord([pitch(7) pitch(5) pitch(9)])])
              sequence([pitch(1) pitch(1)])
              chord([sequence(nil) sequence([pitch(3)])])])
       1}
 '==' chord([sequence([chord([pitch(2) pitch(6) pitch(9)])
                       chord([pitch(4) pitch(8) pitch(1)])
                       chord([pitch(8) pitch(6) pitch(10)])])
             sequence([pitch(2) pitch(2)])
             chord([sequence(nil) sequence([pitch(4)])])])}
{Test {Transpose
       sequence([chord([sequence([chord([pitch(1) pitch(5) pitch(8)])
                                  chord([pitch(3) pitch(7) pitch(0)])
                                  chord([pitch(7) pitch(5) pitch(9)])])
                        sequence([pitch(1) pitch(1)])
                        chord([sequence(nil) sequence([pitch(3)])])])
                 chord([pitch(1) pitch(9)])])
       1}
 '==' sequence([chord([sequence([chord([pitch(2) pitch(6) pitch(9)])
                                 chord([pitch(4) pitch(8) pitch(1)])
                                 chord([pitch(8) pitch(6) pitch(10)])])
                       sequence([pitch(2) pitch(2)])
                       chord([sequence(nil) sequence([pitch(4)])])])
                chord([pitch(2) pitch(10)])])}
```

There is space for your answer on the next page.

Put your answer to the `Transpose` problem here.