Fall, 2007 Name: _____

COP 4020 — Programming Languages 1

Makeup 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. (This means you can use things in the Oz base environment such as `Map`, `FoldR`, `Filter`, `Append`, etc.)

For Grading

Problem	Points	Score
1	5	
2	10	
3	5	
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).

```
ProdFuns = proc {$ Ls Acc Return}
             case Ls of
             Hd | Tail then local Temp in
                               Temp = proc {$ X R} {Prod Hd X R} end
                               local Temp2 in
                                  Temp2 = Temp | Acc
                                  {ProdFuns Tail Temp2 Return}
                               end
                             end
             else {Reverse Acc Return}
             end
           end
```

2. (10 points) The following is another copy of the same kernel language code as above. On the following copy, circle each bound variable identifier occurrence (and only the bound variable identifier occurrences).

```
ProdFuns = proc {$ Ls Acc Return}
             case Ls of
             Hd | Tail then local Temp in
                               Temp = proc {$ X R} {Prod Hd X R} end
                               local Temp2 in
                                  Temp2 = Temp | Acc
                                  {ProdFuns Tail Temp2 Return}
                               end
                             end
             else {Reverse Acc Return}
             end
           end
```

3. (5 points) Desugar the following Oz code into kernel syntax by expanding all syntactic sugars. (Assume that `Times` is a function that is declared elsewhere.)

```
fun {By2 I}
   {Times I 2}
end
```

4. (10 points) Write a function

   ```
   EvenNumbers: <fun {$ <List Integer>}: <List Integer>
   ```

   that takes a list of integers `LoN` and returns a list of the even integers in `LoN`, in their original order. The following are examples, that use the `Test` procedure from the homework.

   ```
   {Test {EvenNumbers nil} '==' nil}
   {Test {EvenNumbers [2]} '==' [2]}
   {Test {EvenNumbers [3]} '==' nil}
   {Test {EvenNumbers [1 2 3 4 5 6 7 6 2 3 1]} '==' [2 4 6 6 2]}
   {Test {EvenNumbers [2 3 4 5 6 7 6 2 3 1]} '==' [2 4 6 6 2]}
   {Test {EvenNumbers [3 4 5 6 7 6 2 3 1]} '==' [4 6 6 2]}
   ```

5. (10 points) Write a function

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

   that takes a list of numbers, `LoN`, and returns the sum of their squares.

   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.)

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

   ```
   {Test {SumSquares nil} '==' 0}
   {Test {SumSquares [5]} '==' 25}
   {Test {SumSquares [5 10]} '==' 125}
   {Test {SumSquares [1 2 3 4]} '==' 30}
   {Test {SumSquares [7 5 10 12 ~7 ~5 ~10 ~12]} '==' 636}
   {Test {SumSquares [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]} '==' 0}
   ```

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

   ```
   fun {SumSquares LoN}
     {FoldL _____________________________________________________________
            _____________________________________________________________
            _____________________________________________________________
     }
   end
   ```

7. (25 points) This problem is about "sales data records." Sales data records are defined by the following grammar.

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

Write a function

```
NewAddress: <fun {$ <SalesData> <String> <String>}: <SalesData>
```

that takes a sales data record `SD`, two strings `New` and `Old`, and returns a sales data record that is just like `SD` except that all store records in `SD` whose address field's value is (== to) `Old` in `SD` are changed to `New` in the result.

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

```
{Test {NewAddress group(name: "StartUP!" members: nil)
       "Downtown" "50 Washington Ave."}
 '==' group(name: "StartUP!" members: nil)}
{Test {NewAddress store(address: "The Mall" amounts: [10 32 55])
      "110 Main St." "The Mall"}
 '==' store(address: "110 Main St." amounts: [10 32 55])}
{Test {NewAddress
       group(name: "Target"
             members: [store(address: "The Mall" amounts: [10 32 55])])
       "NewAddress" "OldAddress"}
 '==' group(name: "Target"
            members: [store(address: "The Mall" amounts: [10 32 55])])}
{Test {NewAddress
       group(name: "Target"
             members: [store(address: "The Mall" amounts: [10 32 55])
                       store(address: "Downtown" amounts: [4 0 2 0])])
       "253 Sears Tower" "The Mall"}
 '==' group(name: "Target"
            members: [store(address: "253 Sears Tower" amounts: [10 32 55])
                      store(address: "Downtown" amounts: [4 0 2 0])])}
{Test {NewAddress
       group(name: "ACME"
             members:
                [group(name: "Robucks"
                       members: [store(address: "The Mall" amounts: [99])
                                 store(address: "Maple St." amounts: [32])])
                 group(name: "Target"
                       members: [store(address: "The Mall" amounts: [10 55])
                                 store(address: "Downtown" amounts: [4])])])
       "High St." "The Mall"}
 '==' group(name: "ACME"
             members:
                [group(name: "Robucks"
                       members: [store(address: "High St." amounts: [99])
                                 store(address: "Maple St." amounts: [32])])
                 group(name: "Target"
                       members: [store(address: "High St." amounts: [10 55])
                                 store(address: "Downtown" amounts: [4])])])}
```

There is space for your answer on the next page.

Put your answer to the `NewAddress` problem here.

8. (25 points) This problem is about "expressions" (that encode the abstract syntax of a subset of Oz that corresponds to the one-argument "λ-calculus").

```
⟨Expression⟩ ::=
     varIdExp(⟨Atom⟩)
   | funExp(⟨Atom⟩ ⟨Expression⟩)
   | applyExp(⟨Expression⟩ ⟨Expression⟩)
```

Write a function

```
FreeVarIds: <fun {$ <Expression>}: <List <Atom>>
```

that takes an expression `Exp` and returns a list of atoms from each free variable identifier occurrence in `Exp`. (The returned list can have duplicate elements.) An atom X should be in the result just when `Exp` contains a record of the form `varIdExp(`X`)`, and that record is not nested within a record of form `funExp(`X E`)` (since that declares X as a formal parameter).

The following are examples using the `Test` function from the homework. Note that your answer should produce a list with the same elements, but the elements need not be in the same order as shown for the test outputs.

```
{Test {FreeVarIds varIdExp(z)} '==' [z]}
{Test {FreeVarIds varIdExp(q)} '==' [q]}
{Test {FreeVarIds applyExp(varIdExp(q) varIdExp(z))} '==' [q z]}
{Test {FreeVarIds applyExp(varIdExp(x) varIdExp(x))} '==' [x x]}
{Test {FreeVarIds funExp(x varIdExp(x))} '==' nil}
{Test {FreeVarIds funExp(z applyExp(varIdExp(q) varIdExp(z)))} '==' [q]}
{Test {FreeVarIds funExp(q applyExp(varIdExp(q) varIdExp(z)))} '==' [z]}
{Test {FreeVarIds funExp(y
                         applyExp(varIdExp(q) varIdExp(z)))}
 '==' [q z]}
{Test {FreeVarIds funExp(q
                         funExp(y
                                applyExp(varIdExp(q) varIdExp(z))))}
 '==' [z]}
{Test {FreeVarIds funExp(z
                         funExp(q
                                funExp(y
                                       applyExp(varIdExp(q) varIdExp(z)))))}
 '==' nil}
{Test {FreeVarIds applyExp(varIdExp(x)
                           funExp(x varIdExp(x)))}
 '==' [x]}
{Test {FreeVarIds applyExp(funExp(x varIdExp(x))
                           varIdExp(x))}
 '==' [x]}
{Test {FreeVarIds applyExp(funExp(x varIdExp(x))
                           varIdExp(y))}
 '==' [y]}
{Test {FreeVarIds
       applyExp(
          applyExp(
             funExp(a funExp(b funExp(y
                                      applyExp(varIdExp(a) varIdExp(b)))))
             applyExp(funExp(x varIdExp(x))
                      varIdExp(y)))
          applyExp(varIdExp(q) varIdExp(z)))}
 '==' [y q z]}
```

There is space for your answer on the next page.

Put your answer to the `FreeVarIds` problem here.