Spring, 2012 Name: _____

(Please *don't* write your id number!)

COP 4020 — Programming Languages 1

Test on Declarative Concurrency, the Message Passing Model, and Programming Models vs. Problems

Special Directions for this Test

This test has 9 questions and pages numbered 1 through 10.
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. We will take some points off for duplicated code, code with extra unnecessary cases, or code that is excessively hard to follow.
You will lose points if you do not follow the grammar when writing programs! You should always assume that the inputs given will follow the grammar for the types specified, and so your code should not have extra cases for inputs that do not follow the grammar.
When you write Oz code on this test, you may use anything in the demand-driven declarative concurrent model (as in chapter 4) and the message passing model (chapter 5). The problem may say which model is appropriate. However, you must not use imperative features (such as cells and assignment). (Furthermore, note that these models do not include the primitive `IsDet` or the library function `IsFree`; thus you are also prohibited from using either of these functions in your solutions.) 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. (This means you can use functions in the Oz base environment such as `Map`, `FoldR`, `Filter`, `Pow`, `Length`, `Wait`, `IntToFloat`, `Min`, etc.) In the message passing model you can use `NewPortObject` and `NewPortObject2` as if they were built-in.
Recall that the grammar for the type of infinite streams with elements of type `T` is as follows.

⟨IStream T⟩ ::= `T '|'` ⟨IStream T⟩

Also recall that the grammar for (possibly infinite) streams with elements of type `T` is as follows.

⟨Stream T⟩ ::= `T '|'` ⟨Stream T⟩
| `nil`

For Grading

Question:	1	2	3	4	5	6	7	8	9	Total
Points:	10	5	10	5	5	10	15	20	20	100
Score:										

1. (10 points) [UseModels] Using the data-driven concurrent model in Oz, write a function

   ```
   Graph2 : <fun {$ <fun {$ <S> <T>}: <U>> <Stream <S>> <Stream <T>>}: <Stream <#-Triple <S> <T> <U>>>
   ```

   where the type ⟨#-Triple ⟨S⟩ ⟨T⟩ ⟨U⟩⟩ is defined by the following grammar:

   ⟨#-Triple ⟨S⟩ ⟨T⟩ ⟨U⟩⟩ ::= ⟨S⟩#⟨T⟩#⟨U⟩

   The function `Graph2` takes a two-argument function, `F`, and two streams `Ss` and `Ts` (with elements of types `S` and `T`, respectively), and incrementally returns a stream of #-triples of type ⟨#-Triple ⟨S⟩ ⟨T⟩ ⟨U⟩⟩. In this result stream the Nth #-triple `X#Y#Z` is such that `X` is the Nth element of `Ss`, `Y` is the Nth element of `Ts`, and `Z` is the result of the call `{F X Y}`. The resulting stream of #-triples is only as long as the shorter of the argument streams `Ss` and `Ts`. The following are examples.

   ```
   \insert 'Graph2.oz'
   \insert 'TestingNoStop.oz'
   {StartTesting 'Graph2Test.oz $Revision: 1.1 $'}
   {Test {Graph2 fun {$ A B} A*B end nil nil} '==' nil}
   {Test {Graph2 fun {$ A B} A-B end nil [10 20 30]} '==' nil}
   {Test {Graph2 Max [5 25 125 625] [10 20 30]} '==' [5#10#10 25#20#25 125#30#125]}
   {Test {Graph2 fun {$ A B} A+B end [1 2 3 4 5 6] [10 20 30 40 50 60 70]}
    '==' [1#10#11 2#20#22 3#30#33 4#40#44 5#50#55 6#60#66]}
   {Test {Graph2 fun {$ A B} [A B] end [he she it they] [ran saw conquered]}
    '==' [he#ran#[he ran] she#saw#[she saw] it#conquered#[it conquered]]}
   {DoneTesting}
   ```

2. (5 points) [Concepts] Does the function `Graph2` need to be lazy?
   (a) Answer "yes" or "no" and (b) briefly explain.

3. (10 points) [UseModels] Consider the following Oz code (where `{Pow 2 31}` is 2^{31}).

```
declare
fun {LinearCongruential M A C}
   fun {$ X} (A*X + C) mod M end
end
Rand31 = {LinearCongruential {Pow 2 31}-1 14 1}
```

This code defines a function, `Rand31` that is a kind of linear congruential function, such as those used to generate "random" numbers. Using `Rand31` your task is to write a function

```
RandomSequence : <fun lazy {$ <Int>}: <IStream <Int>>>
```

which takes an Int, `Seed`, and lazily returns an infinite stream of Ints having the following form.

```
Seed|{Rand31 Seed}|{Rand31 {Rand31 Seed}}|{Rand31 {Rand31 {Rand31 Seed}}}|...
```

That is, the first element is the argument `Seed`, and the next element is the result of applying `Rand31` to `Seed`, and this iteration continues indefinitely. The following are test cases using the testing helpers from the homework.

```
% $Id: RandomSequenceTest.oz,v 1.2 2012/04/24 21:06:08 leavens Exp $
\insert 'RandomSequence.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'RandomSequenceTest.oz $Revision: 1.2 $'}
{Test {List.take {RandomSequence ~1} 5} '==' [~1 ~13 ~181 ~2533 ~35461]}
{Test {List.take {RandomSequence 0} 9} '==' [0 1 15 211 2955 41371 579195 8108731 113522235]}
{Test {List.take {RandomSequence 1} 9} '==' [1 15 211 2955 41371 579195 8108731 113522235 1589311291]}
{Test {List.take {RandomSequence 2} 9} '==' [2 29 407 5699 79787 1117019 15638267 218935739 917616700]}
{Test {List.take {RandomSequence 3} 9} '==' [3 43 603 8443 118203 1654843 23167803 324349243 245922109]}
{Test {List.take {RandomSequence 3} 9} '==' [3 43 603 8443 118203 1654843 23167803 324349243 245922109]}
{Test {List.take {RandomSequence 100} 12} '==' [100 1401 19615 274611 3844555 53823771 753532795
       1959524543 1663539839 1814721277 1783777762 1350568552]}
{DoneTesting}
```

Put your answer below.

```
\insert 'LinearCongruential.oz' % you should use Rand31 in your answer
```

4. (5 points) [Concepts] Consider the `RandomSequence` program called for in the previous problem and its test cases. Which of the following accurately describes the demand-driven concurrent model used in the problem? Circle the letter of all correct answers: there may be more than one.

   A. The test cases of the previous problem show that it is really possible to have observable nondeterminism in the demand-driven concurrent model.

   B. The program called for in the previous problem cannot be written, because in the demand-driven concurrent model there can be no observable nondeterminism, and yet the tests require the program to be nondeterministic.

   C. The test cases for the previous problem clearly show that random values can be generated by a declarative Oz program, contradicting the supposed determinism of the model.

   D. The program called for in the previous problem is observably deterministic, since each time it is called with the same argument it generates the same infinite stream of numbers.

   E. Random numbers are random, which means nondeterministic. Since nondeterminism is impossible in the demand-driven concurrent model, the previous problem cannot be solved.

   F. The program called for in the previous problem can be written, but not in the demand-driven concurrent model.

5. (5 points) [Concepts] Consider the following Oz program.

```
declare
fun lazy {Fractions Num}
   over(Num {Fractions 1+Num})
end

X = {Fractions 1}

fun {See F}
   case F of
      over(A B) then saw(A B)
   end
end

fun {LookAt G}
   case G of
      over(A over(B C)) then lookedAt(A B C)
   end
end

{Browse {See X}}
{Browse {LookAt X}}
```

   What is the final output, if any, that appears in the browser when this program is run? Choose the correct answer.

   A. Nothing appears, the program throws an exception.

   B. Nothing appears, the program goes into an infinite loop when it calls `Fractions`.

   C. The program shows first `saw(1 2)` then `lookedAt(1 2 3)`.

   D. The program shows first `saw(1 _)` then `lookedAt(1 2 _)`.

   E. The programs shows first `saw(1 over(2 _))` then `lookedAt(1 2 _)`.

   F. The program shows first `saw(1 over(2 over(3 _)))` then `lookedAt(1 over(2 over(3 _)) over(3 _))`.

6. [Concepts] Consider the following program in the message passing model of Oz.

```
\insert 'NewPortObject.oz'
declare
fun {NewListTracker}
   local Undet in
      {NewPortObject
       Undet
       fun {$ State Msg}
          case State#Msg of
             elems(Ls)#has(Elem ?Res) then thread Res = {Member Elem Ls} end elems(Ls)
          [] elems(Ls)#add(Elem) then elems(Elem|Ls)
          [] elems(Ls)#size(?Res) then thread Res = {Length Ls} end elems(Ls)
          end
       end
      }
   end
end
\insert 'TestingNoStop.oz'
{StartTesting 'NewListTrackerTest'}
LT = {NewListTracker}
{Send LT add(first)}
{Send LT add(second)}
{Test {Send LT size($)} '==' 2}
{Test {Send LT has(first $)} '==' true}
{Test {Send LT has(second $)} '==' true}
{DoneTesting}
```

(a) (5 points) Circle the letter of the choice below that correctly describes the behavior of this program.

A. This program is not syntactically correct or has static analysis error, so it does not compile.

B. When this program runs it throws an exception or encouters a tell error due to trying to unify two different values.

C. When this program runs it encounters a runtime type error, because functions like `NewPortObject`, `Member`, and `Length` cause problems.

D. When this program is run, the tests suspend forever, because the undetermined dataflow variable, `Undet`, is involved in the case expression's pattern matching.

E. When this program is run it completes normally and passes all the tests.

F. When this program is run the second call to `Test` fails, but all other tests pass.

(b) (5 points) Circle the letter of the correct choice below.

A. The program's use of treads could cause race conditions leading to unpredictable behavior.

B. The program's use of threads cannot change the behavior of the program. (That is, the program will give the same results as it would without using threads.)

C. The program's use of threads is illegal, because multiple threads cannot be used inside a program in the message passing model.

7. [EvaluateModels] For each of the following, name the programming model which we studied that is the *least* expressive programming model that can easily solve the problem and briefly justify your answer.

   (a) (5 points) A program that tracks salaries of graduates of a large university, where many independent users can send information or inquiries simultaneously.

   (b) (5 points) A program that takes two small tree structures, which represent different scientific theories about how certain organisms evolved, and must produce a similarity measure that says how alike they are.

   (c) (5 points) A program that produces an infinite series of increasingly more deeply nested records, for use in testing a compiler.

8. (20 points) [UseModels] Using the message passing model, write a function

   `NewTailRecursionServer : <fun {$ <fun {$ <S>}: <Bool>> <fun {$ <S>}: <S>>}: <Port>>`

   that for some type `S`, takes two function arguments, `IsDone` and `Transform` (of types `<fun {$ <S>}: <Bool>>` and `<fun {$ <S>}: <S>>`, respectively), and returns a port object that remembers these two functions and responds to the following messages:

   - `run(arg: Arg, res: ?X)`, where `Arg` is a (determined) value of type `<S>` and `X` is an undetermined dataflow variable. This message runs the function `Iterate` (see below) starting with `Arg` and using the current values of the functions `IsDone` and `Transform` as arguments, and binds the result to `X`.
   - `newTest(Done)`, where `Done` is a predicate (of type `<fun {$ <S>}: <Bool>>`). This message causes the port object to remember `Done` as the new value of the remembered `IsDone` function.
   - `newTransform(Trans)`, where `Trans` is a function (of type `<fun {$ <S>}: <S>>`). This message causes the port object to remember `Trans` as the new value of the remembered `Transform` function.

   You will be able to use the following higher-order function in your solution.

```
% Iterate: <fun {$ <S> <fun {$ <S>}: <Bool>> <fun {$ <S>}:<S>>}: <S>>
fun {Iterate Arg Done Trans}
   if {Done Arg}
   then Arg
   else {Iterate {Trans Arg} Done Trans}
   end
end
```

   The following has several examples that test the `NewTailRecursionServer` that you are to program

```
\insert 'NewTailRecursionServer.oz'
\insert 'TestingNoStop.oz'
declare
TRS = {NewTailRecursionServer fun {$ Ls#_} Ls==nil end fun {$ (H|T)#Sum} T#(H+Sum) end}
{StartTesting 'NewTailRecursionServerTest.oz $Revision: 1.1 $'}
{Test {Send TRS run(arg: nil#0 res: $)} '==' nil#0}
{Test {Send TRS run(arg: [10 100]#0 res: $)} '==' nil#110}
{Test {Send TRS run(arg: [1 10 100]#0 res: $)} '==' nil#111}
{Test {Send TRS run(arg: [1 10 100]#3000 res: $)} '==' nil#3111}

{Send TRS newTransform(fun {$ (H|T)#R} T#(H|R) end)}
{Test {Send TRS run(arg: [10 100]#nil res: $)} '==' nil#(100|10|nil)}
{Test {Send TRS run(arg: [1 10 100]#nil res: $)} '==' nil#(100|10|1|nil)}

{Send TRS newTransform(fun {$ (H|T)#R} T#(H*2|R) end)}
{Test {Send TRS run(arg: [1 10 100]#nil res: $)} '==' nil#(200|20|2|nil)}

{Send TRS newTest(fun {$ N#_} N =< 1 end)}
{Send TRS newTransform(fun {$ N#R} (N-1)#(N*R) end)}
{Test {Send TRS run(arg: 3#1 res: $)} '==' 1#6}
{Test {Send TRS run(arg: 5#1 res: $)} '==' 1#120}
{Test {Send TRS run(arg: 5#10 res: $)} '==' 1#1200}

{Send TRS newTransform(fun {$ N#R} (N-1)#(N+R) end)}
{Test {Send TRS run(arg: 5#0 res: $)} '==' 1#(5+4+3+2)}

{Send TRS newTest(fun {$ N#_} N =< 0 end)}
{Test {Send TRS run(arg: 5#0 res: $)} '==' 0#(5+4+3+2+1)}
{DoneTesting}
```

   Please write your answer on the next page.

Please write your answer to the NewTailRecursionServer problem below.

```
\insert 'Iterate.oz' % so you can use Iterate in your answer
\insert 'NewPortObject.oz'  % and of course you can use NewPortObject also
```

9. (20 points) [UseModels] [EvaluateModels] Using either the Oz declarative, declarative concurrent, or the message passing model (by to your own choice), write the following functions to implement an abstract data type <Signal>, which is intended to be a (simplified) audio signal (a description of musical sounds).

```
NewSignal : <fun {$ <List <#-Pair <Float> <Float>>>}: <Signal>>
SimilarSignals : <fun {$ <Signal> <Signal>}: <Bool>>
BandPass : <fun {$ <Signal> <fun {$ <Float>}: <Bool>>}: <Signal>>
ClipVolume : <fun {$ <Signal> <Float>}: <Signal>>
BoostBass : <fun {$ <Signal> <Float>}: <Signal>>
```

The type <#-Pair <Float> <Float>> consists of #-pairs of Floats, like 261.2#5.0. The first Float of each pair represents a frequency (in Hz) for the note to be played, and the second the volume (loudness, with larger numbers meaning louder). The functions behave as follows.

- {NewSignal LP} returns a <Signal> value, which remembers the list of pairs of Floats, LP.
- {SimilarSignals S1 S2}, returns true just when S1 and S2 have lists of #-pairs of Floats that are of the same length, and for each $1 \leq N \leq$ {Length S1}, the Nth #-pair of Floats in S1 and the Nth #-pair of Floats in S2 are similar to within 0.01. Two pairs of Floats, Freq1#Vol1 and Freq2#Vol2, are *similar to within 0.01* if {Abs Freq1-Freq2} < 0.01 and also {Abs Vol1-Vol2} < 0.01.
- {BandPass Signal FilterFun} takes a Signal, Signal, and a predicate FilterFun (of type <fun {$ <Float>}: <Bool>>), and returns a Signal that is the same as Signal, except that it does not have any #-pairs of Floats Freq#Vol in Signal such that {FilterFun Freq} is false (i.e., the result only contains those pairs whose frequency satisfies FilterFun).
- {ClipVolume Signal MaxVolume}, returns a Signal that is the same as Signal, except that in each #-pair of Floats Freq#Vol in the result, Vol is no more than MaxVolume.
- {BoostBass Signal Factor}, returns a Signal that is the same as the argument Signal, except that each #-pair of Floats Freq#Vol in Signal where Freq is strictly less than 250.0 is replaced by Freq#(Factor∗Vol).

The following are tests that show how these functions work.

```
\insert 'AudioProcessing.oz'
\insert 'TestingNoStop.oz'
declare
Octaves = {NewSignal [65.4#5.0 130.8#5.0 261.6#5.0 523.2#5.0]}
{StartTesting 'AudioProcessingTest.oz $Revision: 1.1 $'}
{Test {SimilarSignals {NewSignal nil} {NewSignal nil}} '==' true}
{Test {SimilarSignals {NewSignal [65.4#5.0]} {NewSignal [65.399#4.999]}} '==' true}
{Test {SimilarSignals Octaves {NewSignal [65.4#5.0 130.8#5.0 261.6#5.0 523.2#5.0]}} '==' true}
{Test {SimilarSignals Octaves {NewSignal [65.4#4.0 130.8#5.0 261.6#5.0 523.2#5.0]}} '==' false}
{Test {SimilarSignals Octaves {NewSignal [65.6#5.0 130.8#5.0 261.6#5.0 523.2#5.0]}} '==' false}
{Test {SimilarSignals {BandPass {NewSignal nil} fun {$ Frequency} Frequency < 270.0 end}
       {NewSignal nil}} '==' true}
{Test {SimilarSignals {BandPass Octaves fun {$ Frequency} Frequency < 270.0 end}
       {NewSignal [65.4#5.0 130.8#5.0 261.6#5.0]}} '==' true}
{Test {SimilarSignals {BandPass Octaves fun {$ Frequency} Frequency < 150.0 end}
       {NewSignal [65.4#5.0 130.8#5.0]}} '==' true}
{Test {SimilarSignals {ClipVolume {NewSignal nil} 3.0} {NewSignal nil}} '==' true}
{Test {SimilarSignals {ClipVolume {NewSignal [65.4#0.0 130.8#2.0 261.6#4.0 523.2#8.0]} 7.0}
       {NewSignal [65.4#0.0 130.8#2.0 261.6#4.0 523.2#7.0]}} '==' true}
{Test {SimilarSignals {ClipVolume {NewSignal [65.4#9.0 130.8#8.5 261.6#3.4 523.2#8.0]} 6.7}
        {NewSignal [65.4#6.7 130.8#6.7 261.6#3.4 523.2#6.7]}} '==' true}
{Test {SimilarSignals {BoostBass {NewSignal nil} 3.0} {NewSignal nil}} '==' true}
{Test {SimilarSignals
       {BoostBass {NewSignal [261.7#6.0 30.7#4.0 52.0#5.0 73.4#8.0 246.9#6.0 261.6#2.0 30.7#5.1]} 1.1}
       {NewSignal [261.7#6.0 30.7#4.4 52.0#5.5 73.4#8.8 246.9#6.6 261.6#2.0 30.7#5.61]}} '==' true}
```

```
{Test {SimilarSignals {BoostBass Octaves 1.1} {NewSignal [65.4#5.5 130.8#5.5 261.6#5.0 523.2#5.0]}} '==' true}
{DoneTesting}
```

Please put your answer for the Audio Processing problem below.