Fall, 2007 Name: ____________________

COP 4020 — Programming Languages 1

Test on Declarative Concurrency

Special Directions for this Test

This test has 7 questions and pages numbered 1 through 5.

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	10	
2	10	
3	10	
4	10	
5	20	
6	20	
7	20	

1. (10 points)

   Consider the following Oz code.

```
local X Y Z in
   thread Z = Y * 10 end
   thread Y = X + 5 end
   thread X = 3 end
   {Wait Z}
   {Show [Y Z]}
end
```

   Does this code complete normally, suspend, or fail (with an exception)? If it completes normally, say that and give its output, otherwise say what happens and briefly explain why it happens.

2. (10 points) Circle just the statement or statements below that are correct, and then briefly justify your answer.

   (a) The declarative concurrent model we studied can't help with concurrent programming in a real language, like Java.

   (b) Ideas, such as dataflow variables don't help programming in a language like Java in which those ideas are not part of the language itself.

   (c) In Java programmers can use the monitor concept (**`synchronized`** methods and the `wait` and `notifyAll` methods) to program dataflow variables as in Oz.

   (d) In Java it's impossible to use ideas from the declarative concurrent model, because in Java it's impossible to program dataflow variables.

3. (10 points) Circle just the statement or statements below that are correct, and then briefly justify your answer.

   (a) In Java, it's impossible to write a lazy function that incrementally generates a stream, because Java functions aren't lazy.

   (b) In Java one could program the `ByNeed` primitive of Oz, and use that to program lazy functions.

   (c) Java iterators are completely unlike streams, because Java iterators only produce the next item when it is asked for.

   (d) Java's iterators are completely unlike streams, because Java iterators must always generate all the items to be iterated over when the iterator object is created.

4. (10 points) Circle just the statement or statements below that are correct, and then briefly justify your answer.

   (a) Oz programs never have race conditions, even if you use Cells and threads, because all of the features in Oz are referentially transparent, even Cells.

   (b) Adding `ByNeed` to the declarative concurrent model does not allow race conditions to occur, because threads still suspend if they need the value of the variable computed by `ByNeed`.

   (c) Adding `ByNeed` to the declarative concurrent model destroys referential transparency, because a thread that needs the result before it is computed will get an unbound store variable instead of the computed value specified.

   (d) Programs written using threads are necessarily more difficult to reason about than programs that do not use threads, because every program with threads always has race conditions.

5. (20 points) In Oz, without using the library function `List.zip` (which does what this question is asking), write a function

```
Map2: <fun lazy {$ <Stream T> <Stream U> <fun {$ T U}: S>}: <Stream S>>
```

which takes two streams `Xs` and `Ys`, and a function `F` and incrementally generates the stream whose i^{th} element is the result of applying `F` to the i^{th} element of `Xs` and the i^{th} element of `Ys` (in that order). The following are examples:

```
declare
%% Some functions for testing purposes
fun lazy {LRepeat X} X|{LRepeat X} end
% Stream of numbers starting from N (see page 289 of CTM)
fun lazy {LCount N} N|{LCount N+1} end

{Test {List.take {Map2 {LRepeat a} {LCount 1} fun {$ X Y} pair(X Y) end} 5}
 '==' [pair(a 1) pair(a 2) pair(a 3) pair(a 4) pair(a 5)]}
{Test {List.take {Map2 {LCount 1} {LCount 10} fun {$ X Y} pair(X Y) end} 5}
 '==' [pair(1 10) pair(2 11) pair(3 12) pair(4 13) pair(5 14)]}
{Test {List.take {Map2 {LRepeat 0} {LCount 0} fun {$ X Y} X+Y end} 20}
 '==' {List.take {LCount 0} 20}}
{Test {List.take {Map2 {LRepeat 1} {LRepeat 2} fun {$ X Y} X+Y end} 20}
 '==' {List.take {LRepeat 3} 20}}
{Test {List.take {Map2 {LRepeat 1} {LCount 0} fun {$ X Y} X+Y end} 20}
 '==' {List.take {LCount 1} 20}}
{Test {List.take {Map2 {LRepeat 1} {LCount 1} fun {$ X Y} X*Y end} 9999}
 '==' {List.take {LCount 1} 9999}}
```

6. (20 points) In Oz, write a function

   ```
   Limit: <fun lazy {$ <Stream Float> Float}: <Stream Float>>
   ```

   that takes an infinite list of nonnegative floating point numbers, `Nums` and a positive floating point number `UpperBound`, and incrementally generates an infinite stream that is just like `Nums`, except that whenever the i^{th} element `Nums` is strictly greater than `UpperBound`, then the i^{th} element of the output stream is `UpperBound`. The following are examples.

   ```
   declare
   %% Some functions for testing purposes.
   % Return a stream that starts with Xs and then repeats Y.
   fun lazy {LThen Xs Y} {Append Xs {LRepeat Y}} end
   fun lazy {LRepeat Y} Y|{LRepeat Y} end

   {Test {List.take {Limit {LThen [1.0 1.0] 1.0} 2.0} 5}
    '==' [1.0 1.0 1.0 1.0 1.0]}
   {Test {List.take {Limit {LThen [1.0 50.1 49.0 333.0 8.0] 8.0} 50.0} 6}
    '==' [1.0 50.0 49.0 50.0 8.0 8.0]}
   {Test {List.take {Limit {LThen [2.2 2.0 3.0 2.6 7.5 2.4 2.0] 2.0} 2.5} 8}
    '==' [2.2 2.0 2.5 2.5 2.5 2.4 2.0 2.0]}
   {Test {List.take {Limit {LThen [0.4 0.99 1.001 1.00001 1.0] 1.0} 1.0} 6}
    '==' [0.4 0.99 1.0 1.0 1.0 1.0]}
   ```

7. (20 points) Write a function `LimitedAverage`,

```
LimitedAverage: <fun lazy {$ <Stream Float> <Stream Float> Float}: <Stream Float>>
```

that takes two streams of floating point numbers, `Xs` and `Ys`, and a floating point number `UpperBound`, and produces a stream of floating point numbers such that the i^{th} element of the output stream is the average of i^{th} element of `{Limit Xs UpperBound}` and the i^{th} element of `{Limit Ys UpperBound}`. The following are examples.

```
declare
%% Some functions for testing purposes.
% Return a stream that starts with Xs and then repeats Y.
fun lazy {LThen Xs Y} {Append Xs {LRepeat Y}} end
fun lazy {LRepeat Y} Y|{LRepeat Y} end

{Test {List.take
       {LimitedAverage {LThen [1.0 1.0] 1.0} {LThen [2.0 2.0] 2.0}
        2.0}
       5}
 '==' [1.5 1.5 1.5 1.5 1.5]}
{Test {List.take
       {LimitedAverage {LThen [1.0 2.0 3.0 4.0 5.0 4.0] 4.0}
                       {LThen [1.0 2.0 3.0 4.0 5.0 4.0] 4.0}
        4.0}
       6}
 '==' [1.0 2.0 3.0 4.0 4.0 4.0]}
{Test {List.take
       {LimitedAverage {LThen [1.0 2.0 3.0 4.0 5.0 4.0] 4.0}
                       {LThen [10.0 20.0 30.0 40.0 50.0 40.0] 40.0}
        40.0}
       6}
 '==' [5.5 11.0 16.5 22.0 22.5 22.0]}
{Test {List.take
       {LimitedAverage {LThen [1.0 2.0 3.0 4.0 5.0 4.0] 4.0}
                       {LThen [10.0 20.0 30.0 40.0 50.0 40.0] 40.0}
        4.0}
       6}
 '==' [2.5 3.0 3.5 4.0 4.0 4.0]}
```

Your answer should use the solution to the `Map2` and `Limit` problems above.

```
declare
\insert 'Map2.oz'  % assumed solution to the Map2 problem
\insert 'Limit.oz' % assumed solution to the Limit problem
```