Fall, 2008 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 concurrent model (as in chapters 2–4 of our textbook), so you must not use cells and assignment in your Oz solutions. (Furthermore, note that the declarative concurrent model does 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`, `Append`, etc.)

For Grading

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

1. (10 points) [Concepts]

   Consider the following Oz program.

```
declare
local W X Y Z in
   Z = X + 3
   Y = W*X + Z
   W = 5
   X = 10
   {Show values('Z = ' Z 'Y = ' Y)}
end
```

   What happens when this program is run? (Give a brief answer.)

2. (10 points) [Concepts]

   Consider the following Oz program.

```
declare
local W X Y Z in
   thread Z = X + 3 end
   thread Y = W*X + Z end
   thread W = 5 end
   thread X = 10 end
   {Wait Z}
   {Wait Y}
   {Show values('Z = ' Z 'Y = ' Y)}
end
```

   (a) Does this code terminate normally, partially terminate, or fail (with an exception)?

   (b) If it partially terminates or completes normally, give one possible output produced by the call to `Show`, (or write "no output" if there is none); otherwise briefly explain why it fails.

3. (10 points) [Concepts] [EvaluateModels]

   Which of the following statements are true? (Circle the letters of *all* the true statements; there may be more than one true statement.)

   (a) The declarative concurrent model *can be* used in a real language like Java, since a person can implement dataflow variables in Java.

   (b) The declarative concurrent model has *no* use in a real programming language, like Java, since no person can implement dataflow variables in a real language.

   (c) Concurrent Java programs with assignment, mutable fields, threads, and locking are *more* reliable than programs written in the declarative concurrent model.

   (d) Concurrent Java programs with assignment mutable fields, threads, and locking are *less* reliable than programs written in the declarative concurrent model.

4. (10 points) [UseModels]

   Write a lazy function

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

   that takes two integers, `From` and `By`, and lazily produces a stream that contains

   ```
   From | (From+By) | (From+By+By) | ...
   ```

   that is, the stream whose i^{th} item, counting from 1 is `From` + $(i-1) \times$ `By`. The following are examples, that use the `Test` method from the homework.

   ```
   {Test {List.take {FromBy 1 2} 5} '==' [1 3 5 7 9]}
   {Test {List.take {FromBy 1 3} 7} '==' [1 4 7 10 13 16 19]}
   {Test {List.take {FromBy 3 10} 6} '==' [3 13 23 33 43 53]}
   {Test {List.take {FromBy 3 ~1} 6} '==' [3 2 1 0 ~1 ~2]}
   {Test {List.take {FromBy 2 ~1} 6} '==' [2 1 0 ~1 ~2 ~3]}
   ```

5. (10 points) [Concepts]

   Does the function `FromBy` in the previous problem have to be lazy? Answer "yes" or "no" and give a brief explanation.

6. (25 points) [UseModels]

   Using Oz's demand-driven concurrent model, write a lazy function

```
TestOverStreams: <fun lazy {$ <IStream Int> <IStreamInt> <fun {$ <Int>}: <Int>>}
                          : <IStream <Int>#<Int>#<Bool> > >
```

   that takes two infinite streams, `Args` and `Expected`, and a function `F`, and produces an infinite stream of triples such that the i^{th} element of the result, $A\#R\#B$, is such that A is the i^{th} element of `Args`, R is the result of applying `F` to A, and B is true just when R is equal to (using ==) the i^{th} element of `Expected`. The following are examples that use the `Test` procedure from the homework and the `FromBy` function in problem 4.

```
{Test {List.take
       {TestOverStreams {FromBy 1 1} {FromBy 2 2} fun {$ Ith} 2*Ith end}
       6}
 '==' [1#2#true 2#4#true 3#6#true 4#8#true 5#10#true 6#12#true]}
{Test {List.take
       {TestOverStreams {FromBy 1 2} {FromBy 3 3} fun {$ Ith} 3*Ith end}
       7}
 '==' [1#3#true 3#9#false 5#15#false 7#21#false 9#27#false 11#33#false 13#39#false]}
{Test {List.take
       {TestOverStreams {FromBy 1 1} {FromBy 3 3} fun {$ Ith} 3*Ith end}
       7}
 '==' [1#3#true 2#6#true 3#9#true 4#12#true 5#15#true 6#18#true 7#21#true]}
{Test {List.take
       {TestOverStreams {FromBy 10 10} {FromBy 30 30} fun {$ Ith} (2*Ith)+Ith end}
       5}
 '==' [10#30#true 20#60#true 30#90#true 40#120#true 50#150#true]}
```

7. (25 points) [UseModels]

   Using Oz's declarative concurrent model, write a function

```
AddFailCounts: <fun lazy {$ <List <Int>#<Int>#<Bool>>}
                         : <List <Int>#<Int>#<Bool>#<Int> >>
```

   that takes a list of triples `TestOutput` and produces a list of 4-tuples such that if $A\#R\#B$ is the i^{th} element of `TestOutput`, then the i^{th} element of the result is $A\#R\#B\#C$. where C is a count of how many times the third part of the `TestOutput` triple (the B part) has been **`false`** for elements of the stream with index less than or equal to i. The following are examples that use the `Test` procedure from the homework.

```
{Test {AddFailCounts nil} '==' nil}
{Test {AddFailCounts [1#2#false 2#4#true]} '==' [1#2#false#1 2#4#true#1]}
{Test {AddFailCounts [1#2#true 2#4#true 3#6#true
                      4#8#true 5#10#true 6#12#true]}
 '==' [1#2#true#0 2#4#true#0 3#6#true#0
       4#8#true#0 5#10#true#0 6#12#true#0]}
{Test {AddFailCounts [1#3#true 3#9#false 5#15#false
                      7#21#false 9#27#false 11#33#false 13#39#false]}
 '==' [1#3#true#0 3#9#false#1 5#15#false#2
       7#21#false#3 9#27#false#4 11#33#false#5 13#39#false#6]}
{Test {AddFailCounts [1#3#false 3#9#false 5#15#true
                      7#21#false 9#27#true 11#33#false 13#39#false]}
 '==' [1#3#false#1 3#9#false#2 5#15#true#2
       7#21#false#3 9#27#true#3 11#33#false#4 13#39#false#5]}
```