Fall, 2010 Name: _____

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.
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 `Pow`, `Length`, `Wait`, `Map`, `FoldR`, `Filter`, 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⟩

For Grading

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

1. (5 points) [Concepts] This is a question about flow control; consider a program with two threads, one of which is producing a stream and the other consuming it. Which of the following is a true statement? (Circle the letter of the correct answer.)

    A. To get flow control with lazy execution, the producer must be made lazy.

    B. To get flow control with lazy execution, the consumer must be made lazy.

    C. To get flow control with lazy execution, both the producer and the consumer must be made lazy.

    D. To get flow control with lazy execution, neither the producer nor the consumer need to be made lazy.

2. (5 points) [Concepts] The following questions concern the message passing programming model. Which of the following is a true statement? (Circle the letter of the correct answer.)

    A. In the message passing model, message sends are synchronous.

    B. In the message passing model, message sends will block or wait until the port services the message and responds to it.

    C. In the message passing model, message sends can sometimes block or wait until the port services the message and responds to it, depending on the program's state and how busy the port is.

    D. In the message passing model, message sends never block or wait.

3. (10 points) [EvaluateModels] In Oz, the `IsDet` function returns true when passed a dataflow variable with a determined value and false otherwise. Briefly answer this: Why is `IsDet` *not* part of the declarative concurrent computational model?

4. [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 is given two strings of letters, each representing part of the DNA of an organism, and finds a match between them in such a way that the letters A and T and the letters G and C match each other (an A in one string matching a T in the other).

   (b) (5 points) A program that computes the first million digits of the decimal representation of π, using an approximation formula (such as $\pi = 4 \cdot \Sigma_{k=0}^{\infty} \frac{(-1)^k}{2 \cdot k+1}$) that generates an infinite series.

   (c) (5 points) A program that tracks ticket reservations for a football stadium, where many independent web browsers can send requests and make reservations simultaneously.

5. (10 points) [UseModels]

   Using the demand-driven concurrent model, write a lazy function to generate an IStream of numbers representing the amount of a deposit (or loan) at a given interest rate

   ```
   Interest : <fun lazy {$ <Float> <Float>}: <IStream Float> >
   ```

   that takes a floating point number, `Principle`, and another Float, `Rate`, and returns an IStream of numbers where the ith element is the value of the deposit (or loan) at the beginning of the ith year, where each year an amount equal to the previous year's value plus that value times `Rate`. The Value at the beginning is `Principle`. The following are examples, using the `FloatTesting` procedures and functions from the homework.

```
\insert 'Interest.oz'
\insert 'FloatTesting.oz'
declare
% 2 helpers for testing purposes only (you don't have to program them)...
% WithinDime tests if list elements are within 0.10 of each other
WithinDime = {TestMaker Within WithinLists 0.10}
% WithinHundred tests if list elements are within 100.00 of each other
WithinHundred = {TestMaker Within WithinLists 100.00}
% ... The real tests are below
{StartTesting 'Interest'}
{WithinDime {List.take {Interest 100.0 0.02} 5} % 2 percent interest
 '==' [100.0 102.0 104.04 106.12 108.24]}
{WithinDime {List.take {Interest 5000.0 0.05} 5}  % 5 percent interest
 '==' [5000.0 5250.0 5512.5 5788.1 6077.5]}
{WithinDime {List.take {Interest 5000.0 0.10} 6}  % 10 percent interest
 '==' [5000.0 5500.0 6050.0 6655.0 7320.5 8052.6]}
{WithinHundred {List.take {Interest 1.5e5 0.15} 20}  % 15 percent interest
 '==' [1.5e5 1.725e5 1.9838e5 2.2813e5 2.6235e5 3.017e5 3.4696e5 3.99e5
       4.5885e5 5.2768e5 6.0683e5 6.9786e5 8.0254e5 9.2292e5 1.0614e6
       1.2206e6 1.4036e6 1.6142e6 1.8563e6 2.1348e6]}
{StartTesting done}
```

6. (10 points) [UseModels]

   Using the demand-driven concurrent model, write an incremental function

   ```
   ISGreater : <fun lazy {$ <IStream T> <IStream T>}: <IStream Bool> >
   ```

   that for some type `T`, takes two infinite streams of type `T`, `IS1` and `IS2`, and lazily returns an infinite stream of Booleans, with the ith Boolean being true just when the ith element of `IS1` is strictly greater than the ith element of `IS2`.

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

```
\insert 'ISGreater.oz'
declare
% Some helper functions for testing only (which you don't have to implement)...
fun lazy {Count N} N|{Count N+1} end
fun lazy {Squares N} N*N|{Squares N+1} end
fun lazy {Cubes N} N*N*N|{Cubes N+1} end
% ... end of testing helpers
{StartTesting 'ISGreater'}
{Test {List.take {ISGreater {Cubes 1} {Squares 1}} 7}
 '==' [false true true true true true]}
{Test {List.take {ISGreater {Squares 1} {Count 21}} 8}
 '==' [false false false false false true true true]}
{Test {List.take {ISGreater {Squares 1} {Count 20}} 8}
 '==' [false false false false true true true true]}
{Test {List.take {ISGreater {Count 20} {Squares 1}} 8}
 '==' [true true true true false false false false]}
{Test {List.take {ISGreater {Count 2} {Count 1}} 10}
 '==' [true true true true true true true true true true]}
{StartTesting done}
```

7. (15 points) [UseModels] Using the message passing model, write a fuction:

   `NewStack : <fun {$ }: <Port>>`

   that takes no arguments and returns a port object. The returned port object handles the following massages, for some type `T`:

   - `push(Val)`, which contains a value, `Val`, of type `T`.
   - `pop`
   - `size(Variable)`, which contains an undetermined dataflow variable,
   - `top(Variable)`, which contains an undetermined dataflow variable.

   Sending the port object the message `push(Val)`, where `Val` is some value of type `T` pushes `Val` onto the top of the stack.

   Sending the port object the message `pop` pops the top value off of the top of the stack.

   Sending the port object the message `top(Variable)`, where `Variable` is an undetermined dataflow variable, unifies `Variable` with the value on the top of the stack and leaves the stack's state unchanged.

   Sending the port object the message `size(Variable)`, where `Variable` is an undetermined dataflow variable, unifies `Variable` with the number of values on the stack.

   You can assume that the port object never receives the `pop` or the `top` message when the stack is empty (has size 0).

   The following are examples, written using the `Test` procedure from the homework.

```
\insert 'NewStack.oz'
declare
MyStack = {NewStack}
S2 = {NewStack}
{Test {Send MyStack size($)} '==' 0}
{Send MyStack push(first)}
{Test {Send MyStack size($)} '==' 1}
{Test {Send MyStack top($)} '==' first}
{Send MyStack push(second)}
{Test {Send MyStack size($)} '==' 2}
{Test {Send MyStack top($)} '==' second}
{Send MyStack pop}
{Test {Send MyStack size($)} '==' 1}
{Test {Send MyStack top($)} '==' first}
{Send MyStack push(third)}
{Send MyStack push(fourth)}
{Send MyStack push(fifth)}
{Test {Send MyStack size($)} '==' 4}
{Test {Send MyStack top($)} '==' fifth}
{Send MyStack pop}
{Test {Send MyStack size($)} '==' 3}
{Test {Send MyStack top($)} '==' fourth}
{Send MyStack pop}
{Send MyStack pop}
{Test {Send MyStack size($)} '==' 1}
{Test {Send MyStack top($)} '==' first}
{Send MyStack pop}
{Send S2 push(4020)}
{Test {Send S2 size($)} '==' 1}
{Test {Send S2 top($)} '==' 4020}
{Test {Send MyStack size($)} '==' 0}
```

   There is space for your answer on the next page.

Please put your answer for the `NewStack` problem below. It would also make sense to save the size of the state in the state.

```
\insert 'NewPortObject.oz' % you can use this in your solution
```

8. (20 points) [UseModels] [EvaluateModels]

Using either the (demand-driven) declarative concurrent model or the message passing model, implement the abstract datatype `<Dropbox T>`, with the following functions and procedures for any type `T`:

```
NewDropbox: <fun {$ }: <Dropbox T> >
DBReady: <fun {$ <Dropbox T>}: <Bool> >
DBFetch: <fun {$ <Dropbox T>}: <T> >
DBDeposit: <proc {$ <Dropbox T> <T>}>
```

A dropbox has roughly two states. When created by `NewDropbox`, it starts out empty, and when something of type `T` is deposited in it, it becomes full. The functions and procedures whose types are above act as follows.

- The function `NewDropbox`, when called with no arguments, returns a new `<Dropbox T>`.
- The function `DBReady` takes a `<Dropbox T>` argument, `DB`, and returns true just when `DB` is full (and false otherwise).
- The function `DBFetch` takes a `<Dropbox T>` argument, `DB`, and returns the value deposited in the dropbox. `DBFetch` can be called when the drobbox argument `DB` is either empty or full; when `DB` is empty, it waits until a value of type `T` is deposited before returning the deposited value.
- The procedure `DBDeposit` takes two arguments, a `<Dropbox T>` argument, `DB`, and a value of type `T`, `Value`; it fills the dropbox `DB` with `Value`, and this causes any waiting calls to `DBFetch` to complete and return `Value`.

Note that calls to `DBFetch` are allowed to happen when the dropbox is empty, and that `DBDeposit` is a procedure (not a function). However, you can assume that `DBDeposit` is never called when the dropbox argument is already full. The following are examples:

```
\insert 'NewDropbox.oz'
{StartTesting 'Dropbox'}
local DB = {NewDropbox} in
   {Test {DBReady DB} '==' false}
   {DBDeposit DB meeting(7 9 junkyard)}
   {Test {DBReady DB} '==' true}
   {Test {DBFetch DB} '==' meeting(7 9 junkyard)}
   {Test {DBReady DB} '==' true}
end
local MyDB = {NewDropbox} in
   {Test {DBReady MyDB} '==' false}
   local Res R2 in
      thread Res = {DBFetch MyDB} end  % fetch before deposit...
      {Test {DBReady MyDB} '==' false}
      thread R2 = {DBFetch MyDB} end   % threaded because it waits...
      {DBDeposit MyDB attack_at_dawn}
      {Test {DBReady MyDB} '==' true}  % now Res and R2 are determined...
      {Test Res '==' attack_at_dawn}
      {Test R2 '==' attack_at_dawn}
      {Test {DBReady MyDB} '==' true}
      {Test {DBFetch MyDB} '==' attack_at_dawn}
   end
end
{StartTesting done}
```

There is space for your answer on the next page

Please put your answer to the `Dropbox` problem below.

9. (10 points) [UseModels] [EvaluateModels]

   Using either the (demand-driven) declarative concurrent model or the message passing model, implement the following procedure:

   `DoWhile: <proc {$ <fun {$ }: <Bool>> <proc {$}>}>`

   that two arguments, a function, `Continue`, which returns a Boolean when called, and a proc `Body`. A call `{DoWhile Continue Body}` first calls the function `Continue` with no arguments; if that call returns `true`, then it calls the procedure `Body`. Then it repeats this process as long as the call to `Continue` returns true. The following are some examples:

```
\insert 'DoWhile.oz'
% start of code just to aid testing (you don't have to write this part)...
\insert 'NewPortObject.oz'
declare
fun {NewCounter Num}
   {NewPortObject Num
    fun {$ N Msg}
       case Msg of
          add(M) then N+M
       [] value(?V) then V=N N
       end
    end}
end
proc {CounterAdd Counter M} {Send Counter add(M)} end
fun {CounterValue Counter} {Send Counter value($)} end
% ... end of code just to aid testing
{StartTesting 'DoWhile'}
local Counter10 = {NewCounter 10}
      Counter0 = {NewCounter 0}
in
   {DoWhile fun {$} {CounterValue Counter10} > 0 end
    proc {$} {CounterAdd Counter0 10}
       {CounterAdd Counter10 ~1}
    end}
   {Test {CounterValue Counter10} '==' 0}
   {Test {CounterValue Counter0} '==' 100}
   {DoWhile fun {$} {CounterValue Counter10} < 7 end
    proc {$} {CounterAdd Counter10 1}
    end}
   {Test {CounterValue Counter10} '==' 7}
end
{StartTesting done}
```