Spring, 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 8.
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 `Min`, `Max`, `Map`, `FoldR`, `Filter`, `Append`, `IntToFloat`, `List.take`, `List.drop`, 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	10	10	10	15	15	20	100
Score:										

1. (5 points) [Concepts] Consider a function `F`, written using Oz's demand-driven concurrent programming model. Which of the following is a true statement? (Circle the letter of the correct answer.)

   A. A person running `F` twice with the same arguments might see different answers at different times, due to race conditions, which are inherent in concurrent programming.

   B. A person running `F` twice with the same arguments cannot observe different answers.

   C. Two people running `F` with the same arguments may see different results, but the one person will never see different results.

   D. When function `F` is run on some arguments, it never finishes completely, so it's impossible to observe two calls of `F`.

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. Functions written in the message passing model are always observably deterministic.

   B. The message passing model *cannot* be used to merge requests from independent clients.

   C. The message passing model *can* be used to merge requests from independent clients.

   D. The `Send` primitive waits for the server to pick up the message sent to it before the execution of `Send` finishes.

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

   (a) (5 points) A function that takes a recursive tree-like data structure representing an XML configuration file and deleting subtrees that have obsolete types, to produce a new data structure that is similar to the argument but without any subtrees that are of obsolete types.

   (b) (5 points) A function that computes an infinite stream of better and better approximations to the square root of a floating point number, allowing the caller to find the first approximation that is good enough for a specific tolerance.

4. (10 points) [UseModels]

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

   ```
   LMap2 : <fun lazy {$ <IStream S> <IStream T> <fun {$ S T}: U>}: <IStream U>>
   ```

   that for some types `S`, `T`, and `U`, takes an `<IStream S>`, `FirstIStrm`, an `<IStream T>`, `SecondIStrm`, and a function, `G`, of type `<fun {$ S T}: U>`, and produces an infinite stream of elements of `G`'s return type (`U`). The ith element of the produced IStream is the result of applying `G` to the ith element of `FirstIStrm` and the ith element of `SecondIStrm` (in that order).

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

   ```
   \insert 'LMap2.oz'
   declare
   % Some functions just for testing...
   fun lazy {Count N} N|{Count N+1} end
   fun lazy {CountDown N} N|{CountDown N-1} end
   {StartTesting 'LMap2'}
   {Test {List.take {LMap2 {Count 1} {Count 10} fun {$ X Y} X#Y end} 10}
    '==' [1#10 2#11 3#12 4#13 5#14 6#15 7#16 8#17 9#18 10#19]}
   {Test {List.take {LMap2 {Count 1} {Count 10} fun {$ X Y} X+Y end} 10}
    '==' [11 13 15 17 19 21 23 25 27 29]}
   {Test {List.take {LMap2 {Count 1} {CountDown 1} fun {$ X Y} X+Y end} 10}
    '==' [2 2 2 2 2 2 2 2 2 2]}
   {StartTesting done}
   ```

5. (10 points) [UseModels]

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

   ```
   LRange : <fun lazy {$ <IStream Int>}: <IStream <Pair Int Int>>>
   ```

   that takes an infinite stream of Integers (i.e., an `<IStream Int>`), `IStrm`, and produces an infinite stream of `#`-pairs of the form `Small#Big`, where `Small` is the minimum of all the numbers seen in `IStrm` so far and `Big` is the maximum of all the numbers seen in `IStrm` so far. There is one output pair for each element of the `IStrm`, with the ith element of the output stream being a pair `SmallI#BigI`, where `SmallI` is the minimum of the first i elements of the input stream and `BigI` the maximum of the first i elements of the input stream. Be sure that your `LRange` function does not loop forever when its input is infinite.

   Hint: note that the first pair output will always have the form `N#N` where `N` is the first element of the input stream. You can use Oz's built in functions `Min` and `Max`, both of which have type `<fun {$ Int Int}: Int>`.

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

```
\insert 'LRange.oz'
declare
% Some functions just for testing...
fun lazy {Count N} N|{Count N+1} end
fun lazy {CountDown N} N|{CountDown N-1} end
% {Oscillate 0 1 1} == 0|1|~2|3|~4|5|~6|7|~8|9|~10|...
% {Oscillate 0 1 2} == 0|2|~4|6|~8|10|~12|14|~16|18|~20|...
fun lazy {Oscillate N Count Sign} N|{Oscillate Sign*Count Count+1 Sign*~1} end
{StartTesting 'LRange'}
{Test {List.take {LRange {Count 1}} 5} '==' [1#1 1#2 1#3 1#4 1#5]}
{Test {List.take {LRange {CountDown 0}} 5} '==' [0#0 ~1#0 ~2#0 ~3#0 ~4#0]}
{Test {List.take {LRange {Oscillate 0 1 1}} 10} '==' [0#0 0#1 ~2#1 ~2#3 ~4#3 ~4#5 ~6#5 ~6#7 ~8#7 ~8#9]}
{Test {List.take {LRange {Oscillate 0 1 2}} 10} '==' [0#0 0#2 ~4#2 ~4#6 ~8#6 ~8#10 ~12#10 ~12#14 ~16#14 ~16#18]}
{StartTesting done}
```

6. (10 points) [UseModels] Using the message passing model, write a fuction:

   `NewCountSatisfying : <fun {$ <fun {$ T}: Bool>}: <Port>>`

   that, for some type `T` takes a predicate, `Pred`. The argument `Pred` is a function that takes an argument of type `T` and returns a Boolean. The returned port object handles the following massages:

   - `sample(V)`, which contains a value, `V`, of type `T`, and processes it by checking to see if `{Pred V}` is true, and
   - `getPassingCount(<Variable>)`, which contains an undetermined dataflow variable.

   Sending the port object the message `getPassingCount(X)`, where `X` is an undetermined dataflow variable, results in the binding of `X` to the number of `sample(V)` messages processed (so far) by the port object for which `{Pred V}` returns **`true`**.

   You can and should use `NewPortObject` to write your solution.

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

```
\insert 'NewCountSatisfying.oz'
declare
{StartTesting 'NewCountSatisfying'}
GT7 = {NewCountSatisfying fun {$ X} X > 7 end}
{Test {Send GT7 getPassingCount($)} '==' 0}
{Send GT7 sample(1)}  {Send GT7 sample(5)}  {Send GT7 sample(7)}  {Send GT7 sample(8)}
{Test {Send GT7 getPassingCount($)} '==' 1}
{Send GT7 sample(20)}
{Test {Send GT7 getPassingCount($)} '==' 2}
{Send GT7 sample(903908138904714897)}
{Test {Send GT7 getPassingCount($)} '==' 3}
{StartTesting done}
```

7. (15 points) [UseModels]

   Using the message passing model, write a function

   `NewTurtle: <fun {$ <Int> <Int>}: <Port>>`

   that returns a port object that acts as a "Logo"-style computational "turtle." A turtle is something that crawls around on the $X - Y$ plane. It remembers its coordinates and 2 values `XD` and `YD`, which are the amounts to move whenver asked in the X and Y directions. The turtle that results from a call such as `{NewTurtle XDel YDel}` starting at the origin $(0, 0)$, and starts with `XD` as `XDel` and `YD` as `YDel`. Each time it is asked to move it moves `XD` in the X direction and `YD` in the Y direction. The returned port object responds to the following messages:

   - `move(Var)`, where `Var` is an undetermined dataflow variable. This message tells the turtle to make its next move, and unifies `Var` with a `#`-pair consisting of the new X-coordinate position and the new Y-coordinate position.
   - `setXDelta(<Int>)` which changes the amount (i.e., `XD`) that the turtle moves in the X direction when it receives a `move` message to the given integer.
   - `setYDelta(<Int>)` which changes the amount (`YD`) that the turtle moves in the Y direction when it receives a `move` message to the given integer.

   The following are examples:

```
\insert 'NewTurtle.oz'
\insert 'TestingNoStop.oz'
declare
{StartTesting 'NewTurtle'}
T = {NewTurtle 1 1}
{Test {Send T move($)} '==' 1#1}  {Test {Send T move($)} '==' 2#2}  {Test {Send T move($)} '==' 3#3}
{Send T setXDelta(2)}
{Test {Send T move($)} '==' 5#4}  {Test {Send T move($)} '==' 7#5}
{Send T setYDelta(~3)}
{Test {Send T move($)} '==' 9#2}  {Test {Send T move($)} '==' 11#~1}
{Send T setXDelta(~1)}
{Test {Send T move($)} '==' 10#~4}  {Test {Send T move($)} '==' 9#~7}
{StartTesting done}
```

8. (15 points) [UseModels]

   Using the Oz message passing model, write a function that for some types `T` and `S` has type:

   ```
   NewIterator: <fun {$ <fun {$ T S}: S> <S>}: <Port>>
   ```

   that takes a function, `Combiner` and a value of type `S`, `InitialAcc`, and returns a port object. The returned port object tracks an accumulator (of type `S`), which starts out at `InitialAcc`. The port object responds to the following kinds of messages:

   - `element(TVal)`, where `TVal` has type `T`. This message runs `Combiner` passing it `TVal` and the current accumulator as arguments, and it makes the port object's new accumulator be the result returned by `Combiner` on those arguments.
   - `getAcc(X)`, where `X` is an undetermined dataflow variable. This message unifies `X` with the current accumulator value, and leaves the accumulator unchanged.

   The following are some examples:

   ```
   \insert 'NewIterator.oz'
   \insert 'TestingNoStop.oz'
   declare
   {StartTesting 'NewIterator'}
   Sum = {NewIterator fun {$ X Acc} X+Acc end 0}
   {Test {Send Sum getAcc($)} '==' 0}
   {Send Sum element(5)}   {Send Sum element(10)}
   {Test {Send Sum getAcc($)} '==' 15}
   Lister = {NewIterator fun {$ X Acc} X|Acc end nil}
   {Test {Send Lister getAcc($)} '==' nil}
   {Send Lister element(a)}   {Send Lister element(b)}
   {Test {Send Lister getAcc($)} '==' b|a|nil}
   {StartTesting done}
   ```

9. (20 points) [UseModels] [EvaluateModels] Using either one of the declarative concurrent models or the message passing model, write the following

```
NewDispatcher: <fun {$ } : <Dispatcher>>
Await: <proc {$ <Dispatcher> <Variable>}
DispatchAll: <proc {$ <Dispatcher>}
```

where you will represent the type `<Dispatcher>` in a way of your choosing. The function `NewDispatcher` takes no arguments and returns a dispatcher object that remembers a list of "waiting" dataflow variables sent to it in `Await` calls that have not yet been dispatched (see below).

The procedure `Await` takes a dispatcher argument and an undetermined dataflow variable; a call like `{Await D X}`, where `X` is undetermined, adds `X` to `D`'s list of waiting variables. The call `{Await D X}` does not return until the dispatcher `D` is used in a call `{DispatchAll D}`.

The procedure `DispatchAll` takes a dispatcher argument and causes all of the dataflow variables that are in that dispatcher's list of waiting dataflow variables to be unified with **`unit`**, which makes all waiting calls to `Await` on that dispatcher finish (and return).

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

```
\insert 'NewDispatcher.oz'
\insert 'TestingNoStop.oz'
declare
W X Y Z % some fresh variables declared here
{StartTesting 'NewDispatcher'}
D = {NewDispatcher}
thread {Await D W} end  thread {Await D X} end  thread {Await D Y} end
{Delay 2000}
% At this point, W, X, and Y are all *not* determined...
{Assert {Not {IsDet W}} andthen {Not {IsDet X}} andthen {Not {IsDet Y}}}
{DispatchAll D}
{Assert W == unit andthen X == unit andthen Y == unit} % shouldn't suspend
thread {Await D Z} end
{Delay 2000}
% At this point, Z is *not* determined...
{Assert {Not {IsDet Z}}}
{DispatchAll D}
{Assert Z == unit}  % shouldn't suspend
{StartTesting done}
```