Fall, 2011 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. You will lose points if you do not follow the grammar when writing programs!
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`, 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	20	15	10	100
Score:										

1. (5 points) [Concepts] Consider the following Oz program and test.

```
\insert 'TestingNoStop.oz'
declare
fun {ListFromToOffset Initial Limit Offset}
   fun {From N}
      N|{From N+1}
   end

   fun {AddEach H|T}
      Offset+H|{AddEach T}
   end

   IntsFromN = thread {From Initial} end
   Added = thread {AddEach IntsFromN} end
in
   {List.take Added (((Limit-Offset)-Initial)+1)}
end

{Test {ListFromToOffset 0 14 10} '==' [10 11 12 13 14]}
```

The above program doesn't pass the test given at its end. Briefly (a) state what the problem with the program is, and briefly say (b) what simple change would fix it (without changing the test).

2. (5 points) [Concepts] Consider the following Oz program.

```
local A B C D E
      fun lazy {Compute X} 10*X end
in
   thread A = [B+0 C D E] end
   thread B = C + E*100 end
   thread C = {Compute D} end
   thread D = 1 end
   thread E = 50 + D end
   {Wait A} {Wait B} {Wait C} {Wait D} {Wait E}
   {Browse A}
end
```

Which of the following correctly describes what, if anything, is shown in the Browser after this program runs?

A. It is impossible to tell, because the threads cause so many race conditions, the output of the program can't be predicted with any certainty.

B. The program suspends and never shows anything in the Browser.

C. The program shows the name `A` in the Browser..

D. The program shows the list `[5110 10 1 51]` in the Browser..

E. The program shows the list `[1 5 1 1001]` in the Browser..

F. The program shows the list `nil` in the Browser..

G. The program encounters an error or throws an exception.

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

```
\insert 'NewPortObject.oz'
declare
fun {NewProcServer}
   {NewPortObject
    state_is_ignored
    fun {$ State Msg}
       case Msg of
          run(P) then {P} State
       end
    end}
end

PS = {NewProcServer}
local Ten
      Twelve
      proc {DelaySecs N ?Var} {Delay N*1000} Var=done end
in
   {Send PS run(proc {$ } {DelaySecs 10 Ten} end)}
   {Send PS run(proc {$ } {DelaySecs 12 Twelve} end)}
   % part (a) asks how long it takes to get to here
   {Wait Ten}
   {Wait Twelve}
   {Browse all_done} % part (b) asks how long it takes to get here
end
```

You may recall that the procedure `Delay` waits at least the given number of milliseconds.

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

A. The two messages sent to the port `PS` should be both be delivered to that port within 1 second after the program starts.

B. The two messages sent to the port `PS` should both be delivered to the port no sooner than 10 seconds after the program starts, but no later than 11 seconds after the program starts.

C. The two messages sent to the port `PS` should both be delivered to the port no sooner than 12 seconds after the program starts, but no later than 13 seconds after the program starts.

D. The two messages sent to the port `PS` should both be delivered to the port no sooner than 22 seconds after the program starts, but no later than 23 seconds after the program starts.

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

A. The program should finish executing, and it shows `all_done` in the Browser in no less than 2 seconds, but no more than 9 seconds.

B. The program should finish executing, and it shows `all_done` in the Browser in no less than 10 seconds, but no more than 12 seconds.

C. The program should finish executing, and it shows `all_done` in the Browser in no less than 12 seconds, but no more than 20 seconds.

D. The program should finish executing, and it shows `all_done` in the Browser in no less than 22 seconds, but no more than 30 seconds.

E. The program should finish executing, and it shows `all_done` in the Browser in no less than 50 seconds, but no more than 60 seconds.

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 a structured document, such as an HTML page, and must produce a document with the same structure but without images (which appear in the original structure as a certain kind of node).

   (b) (5 points) A program that tests a given function, by generating an infinite series of increasingly complex test cases, and feeding them into the function until either the function breaks or some previously specified limit on the complexity of the tests is reached.

   (c) (5 points) A program that tracks shipping of packages, where many independent web browsers can send inquiries and many of the company's employees can update the status of packages simultaneously.

5. (10 points) [UseModels]

Using the demand-driven concurrent model, write a function

```
PowersOfN : <fun {$ <Int>}: <IStream Int> >
```

that takes one argument, an Int `N`, and lazily returns an IStream of Ints where the ith element (counting from 1) is the value N^{i-1}. The following has several examples.

```
\insert 'TestingNoStop.oz'
\insert 'PowersOfN.oz'
{StartTesting 'PowersOfNTest $Revision: 1.1 $'}
{Test {List.take {PowersOfN 2} 10} '==' [1 2 4 8 16 32 64 128 256 512]}
{Test {List.take {PowersOfN 3} 8} '==' [1 3 9 27 81 243 729 2187]}
{Test {List.take {PowersOfN 4} 5} '==' [1 4 16 64 256]}
{Test {List.take {PowersOfN 5} 18} '==' [1 5 25 125 625 3125 15625 78125 390625
                                         1953125 9765625 48828125 244140625
                                         1220703125 6103515625 30517578125
                                         152587890625 762939453125]}
{Test {List.take {PowersOfN 10} 7} '==' [1 10 100 1000 10000 100000 1000000]}
{DoneTesting}
```

6. (10 points) [UseModels]

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

```
Averages : <fun lazy {$ <IStream <List Float>>}: <IStream Float> >
```

that takes an infinite streams of non-empty, finite Lists of Floats, `ISF`, and lazily returns an infinite stream of Floats, with the ith Float in the result being the average of all the Floats in the ith list in `ISF`. Remember that in Oz you cannot mix Floats and Ints in arithmetic expressions!

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

```
\insert 'FloatTesting.oz'
\insert 'Averages.oz'
declare
fun lazy {RepeatingListOf Elements} % A helper for testing only
   {Append Elements {RepeatingListOf Elements}}
end
fun {FromToBy From To By}  % A helper for testing only
   %% REQUIRES: By \= 0.0 and none of From, To, or By is NaN.
   %% ENSURES: result is the finite list [From From+By From+2*By ... To]
   if (By > 0.0 andthen From > To) orelse (By < 0.0 andthen From < To)
   then nil
   else From|{FromToBy From+By To By}
   end
end
{StartTesting 'AveragesTest $Revision: 1.3 $'}
{WithinTest {List.take {Averages
                        {RepeatingListOf [[1.0] [1.0 2.0] [1.0 2.0 3.0] [1.0 2.0 3.0 4.0]
                                          {FromToBy 1.0 100.0 1.0} {FromToBy 1.0 1000.0 1.0}
                                         ]}}
             7} '~=~' [1.0 1.5 2.0 2.5 50.5 500.5 1.0]}
{WithinTest {List.take {Averages
                        {RepeatingListOf [[~1.0] [~1.0 ~2.0] [~1.0 ~2.0 3.0] [~1.0 ~2.0 3.0 4.0]
                                          {FromToBy ~1.0 ~100.0 ~1.0} {FromToBy ~1.0 ~1000.0 ~2.0}
                                         ]}}
             7} '~=~' [~1.0 ~1.5 0.0 1.0 ~50.5 ~500.0 ~1.0]}
{WithinTest {List.take {Averages
                        {RepeatingListOf [{FromToBy 0.0 6000.0 3.0} {FromToBy 0.0 3.0e3 100.0}
                                         ]}}
             4} '~=~' [3000.0 1500.0 3000.0 1500.0]}
{WithinTest {List.take {Averages
                        {RepeatingListOf {Map {FromToBy 1.0 10.0 1.0} fun {$ N} [0.0 N 2.0*N] end}}}
             10} '~=~' {FromToBy 1.0 10.0 1.0}}
{DoneTesting}
```

7. (20 points) [UseModels] Using the message passing model in Oz, write a function

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

that takes no arguments and returns a port that can be used to send messages to a new port object. The port object acts remembers the value of an integer, which represents the port object's accumulator. (You might think of the port object as a kind of calculator.) This port object responds to the following messages:

- `add(M)`, where `M` is an Int, which adds `M`'s value to the accumulator's value.
- `subtract(M)`, where `M` is an Int, which makes the accumulator become its old value minus the value of `M`.
- `multiply(M)`, where `M` is an Int, which makes the accumulator become its old value times the value of `M`.
- `square`, which makes the accumulator become the square of its old value.
- `enter(M)`, where `M` is an Int, which makes the accumulator's value become the value of `M`.
- `getValue(?Variable)`, where `Variable` is an undetermined dataflow variable. This unifies the accumulator's value with `Variable`, and leaves the accumulator unchanged.

These are the only possible messages that can be sent to the Port returned by `NewAccumulator`.

The following are tests.

```
\insert 'TestingNoStop.oz'
\insert 'Accumulator.oz'
{StartTesting 'AccumulatorTest $Revision: 1.2 $'}
Acc = {NewAccumulator}  A2 = {NewAccumulator}
{Test {Send Acc getValue($)} '==' 0}  {Test {Send A2 getValue($)} '==' 0}
{Send Acc add(7)}  {Test {Send Acc getValue($)} '==' 7}
{Send Acc subtract(5)}  {Test {Send Acc getValue($)} '==' 2}
{Send Acc enter(10)}  {Test {Send Acc getValue($)} '==' 10}
{Send Acc square}  {Test {Send Acc getValue($)} '==' 100}
{Send Acc multiply(5)}  {Test {Send Acc getValue($)} '==' 500}
{Send Acc multiply(8)}  {Test {Send Acc getValue($)} '==' 4000}
{Send Acc add(20)}  {Test {Send Acc getValue($)} '==' 4020}
{Send Acc enter(0)}  {Test {Send Acc getValue($)} '==' 0}
{DoneTesting}
```

8. (15 points) [UseModels] Using either the declarative concurrent or the message passing model in Oz, write all of the following functions and procedures, which will implement a <Set> abstract data type:

```
NewSet : <fun {$ }: <Set>>
SetAdd : <proc {$ <Set> <Value>}>
SetDelete : <proc {$ <Set> <Value>}>
SetSize: <fun {$ <Set>}: <Int>>
SetMember: <fun {$ <Set> <Value>}: <Bool>>
```

These behave as follows.

- {NewSet} creates a new, empty set, and returns it.
- {SetAdd Set Val}, puts Val into the set represented by Set.
- {SetDelete Set Val}, removes Val from the set represented by Set.
- {SetSize Set}, returns the number of distinct values in the Set argument.
- {SetMember Set Val}, returns true just when Val is a member of the Set argument, and false otherwise.

The following are examples, written using the Test procedure from the homework. (There is space for your answer on the next page.)

```
\insert 'Set.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'SetTest $Revision: 1.2 $'}
S = {NewSet}
{Test {SetSize S} '==' 0}
{SetAdd S apl}
{SetAdd S cobol}
{Test {SetSize S} '==' 2}
{SetAdd S apl}
{Test {SetSize S} '==' 2}
{Test {SetMember S cobol} '==' true}
{Test {SetMember S apl} '==' true}
{Test {SetMember S python} '==' false}
{SetAdd S python}
{Test {SetMember S python} '==' true}
{Test {SetSize S} '==' 3}
{Test {SetMember S apl} '==' true}
{SetDelete S apl}
{Test {SetMember S apl} '==' false}
{Test {SetSize S} '==' 2}
Dynamics = {NewSet}
{Test {SetSize Dynamics} '==' 0}
{SetAdd Dynamics lisp}
{SetAdd Dynamics scheme}
{SetAdd Dynamics smalltalk}
{SetAdd Dynamics python}
{Test {SetSize Dynamics} '==' 4}
{SetAdd Dynamics python}
{Test {SetSize Dynamics} '==' 4}
{Test {SetMember Dynamics lisp} '==' true}
{SetAdd Dynamics lisp}
{Test {SetSize Dynamics} '==' 4}
{Test {SetMember Dynamics apl} '==' false}
{SetAdd Dynamics apl}
{Test {SetSize Dynamics} '==' 5}
{Test {SetMember Dynamics apl} '==' true}
{DoneTesting}
```

Please put your answer for the `Set` problem below.

9. (10 points) Using either the declarative concurrent or the message passing model in Oz, write the following functions and procedures to implement an abstract data type <ProcessLater>, which remembers a function and can run that function or change the remembered function:

```
ProcessLaterUsing : <fun {$ <fun {$ <Value>}: <Value>}: <ProcessLater>>
PLCompute : <fun {$ <ProcessLater> <Value>}: <Value>>
PLUseFun : <proc {$ <ProcessLater> <fun {$ <Value>}: <Value>}>
```

These behave as follows.

- {ProcessLaterUsing F} returns a new <ProcessLater> object, which remembers the one-argument function F.
- {PLCompute PL Arg}, runs the remembered function in PL on the argument Arg, and returns the result of that application.
- {PLUseFun PL G}, changes the function remembered by PL to the one-argument function G.

Your program should be such that it can process PLCompute calls in parallel to some extent, without forcing each call to wait for the previous call to completely finish. The following are examples.

```
\insert 'ProcessLater.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'ProcessLaterTest $Revision: 1.2 $'}
PL = {ProcessLaterUsing fun {$ X} X*X end}
{Test {PLCompute PL 3} '==' 9}  {Test {PLCompute PL 5} '==' 25}
{PLUseFun PL fun {$ Y} 1000*Y*Y + 10*Y + 1 end}
{Test {PLCompute PL 3} '==' 9031}  {Test {PLCompute PL 7} '==' 49071}
PL2 = {ProcessLaterUsing fun {$ X} X end}
{Test {PLCompute PL2 ok} '==' ok}  {Test {PLCompute PL 10} '==' 100101}
{PLUseFun PL2 fun {$ Ls} {Map Ls fun {$ X} X+2 end} end}
{Test {PLCompute PL2 [4 0 2 0]} '==' [6 2 4 2]} {Test {PLCompute PL2 nil} '==' nil}
{Test {PLCompute PL2 [10 20 30 40 50 60 70 80]} '==' [12 22 32 42 52 62 72 82]}
{DoneTesting}
```