Fall, 2008 Name: ____________________

COP 4020 — Programming Languages 1

Test on the Message Passing Model, the Relational Model, and Programming Models vs. Problems

Special Directions for this Test

This test has 6 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), the message passing model (chapter 5) or the relational model (chapter 9). The problem will say which model is appropriate. However, you must not use imperative features (such as cells and assignment) or the library functions `IsDet` and `IsFree`. 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`, `Max`, etc.) In the message passing model you can use `NewPortObject` and `NewPortObject2` as if they were built-in, and in the relational model you can use `Solve` as if it was built-in.

For Grading

Problem	Points	Score
1	10	
2	10	
3	10	
4	15	
5	35	
6	20	

1. (10 points) [Concepts]

   (i) Circle the letter of the correct answer, and (ii) give a brief explanation of why that answer is correct.

   (a) The `NewPort` and `Send` primitives can't be programmed in the declarative concurrent model as procedures.

   (b) The `NewPort` and `Send` primitives can be programmed in the declarative concurrent model as procedures, using streams and threads.

   (c) The `NewPort` and `Send` primitives can be programmed in the declarative model, as procedures, using closures.

2. (10 points) [Concepts]

   (i) Circle the letter of the correct answer, and (ii) give a brief explanation of why that answer is correct.

   (a) In the message passing model, messages are sent using the synchronous RMI protocol.

   (b) The message passing model makes it impossible to program synchronous message sends, such as those found in the RMI protocol.

   (c) The message passing model is deterministic and thus in programs written in that model there can be no race conditions.

   (d) Nondeterminism is an important feature of the message passing model, and is useful in many examples.

3. (10 points) [UseModels]

   Using Oz's message passing model, write a function `SynchServer` that takes no arguments and returns a stateless port object that can execute procedure closures that are passed to it. The returned port object responds to the message `synchronized(`P`)`, where P is a procedure that takes no arguments. When the port object receives this message, it simply executes P. Consider the following code, written using the `Test` function as in the homework.

```
declare
SS = {SynchServer}
{Send SS synchronized(proc {$} {Show 'start'} end)}
{Send SS
 synchronized(proc {$}
                 {Show 'outer-1'}
                 {Send SS
                  synchronized(proc {$} {Show 'inner-a'} {Delay 1000} {Show 'inner-b'} end)}
                 {Delay 1000}
                 {Show 'outer-2'}
              end)}
{Send SS synchronized(proc {$} {Show 'done for SS'} end)}
Count = {NewCell 0}
proc {IncCount} Count := @Count+1 {Delay 1000} Count := @Count+1 end
S2 = {SynchServer}
thread {Send S2 synchronized(IncCount)} end
thread {Send S2 synchronized(IncCount)} end
thread {Send S2 synchronized(proc {$} {Test @Count mod 2 '==' 0} end)} end
{Send S2 synchronized(proc {$} {Test @Count mod 2 '==' 0} end)}
```

   The output of this code must be like the following, in which executions of the procedures sent to the same server cannot overlap, although executions of procedures sent to different servers may overlap.

```
start
'outer-1'
0 == 0
'outer-2'
'done for SS'
'inner-a'
'inner-b'
0 == 0
```

   Please write your solution for problem 3 below.

```
\insert 'NewPortObject.oz'    % so you can use NewPortObject
\insert 'NewPortObject2.oz'   % so you can use NewPortObject2
declare
```

4. (15 points) [UseModels]

   Using Oz's message passing model, write a function `NewRepeatChecker` that takes no arguments and returns a port object that checks for repeated words. (This could be used in a grammar checker, or in a multi-player game.) The returned port object responds to the following messages:

   - `word(`A R`)`, where A is an atom, and R is an undetermined store variable.
   - `reset`.

   When the port object is first created, or immediately after processing the `reset` message, it is in an initial state.

   When the port object receives the `word(`A R`)` message, its response depends on its state. If it is in the initial state, then it unifies R with **`false`**, and goes into a state in which it remembers A. If it is not in the initial state, it has a word A' that it is remembering; in this state it unifies R with **`true`** just when A equals A' and unifies R with **`false`** otherwise.

   The following are some examples, written using the `Test` function as in the homework.

```
declare
MyRC = {NewRepeatChecker}
{Test {Send MyRC word(blah $)} '==' false}
{Test {Send MyRC word(blah $)} '==' true}
{Test {Send MyRC word(blah $)} '==' true}
{Send MyRC reset}
{Test {Send MyRC word(blah $)} '==' false}
{Test {Send MyRC word(foo $)} '==' false}
{Test {Send MyRC word(foo $)} '==' true}
{Test {Send MyRC word(blah $)} '==' false}
{Test {Send MyRC word(rosebud $)} '==' false}
{Send MyRC reset}
{Test {Send MyRC word(rosebud $)} '==' false}
{Test {Send MyRC word(rosebud $)} '==' true}
RC2 = {NewRepeatChecker}
{Test {Send RC2 word(rosebud $)} '==' false}
{Test {Send RC2 word(rosebud $)} '==' true}
{Test {Send RC2 word(the $)} '==' false}
{Test {Send RC2 word(good $)} '==' false}
{Test {Send RC2 word(good $)} '==' true}
{Test {Send RC2 word(ship $)} '==' false}
{Send RC2 reset}
{Send RC2 reset}
RC = {NewRepeatChecker}
{Send RC reset}
{Test {Send RC word(the $)} '==' false}
{Test {Send RC word(the $)} '==' true}
{Test {Send MyRC word(the $)} '==' false}
{Test {Send RC2 word(the $)} '==' false}
{Test {Send RC word(the $)} '==' true}
{Test {Send RC word(taking $)} '==' false}
{Test {Send RC word(up $)} '==' false}
{Test {Send RC word(vertical $)} '==' false}
{Test {Send RC word(space $)} '==' false}
```

   There is room for your solution on the next page

Please write your solution for problem 4 below.

```
\insert 'NewPortObject.oz'  % so you can use NewPortObject
declare
```

5. (35 points) [UseModels]

   Using Oz's message passing model, write a function `NewTaskManager` that takes no arguments and returns a port object that acts as a task manager. The returned port object responds to the following messages:

   - `schedule(`F`)`, where F is a function of no arguments, and
   - `request(`V`)`, where V is an undetermined store variable.

   The port object remembers schedule and request messages, and pairs up a schedule message with the oldest unpaired request message, unifying the variable V in the request message with the function F in the schedule message. That is, when the port object receives the `schedule(`F`)` message, it binds F to the variable V from the oldest unpaired request message received so far; if there is no such unpaired request, it just remembers F. Similarly, when the port object receives the `request(`V`)` message, it binds to the variable V the function F from the oldest unpaired schedule message received so far; if there is no such unpaired schedule message, it just remembers V.

   The following tests are written using the `Test` function as in the homework.

```
declare
fun {GenericTask Num} schedule(fun {$} task(Num) end) end % for testing
TM = {NewTaskManager}
{Send TM {GenericTask 1}}
{Send TM {GenericTask 2}}
{Send TM {GenericTask 3}}
{Send TM {GenericTask 4}}
{Send TM {GenericTask 5}}
local T1 in {Send TM request(T1)} {Test {T1} '==' task(1)} end
local T2 in T2={Send TM request($)} {Test {T2} '==' task(2)} end
{Send TM {GenericTask 6}}
{Test {{Send TM request($)}} '==' task(3)} % calls result of request, as above
{Test {{Send TM request($)}} '==' task(4)}
{Test {{Send TM request($)}} '==' task(5)}
thread {Test {{Send TM request($)}} '==' task(6)} end  % use threads so testing
thread {Test {{Send TM request($)}} '==' task(7)} end  % doesn't suspend
{Send TM {GenericTask 7}}
thread {Test {{Send TM request($)}} '==' task(8)} end
{Send TM {GenericTask 8}}
TM2 = {NewTaskManager}
thread {Test {{Send TM2 request($)}} '==' task(a)}
       {Test {{Send TM2 request($)}} '==' task(b)}
       {Test {{Send TM2 request($)}} '==' task(c)}
end
{Send TM2 {GenericTask a}}
{Send TM2 {GenericTask b}}
{Send TM2 {GenericTask c}}
thread {Test {{Send TM request($)}} '==' task(9)} end
{Send TM {GenericTask 9}}
```

   The output of the above looks like

```
task(1) == task(1)
task(2) == task(2)
% similarly for task(3)...task(8)
task(9) == task(9)
task(a) == task(a)
task(b) == task(b)
task(c) == task(c)
```

   There is room for your solution on the next page

Please write your solution for problem 5 below.

```
\insert 'NewPortObject.oz'   % so you can use NewPortObject
declare
```

6. (20 points) [EvaluateModels] For each of the following programming problems, (i) name the best programming model for solving the problem, and (ii) briefly explain why that model is best for the problem. Your answer should favor the least expressive model (i.e., the one with the fewest features) that can solve the problem. (Choose from among the programming models we studied this semester.)

   (a) A program that maintains traffic reports from different sensors in a metropolitan area (like Orlando). Reports from different sensors are short records that can arrive concurrently and at unpredictable times. Also, police and news organizations can independently send short queries to the program to find the status of various roadways in the area. The load is not expected to be too heavy, so the program only needs to handle one sensor report or query at a time.

   (b) A program that takes an XML record (which is tree-structured) and produces a similar XML record, but with a given record inserted as a new field in each subrecord with a given label. This program would be used as part of an XML editing application, which would supply the three arguments (the record, the new part, and the label), and would track the result.

   (c) A program that help your mother find a satisfactory seating arrangement for guests at a dinner party she is throwing at her home. She has helpfully typed into her computer facts describing which guests can sit next to what other guests, and which guests each person will not sit next to. (Some of her guests don't like each other.) But she can't find a good arrangement of the guests, as the problem is too complex for her (and for you) to do by hand. Her dinner party is next week and you are only home for the weekend.

   (d) A program that controls humidity sensor, that receives a potentially infinite series of readings (numbers) from the sensor, and has to filter them to smooth out the readings and reject temporarily senseless (spurious) readings that sometimes occur. The program simply produces a potentially infinite series of the filtered readings (which another, different program reads and logs).