Spring, 2009 Name: _____

COP 4020 — Programming Languages 1

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

Special Directions for this Test

This test has 6 questions and pages numbered 1 through 9.
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

Question:	1	2	3	4	5	6	Total
Points:	10	10	10	20	30	20	100
Score:							

1. (10 points) [Concepts] [EvaluateModels]

   How does the message passing model overcome the limitations of the declarative concurrent model?

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

   A. Threads can use dataflow variables to communicate messages to a server, since dataflow variables prohibit race conditions. Thus by simply setting a dataflow variable to different values at different times, a client can communicate many different values to a server, easily and efficiently.

   B. Threads can send messages to a port without synchronizing among themselves, and so one can easily write servers that read messages from different, independent clients. That is, there is no need to have a predetermined order to poll clients or to require synchronization among the clients.

   C. Threads can place messages into a single stream after they synchronize among themselves, in order to be sure that only one of them adds a message to a stream at a time. This preserves the determinism, while allowing one to write client-server systems that have multiple independent clients very easily and efficiently.

2. (10 points) [Concepts] This is a question about the `Send` primitive in Oz.

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

   A. A call to `Send` is a statement that waits for the message to be received. That is, the call to `Send` blocks until the message is used.

   B. A call to `Send` is an expression that waits for the message to be received. That is, the call to `Send` blocks until the message is used.

   C. A call to `Send` is a statement that does not wait for the message to be received and is buffered in the stream of the port to which the message is sent.

   D. A call to `Send` is an expression that does not wait for the message to be received and is buffered in the stream of the port to which the message is sent.

3. (10 points) [UseModels]

Using Oz's message passing model, write a function `NewGoodBooks` that takes no arguments and returns a port object that tracks book recommendations. The returned port object responds to the following messages:

- `recommend(`T`)`, where T is a string representing a book's title, and
- `check(`T X`)`, where T is a string representing a book's title and X is an undetermined dataflow variable.

The newly created port object has an empty set of recommended books.

When the port object receives the `recommend(`T`)` message, it adds the title T to its set of recommended books.

When the port object receives the `check(`T X`)` message, it unifies X with **`true`** if T is in the port object's current set of recommended books, and otherwise unifies X with **`false`**.

Hint: you can use Oz's built in `Member` function, which performs the proper check (using ==) on strings.

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

```
\insert 'NewGoodBooks.oz'
declare
NGB = {NewGoodBooks}
{Test local R in {Send NGB check("War and Peace" R)} R end '==' false}
{Send NGB recommend("War and Peace")}
{Test {Send NGB check("War and Peace" $)} '==' true}
{Test {Send NGB check("The Gulag Archipelago" $)} '==' false}
local X in
   thread {Send NGB recommend("The Brothers Karamazov")} X=unit end
   {Send NGB recommend("The Gulag Archipelago")}
   {Wait X}
end
{Test {Send NGB check("Harry Potter and the Goblet of Fire" $)} '==' false}
{Test {Send NGB check("The Brothers Karamazov" $)} '==' true}
{Test {Send NGB check("The Great Gatsby" $)} '==' false}
{Test {Send NGB check("The Brothers Karamazov" $)} '==' true}
{Test {Send NGB check("The Gulag Archipelago" $)} '==' true}
```

Please write your solution below.

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

4. (20 points) [Concepts] [UseModels]

   Using Oz's message passing model, write a function `NewDataflowVar` that takes no arguments and returns a port object that acts like an Oz dataflow variable. The returned port object responds to the following messages:

   - `unifyWith(`V K`)`, where V is a value, and K is an undetermined dataflow variable.
   - `read(`X`)`, where X is an undetermined dataflow variable.

   When the port object is first created, it represents an "undetermined" dataflow variable with an empty set of waiting variables.

   When the port object receives the `unifyWith(`V K`)` message, its response depends on its state. If it is in a state representing an "undetermined" dataflow variable with a set of waiting variables S, then it unifies K with **`true`**, unifies each variable s in S with V, and goes into a state that remembers the value V. If it is already in a state that is remembering a value V', then its response to the `unifyWith(`V K`)` message is to unify K with **`true`** if V == V' and with **`false`** otherwise.

   When the port object receives the `read(`X`)` message, its response also depends on its state. If it is in a state representing an "undetermined" dataflow variable with a set of waiting variables S, then it goes into the same kind of state with the set of waiting variables $\{X\} \cup S$. If it is in a state that is remembering a value V', then it unifies X with V' and stays in the same state.

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

```
\insert 'NewDataflowVar.oz'
declare
V1 = {NewDataflowVar}
{Test {Send V1 unifyWith(4020 $)} '==' true}
{Test local R in {Send V1 read(R)} R end '==' 4020}
{Test {Send V1 unifyWith(4020 $)} '==' true}
{Test {Send V1 unifyWith(99 $)} '==' false}

V2 = {NewDataflowVar}
local A B C in
  thread {Send V2 read(A)} end
  thread {Send V2 read(B)} end
  thread {Send V2 read(C)} {Test C '==' atom} end
  {Delay 200}
  {Test {Send V2 unifyWith(atom $)} '==' true}
  {Test {Send V2 read($)} '==' atom}
  {Test A == atom andthen B == atom '==' true}
  {Test {Send V2 unifyWith(nope $)} '==' false}
end
{Test {Send V1 read($)} '==' 4020}

V3 = {NewDataflowVar}
local D E F in
  thread {Send V3 read(D)} end
  thread {Send V3 read(E)} end
  {Delay 200}
  {Test {Send V3 unifyWith(mostlyHarmless $)} '==' true}
  thread {Test {Send V3 read($)} '==' mostlyHarmless} end
  {Test D '==' mostlyHarmless}
  {Test E '==' mostlyHarmless}
  {Test {Send V3 unifyWith(mostlyHarmless $)} '==' true}
  {Test {Send V3 unifyWith(diff $)} '==' false}
end
```

   Hint: you can (and should) use Oz's dataflow variables in your solution. You can represent sets using a list.

   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. (30 points) [UseModels]

Using Oz's message passing model, write a function `NewModel` that takes an initial value of some type `T` as an argument and returns a port object that acts as "model" (as in the model-view-controller technique for user interfaces). The returned port object tracks a list of observer ports (which represent the views) and a value of type `T` that represents the model's current state. The returned port object responds to the following messages, which are designed to minimize the amount data transmitted.

- `register(`O`)`, where O is a port, which can be used to notify an observer,
- `change(`F`)`, where F is a function of type `<fun {$ T}: T>` that takes a value of type `T`, representing the model's current state, and returns a value of that type,
- `query(`G X`)`, where G is a function that takes an argument of type `T` and an undetermined dataflow variable X.

The port object returned remembers a list L of observer ports and a value V of type `T`. Initially L is `nil`, and V is the initial value passed to `NewModel`.

When the port object receives the `register(`O`)` message, it adds O to the list of observers L, as its first element (and where the model's value V is unchanged).

When the port object receives the `change(`F`)` message, it notifies each observer in the list L, starting with the head of L, by sending to the port recorded in L the message `changed(`S`)`, where S denotes the port object receiving the message `change(`F`)`. It then goes into a new state where the new value V' is the result of applying F to the old value V (and has the same set of observers).

When the port object receives the `query(`G X`)` message, it unifies X with the result of applying G to the model's current state V. (The set of observers and the model's state are unchanged.)

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

```
\insert 'NewModel.oz'
\insert 'NewPortObject2.oz'
% Some functions for testing purposes...
declare
fun {CNumOp Op K}
   fun {$ I} {Op I K} end
end
Mul10 = {CNumOp Number.'*' 10}
Id = {CNumOp Number.'+' 0}
Add2 = {CNumOp Number.'+' 2}
Add3 = {CNumOp Number.'+' 3}

% Simple tests where the model is an integer.
% These simple tests show queries on the model's value.
MyInt = {NewModel 0}
{Send MyInt change(Add3)}
{Test local R in {Send MyInt query(Mul10 R)} R end '==' 30}
{Test {Send MyInt query(Add2 $)} '==' 5}
{Send MyInt change(Add3)}
{Test {Send MyInt query(Add2 $)} '==' 8}

% More tests below, which show how observers work...
```

```
% Tests that use "passive observers" that simply record the messages sent
fun {NewPassiveObserver Model}
   local P Strm in
      P = {NewPort Strm}
      {Send Model register(P)}  % register my port with the model
      Strm
   end
end
MyNewInt = {NewModel 0}
ObsStrm1 = {NewPassiveObserver MyNewInt}
Strm2 = {NewPassiveObserver MyNewInt}
{Send MyNewInt change(Add3)}
{Send MyNewInt change(Mul10)}
{Test {List.take ObsStrm1 2} '==' [changed(MyNewInt) changed(MyNewInt)]}
{Test {List.take Strm2 2} '==' [changed(MyNewInt) changed(MyNewInt)]}
{Test {Send MyNewInt query(Id $)} '==' 30}

% Tests that use "callback observers" that send queries (callbacks
% with the query message, in response to receiving a changed message.
fun {NewCallbackObserver Model}
   local ResultStrm
      ResultPort = {NewPort ResultStrm}
      Adapter = {NewPortObject2
                 proc {$ changed(Port)}
                    local NewVal in
                       {Send Port query(fun {$ R} R.x + R.y end NewVal)}
                       {Send ResultPort NewVal}
                    end
                 end}
   in
      {Send Model register(Adapter)}  % register with the model
      ResultStrm
   end
end
MyNewRec = {NewModel point(x: 0 y: 0)}
fun {Add5ToX point(x: X y: Y)} point(x: 5+X y: Y) end
fun {Add100ToY point(x: X y: Y)} point(x: X y: 100+Y) end
POStrm = {NewPassiveObserver MyNewRec}
CBStrm = {NewCallbackObserver MyNewRec}
{Send MyNewRec change(Add5ToX)}
{Test {List.take POStrm 1} '==' [changed(MyNewRec)]}
{Test {List.take CBStrm 1} '==' [5]}
{Send MyNewRec change(Add100ToY)}
{Test {List.take CBStrm 2} '==' [5 105]}
{Send MyNewRec change(Add100ToY)}
{Test {List.take CBStrm 3} '==' [5 105 205]}
```

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. [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) (5 points) A program that takes raw data on economic statistics, such as prices and rates of unemployment, and uses historical data to filter out seasonal variations (such as prices for toys increasing near Christmas, or that housing sales increase in summer). The program receives data in an infinite stream, and filters them to adjust for such variations. The program produces a potentially infinite stream of the adjusted statistics.

   (b) (5 points) A program that tracks reports from a state's bird watchers. The reports from different bird watchers are small records that detail what species of birds have been seen. These reports arrive concurrently and at unpredictable times. Also, scientists and park rangers can independently send short queries to find information about what birds have been seen. Bird watching is not something a lot of people do, so the program only needs to handle one report or query at a time.

   (c) (5 points) A program to generate a list of menus for a cafeteria, which considers several constraints such as not repeating meals within a certain time period, and ensuring that the number of calories of each meal is within a certain range. The owners have a file containing about 500 dishes in their recipe book that can be combined to make up menus. The owners of the cafeteria are only willing to pay you for 2 hours worth of work, as they are not sure what kind of constraints and menus they really want.

   (d) (5 points) A program that takes a tree-structured encoding of a web page's HTML and produces a similar tree-structured encoding with each instance of a particular search term highlighted. This program would be used as part (component) of an search engine, which needs to produce such web pages.