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 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) or the message passing model (chapter 5). The problem will say which model(s) are 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`, `Min`, etc.) In the message passing model you can use `NewPortObject` and `NewPortObject2` as if they were built-in.

For Grading

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

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

   This question compares the computational power of the message passing model and the explicit state model (the one with cells).

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

   A. The explicit state model is not as powerful as the message passing model, since one cannot use cells to simulate the message passing primitives `NewPort` and `Send`.

   B. The explicit state model is at least as powerful as the message passing model, since one can use cells to simulate the message passing primitives `NewPort` and `Send`.

   C. The explicit state model is strictly more powerful than the message passing model, because one can use cells to simulate the message passing primitives `NewPort` and `Send`, but one cannot use these primitives of the message passing model to simulate the operations of a cell.

2. (10 points) [UseModels]

   Using Oz's message passing model, write a function `NewAccumulator` that takes one argument and returns a port object that accumulates the results of apply messages (see below), starting with the argument as the accumulator's initial value. The returned port object responds to the following messages:

   - `apply(Fun)`, which applies the one-argument function `Fun` to the accumulator's value, and makes the accumulator remember the result as its new accumulator value, and
   - `fetch(`X`)`, where X is an undetermined dataflow variable.

   The result of `{NewAccumulator Init}` is a port object that remembers `Init` as its accumulator value.

   When the port object receives the `apply(Fun)` message, it applies `Fun` to the accumulator's value, and uses the result of that call as its new accumulator value.

   When the port object receives the `fetch(`X`)` message, where X is an undetermined dataflow variable, it unifies X with the current accumulator value, and leaves the accumulator value unchanged.

   The following are tests, written using the `Test` function from the homework.

```
\insert 'NewAccumulator.oz'
\insert 'TestingNoStop.oz'
declare
MyAcc = {NewAccumulator 7}
{Test {Send MyAcc fetch($)} '==' 7}
{Send MyAcc apply(fun {$ Val} Val+100 end)}
{Test {Send MyAcc fetch($)} '==' 107}
{Send MyAcc apply(fun {$ Val} Val+1 end)}
{Test {Send MyAcc fetch($)} '==' 108}
{Send MyAcc apply(fun {$ Val} 2*Val+1000 end)}
{Test {Send MyAcc fetch($)} '==' 1216}
{Send MyAcc apply(fun {$ _} 1 end)}
{Test {Send MyAcc fetch($)} '==' 1}
A2 = {NewAccumulator 3}
{Test {Send A2 fetch($)} '==' 3}
{Send A2 apply(fun {$ Val} Val+4017 end)}
{Test {Send A2 fetch($)} '==' 4020}
{Test {Send MyAcc fetch($)} '==' 1}
{Send A2 apply(fun {$ Val} val_is(Val) end)}
{Test {Send A2 fetch($)} '==' val_is(4020)}
{Send A2 apply(fun {$ val_is(Val)} val_is(Val div 10) end)}
{Test {Send A2 fetch($)} '==' val_is(402)}
{Send A2 apply(fun {$ val_is(Val)} Val - 360 end)}
{Test {Send A2 fetch($)} '==' 42}
```

   Please write your answer below.

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

3. (20 points) [UseModels]

Using Oz's message passing model, write a function `NewHeapSort` that takes no arguments and returns a port object that can track a multiset of data and return the minimum element of the current data. The returned port object responds to the following messages:

- `add(Data)`, where `Data` is a value of some comparable type, which is added to the data set, and
- `getMin(`X`)`, where X is an undetermined dataflow variable.

The newly created port object has an empty multiset of data.

When the port object receives the `add(Data)` message, it remembers `Data` as part of its data set.

When the port object receives the `getMin(`X`)` message, it binds the undetermined dataflow variable X to the minimum value in the data set, and change its data set to have one less occurrence of that value.

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

```
\insert 'NewHeapSort.oz'
\insert 'TestingNoStop.oz'
declare
HS = {NewHeapSort}
{Send HS add(9)}
{Send HS add(3)}
{Send HS add(22)}
{Send HS add(1)}
{Send HS add(10)}
{Send HS add(6)}
{Send HS add(0)}
{Send HS add(333)}
{Test {Send HS getMin($)} '==' 0}
{Test {Send HS getMin($)} '==' 1}
{Test {Send HS getMin($)} '==' 3}
{Test {Send HS getMin($)} '==' 6}
{Test {Send HS getMin($)} '==' 9}
{Test {Send HS getMin($)} '==' 10}
{Test {Send HS getMin($)} '==' 22}
{Test {Send HS getMin($)} '==' 333}

HS2 = {NewHeapSort}
{Send HS2 add(hmm)}
{Send HS2 add(a_sym)}
{Send HS2 add(cod)}
{Send HS2 add(tuna)}
{Send HS2 add(herring)}
{Send HS2 add(cod)}
{Send HS2 add(lobster)}
{Send HS2 add(zebra)}
{Send HS2 add(alphabet_soup)}
{Test {Send HS2 getMin($)} '==' a_sym}
{Test {Send HS2 getMin($)} '==' alpahbet_soup}
{Test {Send HS2 getMin($)} '==' cod}
{Test {Send HS2 getMin($)} '==' cod}
{Test {Send HS2 getMin($)} '==' herring}
{Test {Send HS2 getMin($)} '==' hmm}
{Test {Send HS2 getMin($)} '==' lobster}
{Test {Send HS2 getMin($)} '==' tuna}
{Test {Send HS2 getMin($)} '==' zebra}
```

Hint, you can use Oz's built in `Min` function in your solution. (Despite the name of this problem, you don't have to use a "heap" data structure, and your program doesn't need to have optimal performance.)

Please write your solution on the next page.

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

4. (20 points) [UseModels]

Using Oz's message passing model, write a function `NewTVGuide` that takes no arguments and returns a port object that can track television (TV) programs (for a single day) and can answer requests about what is on at a specific time. The returned port object responds to the following messages:

- `addSchedule(time: Time channel: Chan show: Name)`, which adds to the tracked schedule information that at the given `Time` (an `<Int>`), on the given channel `Chan` (an `<Atom>`), that the show named `Name` (a `<String>`) is playing, and
- `whatsOn(Time` X`)`, where `Time` is an `<Int>` giving the time that is sought, and X is an undetermined dataflow variable.

The newly created port object has an empty schedule.

When the port object receives the `addSchedule(time: Time channel: Chan show: Name)` message, it remembers that at the given `Time`, on the given channel `Chan`, the show named `Name` is playing.

When the port object receives the `whatsOn(Time` X`)` message, it should bind the undetermined dataflow variable X to a list of program records of the form `program(channel: Chan show: Name)`, each of which means that at the given `Time`, on channel `Chan` (an `<Atom>`), the show named `Name` (a `<String>`) is playing.

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

```
\insert 'NewTVGuide.oz'
declare
TVG = {NewTVGuide}
{Test local R in {Send TVG whatsOn(7 R)} R end '==' nil}
{Test {Send TVG whatsOn(20 $)} '==' nil}
{Send TVG addSchedule(time: 7 channel: hdthtr show: "Sunrise Earth")}
{Send TVG addSchedule(time: 7 channel: nbc show: "Today")}
{Send TVG addSchedule(time: 7 channel: cbs show: "Good Morning America")}
{Send TVG addSchedule(time: 20 channel: comcent show: "The Daily Show")}
{Send TVG addSchedule(time: 21 channel: comcent show: "South Park")}
{Test {Send TVG whatsOn(7 $)}
 '==' [program(channel: cbs show: "Good Morning America")
       program(channel: nbc show: "Today")
       program(channel: hdthtr show: "Sunrise Earth")]}
{Test {Send TVG whatsOn(20 $)}
 '==' [program(channel: comcent show: "The Daily Show")]}

{Send TVG addSchedule(time: 20 channel: pbs show: "War and Peace")}
{Test {Send TVG whatsOn(20 $)}
 '==' [program(channel: pbs show: "War and Peace")
       program(channel: comcent show: "The Daily Show")]}

TVG2 = {NewTVGuide}
{Test {Send TVG2 whatsOn(7 $)} '==' nil}
{Send TVG2 addSchedule(time: 7 channel: fox show: "Fox and Friends")}
{Test {Send TVG2 whatsOn(7 $)}
 '==' [program(channel: fox show: "Fox and Friends")]}
{Send TVG2 addSchedule(time: 13 channel: nbc show: "Generic Hospital")}
{Send TVG2 addSchedule(time: 13 channel: cbs show: "As the Stomach Turns")}
{Send TVG2 addSchedule(time: 13 channel: abc show: "Soapy Soap Dish")}
{Test {Send TVG2 whatsOn(13 $)}
 '==' [program(channel: abc show: "Soapy Soap Dish")
       program(channel: cbs show: "As the Stomach Turns")
       program(channel: nbc show: "Generic Hospital")]}
```

Please write your solution on the next page, where there is more space.

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

5. (20 points) [UseModels]

Using Oz's message passing model, write a function `NewResourceArbiter` that takes no arguments and returns a port object that tracks the status of a some resource (which resource is not important for this problem). The returned port object responds to the following messages:

- `query(`X`)`, where X is an undetermined dataflow variable,
- `reserve(`X`)`, where X is an undetermined dataflow variable, and
- `release`.

The port object should track a list of (undetermined) dataflow variables that have been sent to it in `reserve(`X`)` messages but not yet granted the resource, and it also should track the current status of the resource. The resource status can be either: "in use" or "not used." A newly created port object starts with the resource not used.

When the port object receives the `query(`X`)` message, it unifies X with an atom representing the current status of the resource, which will be either `inUse` (if the resource is being used) or `notUsed` (if the resource is not being used).

When the port object receives the `reserve(`X`)` message, the resource status becomes "in use" if it is not already, but what happens to X and the list of variables waiting depends on the resource's current status. If the resource is not currently in use, then it binds X to some atom, allowing the sender, which is a thread that should be waiting for X to be determined, to proceed. Otherwise, if the resource is in use, then the port object puts X at the end of the list of variables that represent processes waiting to use the resource.

When the port object receives the `release` message, what happens depends on the list of dataflow variables representing waiting threads. If that list is empty, then the resource status changes to being not used. If that list has some elements, then the first element in the list is unified with some atom (e.g., **`unit`**), which lets the thread that is waiting on that dataflow variable proceed, and that variable is taken out of the list of waiting variables and the resource remains in use.

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

```
\insert 'NewResourceArbiter.oz'
\insert 'TestingNoStop.oz'
declare
proc {Acquire Port} % to avoid repeated testing code, for testing only
   local WaitVar in {Send Port reserve(WaitVar)} {Wait WaitVar} end
end

RA = {NewResourceArbiter}
{Test {Send RA query($)} '==' notUsed}
{Acquire RA}
{Test {Send RA query($)} '==' inUse}
{Send RA release}
{Test {Send RA query($)} '==' notUsed}

RA2 = {NewResourceArbiter}
local TestStrm P={NewPort TestStrm} in
   for ID in 1..2 do
      thread {Acquire RA2} {Send P ID} {Delay 5} {Send P ID} {Send RA2 release} end
   end
   local T4 = {List.take TestStrm 4} in
      {Test (T4 == [1 1 2 2] orelse T4 == [2 2 1 1]) '==' true}
   end
end
```

Please write your solution on the next page, where there is more space.

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

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 receives a three dimensional model of a cartoon character, and a motion plan, and produces an infinite stream of three dimensional models that show the character moving according to the plan.

   (b) (5 points) A program that takes tree-structured XML data encoding an RSS feed and produces a web page (in tree-structured HTML) with the same information. This program could be used as a part of a web browser, which needs to produce such web pages from RSS feeds.

   (c) (5 points) A program that takes raw data on ocean temperatures from a single sattelite, and uses a model of the sattelite's temperature sensor to filter out abnormalities in its readings (such as occur when the sun is directly behind the sattelite). The program receives data in an infinite stream, and filters them to remove the abnormal (i.e., suspect) readings. The program produces a potentially infinite stream of the adjusted temperature data.

   (d) (5 points) A program that receives reports on traffic conditions from many independent traffic sensors and police units throughout a major metropolitan area. These reports may arrive concurrently and at unpredictable times. Also, web clients, and media outlets can independently send queries to find information about traffic conditions. It is expected that the volume of queries will be fairly light.