Homework 5: Message Passing

See Webcourses and the syllabus for due dates.

In this homework you will learn about the message passing model and basic techniques of programming in that model. The programming techniques include using port objects to create agents (state machines) [Concepts] [UseModels]. A few problems also make comparisons with the other models we have studied, and also with message passing features in other languages [EvaluateModels] [MapToLanguages].

Answers to English questions should be in your own words; don't just quote text from the textbook.

We will take some points off for duplicated code, code with extra unnecessary cases, or code that is excessively hard to follow. You should always assume that the inputs given will follow the grammar for the types specified, and so your code should not have extra cases for inputs that do not follow the grammar. Avoid duplicating code by using helping functions or by using syntactic sugars and local definitions. It is a good idea to check your code for these problems before submitting.

Your code should be written in the message passing model, so unless we specify otherwise, you must *not* use cells and assignment in your Oz solutions. Furthermore, note that the message passing model does *not* include the primitive `IsDet` or the library function `IsFree`; thus you are also prohibited from using either of these functions in your solutions. Although you can simulate cells using the message passing model (see problem 4), you should avoid using this simulation directly, instead use a port object's state directly. (You can also store state in the message queue of a port object.) Using the model as it is intended will make it clear what the state of each port object is.

You should use helping functions whenever you find that useful. Unless we specifically say how you are to solve a problem, feel free to use any non-mutating functions from the Oz library (base environment), especially functions like `Map` and `FoldR`. Of course you can use `NewPortObject` and `NewPortObject2`.

For all Oz programming exercises, you must run your code using the Mozart/Oz system. For programming problems for which we provide tests, you can find them all in a zip file, which you can download from problem 1's assignment on Webcourses.

If the tests don't pass, please try to say why they don't pass, as this enhances communication and makes commenting on the code easier and more specific to your problem.

What to Turn In: For each problem that requires code, turn in (on Webcourses) your code and output of your testing. Please upload code as a plain (text) file with the name given in the problem or testing file and with the suffix `.oz`, and also paste the output from our tests into the answer box on webcourses. For English answers, please paste your answer into the answer box in the problem's "assignment" on Webcourses. For a problem with a mix of code and English, follow both of the above.

Your code should compile with Oz, if it doesn't you probably should keep working on it. Email the staff with your code file if you need help getting it to compile. If you don't have time, at least tell us that you didn't get it to compile. Don't hesitate to contact the staff if you are stuck at some point.

Read Chapter 5 of the textbook [VH04]. (See the syllabus for optional readings.)

Reading Problems

The problems in this section are intended to get you to read the textbook, ideally in advance of class meetings.

Read chapter 5, through section 5.1 of the textbook [VH04] and answer the following questions.

Read chapter 5, through section 5.2 of the textbook [VH04] and answer the following questions. For examples of uses of `NewPortObject`, see the code examples page.

1. (2 points) [Concepts] [UseModels] By default, does a server created by using `NewPortObject.oz` or `NewPortObject2.oz` process messages concurrently, or one at a time?

Regular Problems

For these probems, use the `NewPortObject.oz` or `NewPortObjectDebug.oz` files that are included with the homework's testing files. The `NewPortObjectDebug.oz` is handy for seeing what is happening (i.e., for debugging).

Message Passing Semantics and Expressiveness

For the problems in this subsection, see especially sections 5.1 and 5.2 of the textbook [VH04].

2. (15 points) [UseModels] Using the message passing model, write a function:

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

   that takes no arguments and returns a port object. The returned port object handles the following massages, for some type `T`:

   - `push(Val)`, which contains a value, `Val`, of type `T`.
   - `pop`,
   - `size(?Variable)`, which contains an undetermined dataflow variable,
   - `top(?Variable)`, which contains an undetermined dataflow variable.

   Sending the port object the message `push(Val)`, where `Val` is some value of type `T` pushes `Val` onto the top of the stack.

   Sending the port object the message `pop` pops the top value off of the top of the stack.

   Sending the port object the message `top(Variable)`, where `Variable` is an undetermined dataflow variable, unifies `Variable` with the value on the top of the stack and leaves the stack's state unchanged.

   Sending the port object the message `size(Variable)`, where `Variable` is an undetermined dataflow variable, unifies `Variable` with the number of values on the stack.

   You can assume that the port object never receives the `pop` or the `top` message when the stack is empty (has size 0).

   You must use `NewPortObject` in your solution (see the textbook, and the `NewPortObject.oz` file supplied with the test cases for this homework).

   Figure 1 on the following page contains examples.

```
% $Id: NewStackTest.oz,v 1.2 2012/04/12 14:59:55 leavens Exp leavens $
\insert 'NewStack.oz'
\insert 'TestingNoStop.oz'
declare
{StartTesting 'NewStackTest $Revision: 1.2 $'}
MyStack = {NewStack}
S2 = {NewStack}
{Test {Send MyStack size($)} '==' 0}
{Send MyStack push(first)}
{Test {Send MyStack size($)} '==' 1}
{Test {Send MyStack top($)} '==' first}
{Send MyStack push(second)}
{Test {Send MyStack size($)} '==' 2}
{Test {Send MyStack top($)} '==' second}
{Send MyStack pop}
{Test {Send MyStack size($)} '==' 1}
{Test {Send MyStack top($)} '==' first}
{Send MyStack push(third)}
{Send MyStack push(fourth)}
{Send MyStack push(fifth)}
{Test {Send MyStack size($)} '==' 4}
{Test {Send MyStack top($)} '==' fifth}
{Send MyStack pop}
{Test {Send MyStack size($)} '==' 3}
{Test {Send MyStack top($)} '==' fourth}
{Send MyStack pop}
{Send MyStack pop}
{Test {Send MyStack size($)} '==' 1}
{Test {Send MyStack top($)} '==' first}
{Send MyStack pop}
{Send S2 push(4020)}
{Test {Send S2 size($)} '==' 1}
{Test {Send S2 top($)} '==' 4020}
{Test {Send MyStack size($)} '==' 0}
{DoneTesting}
```

Figure 1: Testing for problem 2.

3. (0 points) (suggested practice) [UseModels]

   Using Oz's message passing model, write a function `NewCounter` that takes no arguments and returns a port object. The returned port object should store an integer value in its state and can respond to the messages `inc` and `value(Var)`. Initially the value stored in the port object is 0. The message `inc` increments this value (adds 1 to it). The `value(Var)` message binds `Var` to the current value in the port object; you can assume that `Var` is undetermined. Figure 2 contains some examples.

   You must use `NewPortObject` in your solution (see the textbook, and the `NewPortObject.oz` file supplied with the test cases for this homework).

```
% $Id: NewCounterTest.oz,v 1.4 2012/04/12 14:59:55 leavens Exp leavens $
\insert 'NewCounter.oz'
\insert 'TestingNoStop.oz'
declare
{StartTesting 'NewCounterTest $Revision: 1.4 $'}
MyCounter = {NewCounter}
{Test local R in {Send MyCounter value(R)} R end '==' 0}
{Send MyCounter inc}
{Test {Send MyCounter value($)} '==' 1}
{Test local Z in {Send MyCounter value(Z)} Z end '==' 1}
{Send MyCounter inc}
{Test {Send MyCounter value($)} '==' 2}
{Send MyCounter inc}
{Send MyCounter inc}
{Test {Send MyCounter value($)} '==' 4}

MyCounter2 = {NewCounter}
{Test {Send MyCounter2 value($)} '==' 0}
{Send MyCounter2 inc}
{Test {Send MyCounter2 value($)} '==' 1}
{Send MyCounter2 inc}
{Test {Send MyCounter2 value($)} '==' 2}
for I in 1..12 do {Send MyCounter2 inc} end
{Test {Send MyCounter2 value($)} '==' 14}
{DoneTesting}
```

Figure 2: Testing code for Problem 3.

4. (30 points) [Concepts] [UseModels]

   Using Oz's message passing model, implement a data abstraction `<PCell>`, by writing the following functions and procedures.

```
NewPCell: <fun {$ <Value>}: <PCell>>
PCellSwap: <proc {$ <PCell> <Value> <Value>}>
PCellSet: <proc {$ <PCell> <Value>}>
PCellGet: <fun {$ <PCell>}: <Value>>
```

   A PCell is like a Cell, in that it holds a value (of any type). The function call `{NewPCell X}` returns a new port object representing a PCell containing the value `X`. The procedure call `{PCellSwap PC Old New}` atomically binds `Old` to the value in PCell `PC` and makes `New` be the new value contained in PCell `PC`. The procedure call `{PCellSet PC V}` makes the value `V` be the new value of PCell `PC`. The function call `{PCellGet PC}` returns the value contained in PCell `PC`.

   Note that `PCellSwap` and `PCellSet` are **proc**s, not **fun**s. If you make them **fun**s instead, your code will not work with the tests!

You must use the `NewPortObject` function, given in the book and supplied with the test cases for this homework, in your solution. You must represent a PCell with a port object, so have `NewPCell` return the port of the port object, and have the other functions send messages to that port. The state of the port object will be the PCell's value.)

You are *not* allowed to use cells in your solution!

Your code should pass the tests shown in Figure 3 on the next page.

5. (0 points) (suggested practice) [Concepts] [UseModels]

   Using Cells, but without using the message passing primitives `NewPort` and `Send`, define in Oz an ADT `PortAsCell`, which acts like the built-in port type, but is represented as a Cell. (For more about the imperative model and cells, look back at Section 1.12 of the textbook or forward to chapter 6.) The `PortAsCell` ADT has two operations:

   ```
   MyNewPort: <fun {$ <Stream>}: <PortAsCell>>
   MySend: <proc {$ <PortAsCell> <Value>}>,
   ```

   which are intended to act like `NewPort` and `Send`. That is, the function `MyNewPort` takes an undetermined store variable, and returns a `<PortAsCell>`, which is a Cell that we want to act like a Port. The procedure `MySend` takes such a `PortAsCell` and a Value and adds the Value to the corresponding stream. In other words, the idea behind the ADT `<PortAsCell>` is that it should act like the built-in Port ADT of Oz, but be represented using a Cell.

   For this problem, don't worry about catching or prohibiting improper uses of the stream argument to `MyNewPort` (although that is part of the semantics of ports). Also for the moment, don't worry about potential race conditions when multiple threads are used.

   Since `MyNewPort` should act like `NewPort` and you are representing ports as cells, have `MyNewPort` return a cell containing the undetermined (stream) variable passed to it. Since `MySend` should act like `Send`, it should follow the semantics given in section 5.1 of the textbook. That is, it should extract the undetermined variable that represents the (old) end of the stream from the cell passed to it (in the first argument) and then it should make a new dataflow variable to hold the new end of the stream, put that new dataflow variable in the cell, and then unify the old end of the stream with a '`|`'-record that holds the data and new end of the stream variable.

   Your code should pass the tests shown in Figure 4 on page 7.

   You are *not* allowed to use `Send`, `NewPort`, or functions that call them (such as `NewPortObject`) in your solution, and you *must* use `Cell`s in your solution.

```
% $Id: PCellTest.oz,v 1.5 2012/04/12 14:59:55 leavens Exp leavens $
\insert 'PCell.oz'
\insert 'TestingNoStop.oz'
declare % just once!
PC1 PC2 V1old V2old V1x V2x % these are variables being declared
{StartTesting 'PCellTest $Revision: 1.5 $'}
PC1 = {NewPCell 1}
PC2 = {NewPCell 2}
{PCellSwap PC1 V1old 7}
{PCellSwap PC2 V2old 99}
{Test V1old '==' 1}
{Test V2old '==' 2}
{PCellSwap PC1 V1x 88}
{PCellSwap PC2 V2x 333}
{Test V2x '==' 99}
{Test V1x '==' 7}
EE = {PCellGet PC1}
{Test EE '==' 88}
TTT0 = {PCellGet PC2}
{Test TTT0 '==' 333}
TTT1 = {PCellGet PC2}
{Test TTT1 '==' 333}
EE2 = {PCellGet PC1}
{Test EE2 '==' 88}
{PCellSet PC1 4}
Four = {PCellGet PC1}
{Test Four '==' 4}
TTT = {PCellGet PC2}
{Test TTT '==' 333}
{PCellSet PC1 asymbolliteral}
ASL = {PCellGet PC1}
{Test ASL '==' asymbolliteral}
X=1 Y=2 Z=3
PC={NewPCell Z}
{StartTesting 'Some equations'}
{Test {PCellGet {NewPCell X}} '==' X}
{PCellSet PC Y}
End = {PCellGet PC}
{Test End '==' Y}
% Waits make the done message come out after all the others.
% {Wait TTT0} {Wait TTT1} {Wait ASL} {Wait EE} {Wait EE2} {Wait Four}
% {Wait TTT} {Wait End}
{DoneTesting}
```

Figure 3: Testing code for Problem 4 on page 4.

```
% $Id: PortAsCellTest.oz,v 1.5 2012/04/12 14:59:55 leavens Exp leavens $
\insert 'PortAsCell.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'PortAsCellTest $Revision: 1.5 $'}
% Simulating basic semantics of NewPort and Send
declare Strm Port in
Port = {MyNewPort Strm}
{StartTesting 'MySend'}
{MySend Port 3}
{MySend Port 4}
% Must use List.take, otherwise Test suspends...
{Test {List.take Strm 2} '==' [3 4]}
{MySend Port 5}
{MySend Port 6}
{Test {List.take Strm 4} '==' [3 4 5 6]}

{StartTesting 'MyNewPort second part'}
declare S2 P2 U1 U2 in
P2 = {MyNewPort S2}
{StartTesting 'MySend second part'}
{MySend P2 7}
{MySend P2 unit}
{MySend P2 true}
{MySend P2 U1}
{MySend P2 hmmm(x:U2)}
U1 = 4020
{Test {List.take S2 5} '==' [7 unit true 4020 hmmm(x:U2)]}
{Test {List.take Strm 4} '==' [3 4 5 6]}
{DoneTesting}
```

Figure 4: Testing code for Problem 5 on page 5. Note that this is only a suggested practice problem, so you don't have to write this if you don't want to practice!

Message Passing Programming

For additional examples of message passing programming, see also the code examples page.

6. (20 points) [UseModels]

   Using either the Oz message passing model, implement the abstract datatype `<Dropbox T>`, with the following functions and procedures, for any type `T`:

   ```
   NewDropbox: <fun {$ }: <Dropbox T> >
   DBDeposit: <proc {$ <Dropbox T> <Atom> <T>}>
   DBDelete: <proc {$ <Dropbox T> <Atom>}>
   DBFetch: <fun {$ <Dropbox T> <Atom>}: <T> >
   DBNames: <fun {$ <Dropbox T>}: <List Atom> >
   ```

   The state of a dropbox is a set of mappings, where each mapping takes an atom (a name) to a value of type `T`. When created by `NewDropbox`, it has an empty set of mappings. When a named value is deposited in it, its state changes to add the mapping from the given name to the given value. The functions and procedures whose types are above act as follows.

   - The function `NewDropbox`, when called with no arguments, returns a new `<Dropbox T>`.
   - The procedure `DBDeposit` takes three arguments, a `<Dropbox T>`, `DB`, an atom, `Name`, and a value of type `T`, `Value`. It changes the state of `DB` so that `DB` remembers the mapping from `Name` to `Value`. This also causes any waiting calls to `DBFetch` to complete and return `Value`.
   - The procedure `DBDelete` takes a `<Dropbox T>` argument, `DB`, and an atom, `Name`, and removes from the state of `DB` any mapping from `Name` to some value.
   - The function `DBFetch` takes a `<Dropbox T>` argument, `DB`, and an atom, `Name`, and returns the value that `DB` maps `Name` to. Your code may either assume that `DB` has a mapping from `Name` to some value (or you can return a failed value, see page 328 in the textbook [VH04]).
   - The function `DBNames` takes a `<Dropbox T>` argument, `DB`, and returns a list that represents the domain of all the mappings in the state of `DB`.

   Note that `DBDeposit` and `DBDelete` are **proc**s, not **fun**s. If you make them **fun**s instead, your code will not work with the tests!

   Figure 5 on the following page has tests for this type. Note that this requires that your code goes in a file named `Dropbox.oz`.

```
% $Id: DropboxTest.oz,v 1.4 2012/04/12 14:59:55 leavens Exp leavens $
\insert 'Dropbox.oz'
\insert 'TestingNoStop.oz'
declare
% Two set helpers for testing, which you don't have to implement
fun {SubsetOf LS1 LS2}
   {FoldR LS1 fun {$ Atm Res} {Member Atm LS2} andthen Res end true}
end
fun {EqualAsSets LS1 LS2} {SubsetOf LS1 LS2} andthen {SubsetOf LS2 LS1} end

{StartTesting 'DropboxTest $Revision: 1.4 $'}
Books = {NewDropbox}
Languages = {NewDropbox}
{Test {DBNames Books} '==' nil}
{DBDeposit Books secret [studying helps]}
{Test {DBNames Books} '==' [secret]}
{Test {DBNames Languages} '==' nil}
{Test {DBFetch Books secret} '==' [studying helps]}
{DBDeposit Books tale_of_two_cities
 [it was the best 'of' times it was the worst 'of' times]}
{Test {EqualAsSets {DBNames Books} [tale_of_two_cities secret]} '==' true}
{Test {DBFetch Books secret} '==' [studying helps]}
{Test {DBFetch Books tale_of_two_cities}
 '==' [it was the best 'of' times it was the worst 'of' times]}
{DBDelete Books secret}
{DBDeposit Books moby_dick [call me ishmael]}
{DBDeposit Books pride_and_prejudice
 [it is a truth universally acknowledged]}
{Test {EqualAsSets {DBNames Books}
       [moby_dick pride_and_prejudice tale_of_two_cities]} '==' true}
{Test {DBFetch Books tale_of_two_cities}
 '==' [it was the best 'of' times it was the worst 'of' times]}
{Test {DBFetch Books moby_dick} '==' [call me ishmael]}
{Test {DBFetch Books pride_and_prejudice}
 '==' [it is a truth universally acknowledged]}
{DBDeposit Languages french 7#sept}
{DBDeposit Languages spanish 6#seis}
{Test {EqualAsSets {DBNames Languages} [french spanish]} '==' true}
{Test {DBFetch Languages french} '==' 7#sept}
{DBDeposit Languages danish  5#fem}
{DBDeposit Languages catalan 4#cuatro}
{DBDeposit Languages irish 3#tri}
{Test {EqualAsSets {DBNames Languages} [irish catalan danish spanish french]}
 '==' true}
{Test {DBFetch Languages danish} '==' 5#fem}
{Test {DBFetch Languages irish} '==' 3#tri}
{DBDelete Languages danish}
{Test {EqualAsSets {DBNames Languages} [irish catalan spanish french]}
 '==' true}
{Test {DBFetch Languages catalan} '==' 4#cuatro}
{DoneTesting}
```

Figure 5: Tests for problem 6.

7. (0 points) (suggested practice) [UseModels]

   Using Oz's message passing model, write a function `NewFutureServer` that takes no arguments and returns a port object that acts as a "future server". A future server remembers a list of requests for the value of a computation, or a computed result, which is the value of the last computation it was called on to perform.

   The returned port object understands the following messages:

   - The `request(`X`)` message includes an undetermined dataflow variable, X; if the server already has a computed result, then X is unified with the result. Otherwise if the server does not already have a computed result, then it remembers this request, in particular it remembers X, and when the server eventually has a computed result, it unifies X with that result.
   - The `compute(F)` message includes a function `F` that takes no arguments; this function is run and its result defines the "computed result" that the server remembers. The computed result is unified with all dataflow variables from previous unfulfilled requests received by the server.

   Note that when the future server already has a computed result, if it receives a `compute(F)` message, then it uses the result of `{F}` as the new computed result, which is used to answer further requests.

   Hint: you can store undetermined dataflow variables in a list or other data structure, but be very careful not to try to pattern match or perform other computations that would need their values. You might try using different kinds of state records to remember the necessary information in the port object's state.

   Figure 6 on the next page contains various tests.

```
% $Id: NewFutureServerTest.oz,v 1.7 2012/04/12 14:59:55 leavens Exp leavens $
\insert 'NewFutureServer.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'NewFutureServerTest $Revision: 1.7 $'}
declare
FS = {NewFutureServer}
local R1 R2 R3 R4 in
   {StartTesting 'initial requests'}
   {Send FS request(R1)}
   {Send FS request(R2)}
   {Send FS request(R3)}
   {Delay 1000}
   {Send FS compute(fun {$} {Pow 3 24} end)}
   {Wait R1}
   {Test R1 '==' 282429536481}
   {Test R2 '==' 282429536481}
   {Test R3 '==' 282429536481}
   {Send FS request(R4)}
   {Test R4 '==' 282429536481}
   local R5 in
      {Send FS compute(fun {$} {Pow 3 4} end)}
      {Send FS request(R5)}
      {Test R5 '==' 81}
   end
end
{StartTesting 'first part done'}
FS2 = {NewFutureServer}
local R1 R2 in
   {Send FS2 request(R1)}
   {Send FS2 request(R2)}
   {Send FS2 compute(fun {$} nil end)}
   {Wait R2}
   {Test R1 '==' nil}
   {Test R2 '==' nil}
end
{StartTesting 'second part done'}
FS3 = {NewFutureServer}
local R1 R2 in
   {Send FS3 compute(fun {$} 4000+20 end)}
   {Send FS3 request(R1)}
   {Send FS3 request(R2)}
   {Wait R2}
   {Test R1 '==' 4020}
   {Test R2 '==' 4020}
end
{DoneTesting}
```

Figure 6: Testing for Problem 7 on the previous page. Note that this is a suggested practice problem, so you don't have to program it if you don't want the practice!

8. (20 points) [UseModels]

   Using Oz's message passing model, write a function `NewEBay` that takes no arguments and returns a port object that acts as an electronic auction house, like EBay.

   This port object understands several messages.

   The `bid(Amt Info DidIWin)` message places a bid on the item, where `Amt` is a non-negative integer (the number of dollars bid), `Info` is some information about the bidder (e.g., their name), and `DidIWin` is an undetermined store variable.

   The `finish` message ends the auction and does all notification of the bidders. When the `finish` message is received, the `DidIWin` variable in the first `bid` message received whose `Amt` was the largest is bound to `true`, and all other `DidIWin` variables in other `bid` messages are bound to `false`. (Note that there may be no bids. In case of a tie, the first `bid` message received by the port with the highest amount wins. If any `bid` message is received after the `finish` message, it does not win. You can assume that only one `finish` message is ever sent to the port object.)

   The `whoWon(Winner)` message will only be sent after the `finish` message. This message contains an undetermined dataflow variable, `Winner`, which is unified with the `Info` from the winning `bid` message (if any), or the atom none if there were no bids received by the port object.

   Note that, to prevent bidders from gaining information before the auction is over, none of the `DidIWin` variables sent in the `bid`messages may be determined by before the port object receives the `finish` message. Thus the port object will have to remember at least the `DidIWin` variables from all the bids until the `finish` message is received.

   You should use `NewPortObject` in your solution (see the textbook, and the `NewPortObject.oz` file supplied with the test cases for this homework).

   Hint: you may want to use helping functions or procedures.

   Figure 7 on the following page contains some examples.

9. (0 points) (suggested practice) [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, such as a printer or access to a critical section of code. (The exact 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

```
% $Id: NewEBayTest.oz,v 1.4 2012/04/12 14:59:55 leavens Exp leavens $
\insert 'NewEBay.oz'
\insert 'TestingNoStop.oz'
declare
{StartTesting 'NewEBayTest $Revision: 1.4 $'}
MyEBay = {NewEBay}
local Status1 Status2 Status3 Status4 in
   {Send MyEBay bid(8 palin Status1)}
   {Send MyEBay bid(17 mccain Status2)}
   {Send MyEBay bid(27 bush Status3)}
   {Send MyEBay bid(24 romney Status4)}
   % losing bids should not yet be determined
   {Test {IsDet Status1} '==' false}
   {Send MyEBay finish}
   {Test Status1 '==' false}
   {Test Status2 '==' false}
   {Test Status3 '==' true}
   {Test Status4 '==' false}
   {Test {Send MyEBay whoWon($)} '==' bush}
end
MyEBay2 = {NewEBay}
local Status1 Status2 Status3 Status4
   Status5 Status6 Status7
in
   {Send MyEBay2 bid(100 agent007 Status1)}
   {Send MyEBay2 bid(15 agent86 Status2)}
   {Send MyEBay2 bid(99 agent99 Status3)}
   {Send MyEBay2 bid(99 agent992 Status4)}
   {Send MyEBay2 bid(100 agent1002 Status5)}
   {Send MyEBay2 bid(100 agent1003 Status6)}
   {Send MyEBay2 bid(100 agent1004 Status7)}
   {Send MyEBay2 finish}
   {StartTesting 'first of the highest bids wins'}
   {Test Status1 '==' true}
   {Test Status2 '==' false}
   {Test Status3 '==' false}
   {Test Status4 '==' false}
   {Test Status5 '==' false}
   {Test Status6 '==' false}
   {Test Status7 '==' false}
   {Test {Send MyEBay2 whoWon($)} '==' agent007}
end
{StartTesting done}
{StartTesting 'finishing with no bids'}
MyEBay3 = {NewEBay}
{Send MyEBay3 finish}
{Test {Send MyEBay3 whoWon($)} '==' none}
{StartTesting 'bidding after an auction is over loses'}
{Test {Send MyEBay3 bid(50 mad_hatter $)} '==' false}
{Test {Send MyEBay bid(999 alice $)} '==' false}
{DoneTesting}
```

Figure 7: Testing for Problem 8 on the previous page.

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.

Figure 8 has some examples.

```
% $Id: NewResourceArbiterTest.oz,v 1.3 2012/04/12 14:59:55 leavens Exp leavens $
\insert 'NewResourceArbiter.oz'
\insert 'TestingNoStop.oz'
declare
{StartTesting 'NewResourceArbiterTest $Revision: 1.3 $'}
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
{DoneTesting}
```

Figure 8: Testing for Problem 9 on page 12. Note that this is a suggested practice problem, so you don't have to program it unless you want the practice!

Comparisons Among Models

10. (0 points) (suggested practice) [EvaluateModels]

    Suppose you are asked to program a simulation of an agent-based auction system for someone doing research in economics. This system consists of several independent agents, each of which must communicate with a central auction server to evaluate merchandise, place bids, and make payments.

    Among the programming models we studied this semester, what is the most restrictive (i.e., the least expressive or smallest) programming model that can practically be used program the overall structure of such a system? Briefly justify your answer.

11. (6 points) [EvaluateModels]

    Add a row to your table from homework 4 that listed all the different programming techniques. The row you should add is for the message passing model. That is for the message passing model:

    (a) (3 points) What are the characteristics of problems that are best solved with the message passing model's techniques (i.e., when should the message passing model be used)?

    (b) (3 points) Describe (i.e., name) one example for which the message passing technique is particularly well suited.

Points

This homework's total points: 99.

References

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.