Homework 1: Introduction to Programming Concepts

See webcourses and the syllabus for due dates.
In this homework you will learn some of the basics of Oz and the Mozart system [UseModels], and, more importantly, you will get an overview of programming concepts [Concepts].

General Directions

Answers to English questions should be in your own words; don't just quote text from the textbook.
For all Oz programing exercises, you must run your code using the Mozart/Oz system (use the "Feed Buffer" item in the Oz menu to run a file's code). See the course's "Running Oz" page for instructions about installation and troubleshooting of the Mozart/Oz system on your own computer.
For programming problems for which we provide tests, you can find them all in a zip file, which you can download from Webcourses in the attachments to problem 1.
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.
Our tests use the functions in the course library's `TestingNoStop.oz`. The `Test` procedure in this file can be passed an actual value, a connective (which is used only in printing), and an expected value, as in the following statement.

```
{Test {CombA 4 3} '==' 24 div (6*1)}
```

The `Assert` procedure in this file can be passed a boolean, as in the following statement

```
{Assert {Comb J I} == {CombB J I}}
```

Calls to `Assert` produce no output unless they are passed the argument **`false`**. Note that you would not pass to `Browse` or `Show` a call to `Test` or `Assert`, since neither of these procedures returns a value. If you're not sure how to use our testing code, ask us for help.

What to turn in

For problems that require code, you must turn in both: (1) the code file and (2) the output of our tests. Please upload code as text files with the name given in the problem or testing file and with the suffix `.oz`. Please use the name of the main function as the name of the file. Please upload test output and English answers either directly into the answer box in the webcourses assignment, or as plain text files with suffix `.txt`. When you upload files, don't put spaces or tabs in your file names!
Your code should compile with Oz, if it doesn't you should keep working on it. (Email the staff with your code file if you need help getting it to compile.)

Other directions

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 functions from the Oz library (base environment).
Don't hesitate to contact the staff if you are stuck at some point.
For background, you should read Chapter 1 of the textbook [VH04] (except section 1.7). But you may also want to refer to the reference and tutorial material on the Mozart/Oz web site. See also the course resources page.

Reading Problems

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

Read section 1.1 and 1.2 of the textbook [VH04] and answer the following questions.

1. [UseModels]
   (a) (2 points) What does `{Browse funny}` do in Oz?
   (b) (3 points) What kind of character must a variable identifier, start with in Oz?
   (c) (5 points) Can variables in Oz be assigned a value more than once? (Answer "yes" or "no" and give a brief explanation.

Read sections 1.3-1.15 of the textbook and answer the following questions.

2. (5 points) [Concepts] [UseModels] This question deals with lists that represent "3 address instructions." Such lists always have 4 elements and look something like `[add 2 5 7]` where the symbol `add` tells what instruction it is, and the numbers 2, 5, and 7 stand for register numbers or addresses. These lists will always have exactly 4 elements. Using Oz's pattern matching feature, write a function that returns true just when the second and third elements of such a list are equal (i.e., compare equal using ==), and false otherwise. Call this function `First2OpsEqual`. The following are some tests, found in the file First2OpsEqualTest.oz; you should run these tests on your code and hand in the output along with your code (as described above).

   To run our tests, put your file `First2OpsEqual.oz` and our test file `First2OpsEqualTest.oz` in the same directory. Then run our tests by feeding the buffer `First2OpsEqualTest.oz` to Oz. You will have to look at the `*Oz Emulator*` buffer to see the output. Upload to webcourses your `First2OpsEqual.oz` file and paste the contents of the `*Oz Emulator*` buffer into the answer box in Webcourses. (See also the general directions at the beginning of this homework.)

```
% $Id: First2OpsEqualTest.oz,v 1.1 2010/08/12 00:40:17 leavens Exp leavens $
\insert 'First2OpsEqual.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'First2OpsEqualTest $Revision: 1.1 $'}
% If you fix your code, then you may have to restart Oz to make these pass...
{Test {First2OpsEqual add|2|7|5|nil} '==' false}
{Test {First2OpsEqual add|3|3|5|nil} '==' true}
{Test {First2OpsEqual mult|3|3|8|nil} '==' true}
{Test {First2OpsEqual divide|0|1|8|nil} '==' false}
{Test {First2OpsEqual sub|1|1|8|nil} '==' true}
{StartTesting 'done'}
```

   Hint: Note that you can program this with a single pattern match in a single **`case`** expression. Note: Be sure to put your code in a file named `First2OpsEqual.oz`, otherwise our testing code won't find your solution. Our tests only call `First2OpsEqual` on lists that are 4 elements long, so you should assume that the inputs will have that type.

   You are prohibited from using the numeric field deference operators, such as `.1` and `.2`, in your solution.

   See the course examples page for many examples of Oz functions.

3. (5 points) [Concepts] What happens when the following code executes in Oz? Briefly explain why that happens.

```
local SetIt It in
   It = good
   SetIt = proc {$ X}
              It = X
           end
   {SetIt bad}
   {Browse 'It is '#It}
end
```

4. (5 points) [Concepts] What happens when the following code executes in Oz? Briefly explain why that happens.

```
local X in
   X = X+1
   {Browse 'X is '#X}
end
```

Read sections 1.3-1.15 of the textbook and answer the following questions.

5. (5 points) [Concepts] According to chapter 1, what problems does nondeterminism cause in concurrent programming?

Regular Problems

We expect you'll do the problems in this section after reading the entire chapter. However, you can probably do some of them after reading only part of the chapter.

The textbook problems are from the *Concepts, Techniques and Models of Computer Programming* book [VH04, section 1.18].

6. (20 points) [UseModels]

   Do problem 2 in chapter 1, calculating combinations. Note that this should be done without using cells or assignment (that is, in the declarative model).

   Your solution's Oz code should be in a file `Comb.oz`, and that file should contain two functions. Part (a)'s solution should be called `CombA`, and part (b)'s solution called `CombB`.

   Hint: use the function `Comb` from section 1.3, and use recursion. Don't write the same code twice, instead make function calls. In Oz, use **`div`** to divide two integers; for example, `12` **`div`** `4` returns 3.

   You must test your code using Mozart/Oz. After doing your own tests (with `Show` or `Browse`) you must run our tests. To do this, put your file `Comb.oz` and our test file `CombTest.oz` in the same directory. Then run our tests by feeding the buffer `CombTest.oz` to Oz. You will have to look at the `*Oz Emulator*` buffer to see the output. Then upload your `Comb.oz` file to webcourses and paste the contents of the `*Oz Emulator*` buffer into the answer box in Webcourses. (See also the general directions at the beginning of this homework.)

   If you have trouble running our tests, see the troubleshooting section of the course's running Oz page. If that doesn't help, contact the course staff.

   See the course examples page for many examples of Oz functions.

7. (10 points) [UseModels]

   Do problem 5 in chapter 1, lazy evaluation.

8. (10 points) [UseModels]

   Do Problem 7 in chapter 1, explicit state.

9. (15 points) [UseModels]

   Do problem 10 in chapter 1, explicit state and concurrency.

10. [Concepts]

    Consider the code in Figure 1 on the following page, which is also included in the files available for download with this homework from Webcourses (see the attachments to problem 1).

    (a) (5 points) What symbol is shown (in the emulator on the line above the dot that shows that the run is complete) when you feed `ReserveSeat.oz` to Oz? Do you get the same output each time?

    (b) (5 points) What is shown in the emulator when you comment out the indicated line containing `{Delay {OS.random}}`?

    (c) (5 points) Is the implementation of Oz permitted to introduce delays where the statement `{Delay {OS.rand}}` appears in Figure 1 on the next page?

    (d) (5 points) Suppose the threads in Figure 1 on the following page were created at unpredictable times by HTTP requests. Would you recommend the way Figure 1 on the next page is coded (with the `{Delay {OS.rand}}` commented out) as a reliable way to achieve the effect of giving the seat to the first visitor to the associated website? Answer "yes" or "no" and give a brief reason.

```
% $Id: ReserveSeat.oz,v 1.3 2010/08/29 22:51:56 leavens Exp leavens $
declare
TheSeat = {NewCell nobody}
proc {ReserveSeat Who}
   if @TheSeat == nobody
   then
      {Delay {OS.rand}} % comment this out for part (b), discussed in part (c)
      TheSeat := Who
   end
end

local Done A B C D E F G in
   thread {ReserveSeat amy} Done=A end
   thread {ReserveSeat bob} A=B end
   thread {ReserveSeat carli} B=C end
   thread {ReserveSeat dan} C=D end
   thread {ReserveSeat eloise} D=E end
   thread {ReserveSeat fred} E=F end
   thread {ReserveSeat gayle} F=G end
   G=unit

   {Wait Done}
   {System.showInfo @TheSeat}
   {System.showInfo "."}
end
```

Figure 1: Oz code in the file `ReserveSeat.oz`, which is included in the hw1-test.zip file.

Points

This homework's total points: 105.

References

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