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]. We will also motivate careful comparisons with other langauges [MapToLanguages].

General Directions

Answers to English questions should be in your own words; don't just quote 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 various material, including the textbook, ideally in advance of class meetings.

Read the essay "Beating the Averages" by Paul Graham (from the book *Hackers and Painters* [Gra04]). The essay can be found on the web at http://www.paulgraham.com/avg.html.

1. [MapToLanguages] For this problem, please put your answer in the answer box on webcourses, and clearly identify each part of the answer.
   (a) (5 points) What is the main claim that Graham makes in this essay?
   (b) (5 points) Can you state the essay's main claim in a way that is scientifically testable? If so, do that; otherwise say why it cannot be stated in a way that is testable.
   If you do not think the claim can be made to be scientifically testable, then state an alternative hypothesis that is as similar as you can make it, but which is scientifically testable.
   (c) (5 points) What experiment could we do during this class to provide evidence for or against the scientifically testable claim you proposed in the previous part? (Give a brief description.)

Read section 1.1 and 1.2 of the textbook [VH04] and answer the following questions. The **`local`** expressions that are used in stating the problem allow local declarations within expressions. They are explained in more detail on page 63.

2. [Concepts] [UseModels] For this problem, please put your answers in the answer box on webcourses, and clearly indicate which part of your answer goes with each part of the question.
   (a) (2 points) What does `{Browse local UCF = cool in UCF end}` do in Oz?
   (b) (3 points) Are `Browse` and `UCF` variable identifiers or symbols (atoms) in Oz?
   (c) (3 points) What happens when you execute `{Browse local UF = good in UF = bad end}` in Oz?
   (d) (2 points) Explain why what happens in the previous part happens.

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

3. (15 points) [Concepts] [UseModels] This question deals with 3 element lists of atoms, that represent simple sentences, hence we will write the type of such lists as `<Sentence>` in what follows. An example `<Sentence>` is `[robby ate lunch]`, where the first element of the list is considered the subject, the second the verb, and the third the object.

   Using Oz's pattern matching feature, write a three functions:

   ```
   Subject : <fun {$ <Sentence>}: <Atom>>
   Verb : <fun {$ <Sentence>}: <Atom>>
   Object : <fun {$ <Sentence>}: <Atom>>
   ```

   such that `Subject` returns the first element (an atom) of a `<Sentence>` passed as its argument, `Verb` returns the second element (also an atom), and `Object` returns the third element.

   You are prohibited from using recursion or the numeric field dereference operators, such as `.1` and `.2`, in your solution. You must not call the Oz library function `Nth` either. Instead, use **`case`** and pattern matching. Note that you can use nested patterns or nested **`case`** expressions.

   Put your code in a file named `SentenceOps.oz`, in the same directory with our testing file `SentenceOpsTest.oz`. Then run our tests by feeding the buffer `SentenceOpsTest.oz` to Oz. You will have to look at the `*Oz Emulator*` buffer to see the output. Upload to webcourses your `SentenceOps.oz` file. Then copy the contents of the `*Oz Emulator*` buffer and paste them into the answer box in Webcourses. (See also the general directions at the beginning of this homework.)

   The file SentenceOpsTest.oz contains the following tests. you should run these tests on your code and hand in the output along with your code (as described above).

```
% $Id: SentenceOpsTest.oz,v 1.1 2011/08/25 15:19:56 leavens Exp leavens $
\insert 'SentenceOps.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'SentenceOpsTest $Revision: 1.1 $'}
% If you fix your code, then you may have to restart Oz to make these pass...
{Test {Subject penny|ate|borscht|nil} '==' penny}
{Test {Subject monsters|killed|john|nil} '==' monsters}
{Test {Subject rex|eats|alpo|nil} '==' rex}

{Test {Verb rex|eats|alpo|nil} '==' eats}
{Test {Verb sarah|accuses|media|nil} '==' accuses}
{Test {Verb joe|enjoys|dancing|nil} '==' enjoys}

{Test {Object joe|enjoys|dancing|nil} '==' dancing}
{Test {Object cindy|sings|rock|nil} '==' rock}
{Test {Object bacteria|kill|humans|nil} '==' humans}
{StartTesting 'done'}
```

Note that our tests only call these functions on <Sentence> arguments, so you can assume that the argument will be a list of exactly 3 atoms. Thus you don't have to write any code for cases where the input is not a <Sentence>.

Hint: Note that you can program each function with a single pattern match in a single **case** expression.

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

4. (5 points) [Concepts] What happens when the following code executes in Oz? Briefly explain why that happens. (Note that the notation 'X is '#X makes a pair, which is special kind of tuple, i.e., a record with label # and numbered fields. See page 52.)

```
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?

   Please put your answer into the answer box on webcourses.

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. (15 points) [UseModels]

   In Oz, write a function

   Log2: <**fun** {$ Int}: Int>

that takes an integer N that is strictly greater than 0, and which returns the base 2 logarithm of N, rounded down to the nearest whole integer. That is, if ℓ is the answer returned, then $2^\ell \leq$ N and N $< 2^{\ell+1}$. You are prohibited from using the Oz Log or Pow functions in your solution; also, do not do conversions from integers to floating point or vice versa. Instead use recursion, and Oz's built-in **div** operator (which does integer division) or Oz's built-in **mod** operator, and the built-in functions IsOdd or IsEven.

Hint: It's a good idea to use a helping function, so that you have an extra variable to use during the recursions. See section 3.2.3 of the textbook, or the chapter 3 examples in the course's Code Examples Page. Thus your code might look, in outline, like:

```
declare
fun {Log2 N}
   % assume N > 0
   {Log2Helper N _____} % the blank is the initial value of Acc
end

fun {Log2Helper N Acc}
   % assume N >= 0
   ____________________  % your code goes here, perhaps on several lines...
end
```

Your solution's Oz code should be in a file Log2.oz.

You must test your code using Mozart/Oz. After doing your own tests (with Show or Browse) you must run our tests, shown in Figure 1 on the following page. To do this, put your file Log2.oz and our test file Log2Test.oz in the same directory. Then run our tests by feeding the buffer Log2Test.oz to Oz. You will have to look at the *Oz Emulator* buffer to see the output. Then upload your Log2.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.

   Please put your answer in the answer box on webcourses.

8. (15 points) [UseModels]

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

9. [Concepts]

   Consider the code for ReserveSeat in Figure 2 on page 6, which is also included in the files available for download with this homework from Webcourses (see the attachments to problem 1).

   For this problem, please put your answers in the answer box on webcourses, and clearly indicate which part of your answer goes with each part of the question.

   (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; that is if you comment out the line containing {Delay {OS.random} **mod** 30}?

   (c) (5 points) Is the implementation of Oz permitted to introduce delays where the statement {Delay {OS.rand}} appears in Figure 2 on page 6?

```
% $Id: Log2Test.oz,v 1.1 2011/08/25 14:53:10 leavens Exp $
\insert 'Log2.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'Log2Test $Revision: 1.1 $'}
{Test {Log2 64} '==' 6}
{Test {Log2 1} '==' 0}
{Test {Log2 2} '==' 1}
{Test {Log2 3} '==' 1}
{Test {Log2 4} '==' 2}
{Test {Log2 5} '==' 2}
{Test {Log2 7} '==' 2}
{Test {Log2 8} '==' 3}
{Test {Log2 9} '==' 3}
{Test {Log2 12} '==' 3}
{Test {Log2 15} '==' 3}
{Test {Log2 16} '==' 4}
{Test {Log2 17} '==' 4}
{Test {Log2 31} '==' 4}
{Test {Log2 32} '==' 5}
{Test {Log2 63} '==' 5}
{Test {Log2 256} '==' 8}
{Test {Log2 128} '==' 7}
{Test {Log2 129} '==' 7}
{Test {Log2 255} '==' 7}
{Test {Log2 257} '==' 8}
{Test {Log2 511} '==' 8}
{Test {Log2 512} '==' 9}
{Test {Log2 1023} '==' 9}
{Test {Log2 1024} '==' 10}
{Test {Log2 1025} '==' 10}
{Test {Log2 {Pow 2 24}} '==' 24}
{Test {Log2 {Pow 2 57}} '==' 57}
{Test {Log2 {Pow 2 57}-1} '==' 56}
{StartTesting 'done'}
```

Figure 1: Testing code for problem 6.

.

```
% $Id: ReserveSeat.oz,v 1.4 2011/08/25 15:11:29 leavens Exp $
declare
TheSeat = {NewCell nobody}
proc {ReserveSeat Who}
   if @TheSeat == nobody
   then
      {Delay {OS.rand} mod 30} % comment out for part (b), discussed in part (c)
      TheSeat := Who
   end
end

local Done A B C D E F G in
   thread {ReserveSeat ann} Done=A end
   thread {ReserveSeat bill} A=B end
   thread {ReserveSeat carla} B=C end
   thread {ReserveSeat don} C=D end
   thread {ReserveSeat ellen} D=E end
   thread {ReserveSeat frank} E=F end
   thread {ReserveSeat gina} F=G end
   G=unit

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

Figure 2: Oz code in the file `ReserveSeat.oz`, which is included in the hw1-test.zip file.

(d) (5 points) Suppose the threads in Figure 2 were created at unpredictable times by HTTP requests. Would you recommend the way Figure 2 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.

Points

This homework's total points: 110.

References

[Gra04] Paul Graham. *Hackers and Painters – Big Ideas from the Computer Age*. O'Reilly, 2004.

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