Homework 4: Declarative Concurrency

See Webcourses and the syllabus for due dates.

In this homework you will learn about the declarative concurrent model and basic techniques of declarative concurrent programming. The programming techniques include stream programming and lazy functional programming [Concepts] [UseModels]. A few problems also make comparisons with the declarative model and with concurrency features in Java [MapToLanguages]. A few problems also make comparisons between programming techniques [EvaluateModels]; you may want to look at Problem 27 on page 13 right now so you can be thinking about it while you do the other problems.

Your code should be written in the declarative concurrent model, so you must not use cells and assignment in your Oz solutions. (Furthermore, note that the declarative 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.) But please use all linguistic abstractions and syntactic sugars that are helpful.

For all Oz programing 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 the course resources web page or 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.

Turn in (on Webcourses) your code and output of your testing for all exercises that require code. 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 as plain text files with suffix `.txt` or as PDF files with suffix `.pdf`. If you have a mix of code and English, use a text file with a `.oz` file suffix, and put comments in the file for the English parts. (In any case, don't put any spaces in your file names!)

Your code should compile with Oz, if it doesn't you probably should keep working on it. If you don't have time, at least tell us that you didn't get it to compile.

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 that are compatible with the declarative model from the Oz library (base environment), especially functions like `Map` and `FoldR`.

Your code should compile with Oz, if it doesn't you probably should keep working on it. If you don't have time, at least tell us that you didn't get it to compile.

You might lose points even if your code is passing all the tests, if you don't follow the grammar or if your code is confusing or needlessly inefficient. Make sure you don't have extra case clauses or unnecessary if tests, make sure you are using as much function definitions/calls as needed, check if you are making unnecessary loops or passes in your code, efficient code means full mark, working code does not necessarily mean full marks.

Don't hesitate to contact the staff if you are stuck at some point.

Read Chapter 4 of the textbook [RH04]. (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 4, through section 4.1 of the textbook [RH04] and answer the following questions.

1. (5 points) [Concepts] [MapToLanguages]

   The declarative concurrent model of chapter 4 adds threads to Oz. Threads are known to cause difficulties for reasoning about programs, because multiple threads can interleave their execution in many different orders. In a language like Java, C, or C++, the program can observed these different execution orders, resulting in programs that give different outputs, and are thus observably nondeterministic. What's different about the declarative concurrent model that makes it observably deterministic?

2. [Concepts]

   Consider the following Oz code.

```
declare
fun {By10 X}
   local P in
      thread P = X*10 end
      P
   end
end
```

Suppose we execute the expression `{By10 2}+4000`.

(a) (3 points) Describe two different interleavings that are possible in the execution of this expression.

(b) (2 points) What possible values can we observe for this expression?

Read chapter 4, through section 4.2 of the textbook [RH04] and answer the following questions.

3. [Concepts] [UseModels]

   (a) (2 points) What example in section 4.2 demonstrates the use of dataflow behavior to give incremental output?

   (b) (3 points) What part of this example's code makes it possible for the output to be incrementally produced?

Read chapter 4, through section 4.3 of the textbook [RH04] and answer the following questions.

4. (5 points) [Concepts] [MapToLanguages]

   In addition to the types called Steams in Java and C#, which are commonly used for input/output, what other kinds of objects in these languages behave like streams? In other words, what kind of object has operations that allows one to ask for the next element in a sequence of elements, and to work with a potentially unbounded sequence without the computation going into an infinite loop if only a finite amount of the sequence is needed? Briefly explain.

5. (5 points) [UseModels]

   Describe one way to program a producer-consumer architecture in Oz.

6. (5 points) [UseModels]

   Briefly describe one technique for doing flow control in a stream-based system.

Read chapter 4, through section 4.4 of the textbook [RH04] and answer the following questions.

7. (5 points) [Concepts]

   What operations in Oz determine when the value of a by-need suspension is "needed"?

Read chapter 4, through section 4.5 of the textbook [RH04] and answer the following questions.

8. (5 points) [Concepts] [MapToLanguages]

   Would it be a good idea in Java (or C++, or C) if all functions were lazy by default? Briefly explain.

9. (5 points) [Concepts] [UseModels]

   In what sense is using a bounded buffer a compromise between the data-driven and demand-driven models?

Read chapter 4, section 4.8 of the textbook [RH04] and answer the following questions.

10. (5 points) [EvaluateModels]

    Is it possible for a server, such as a retail web site, to adequately serve multiple independent clients if it is programmed solely using the declarative concurrent model? Briefly explain.

Read chapter 4, section 4.9.2 of the textbook [RH04] and answer the following questions.

11. (5 points) [Concepts] [EvaluateModels]

    Why is it useful to support nonstrict evaluation on a multi-core (or parallel) processor?

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 following problems are from the textbook [RH04, section 4.11].

Thread and Dataflow Behavior Semantics

The following problems explore the semantics of the declarative concurrent model.

12. (10 points) [Concepts]

    Do problem 4 in section 4.11 of the textbook [RH04] (Order-determining concurrency).

13. This problem explores threads and dataflow in Java [MapToLanguages]

    For more information about threads in Java see, for example, *The Java Tutorial* [CW98], which is online at `http://java.sun.com/docs/books/tutorial/`. The concurrency section of that tutorial has several examples.

    (a) (10 points) Write, in Java, write a version of the program in the textbook's problem 4 (of section 4.11) in which the Oz program's dataflow variables `A`, `B`, `C`, and `D` are represented by fields (perhaps static fields) in the Java code. Call your Java class `FourThreads`. This version the Java code does not need to faithfully match the semantics of Oz exactly, just translate to Java in the most straightforward way possible.

    Hints: You will need to either use package-visible fields or nest your subclasses of `Thread` or subtypes of `Runnable` within your main `FourThreads` class. You can use anonymous inner classes to make your code shorter.

    (b) (5 points) How much extra code is present in the Java version compared with the Oz code?

    (c) (5 points) Does your Java code have the possibility of race conditions? Briefly explain.

    (d) (5 points) Does your Java code always give the same answer as the Oz Code? Briefly explain.

    (e) (40 points) Write, in Java, a class `IntDataflowVariable`, with operations `int get()` and `void set(int)`. The `get` operation should suspend the calling thread until the `set` method is called. The `set` method should throw an `IllegalArgumentException` if it is called again with a different argument than the value used the first time it was called. Test by using it to write another version of the textbook's problem 4 (of section 4.11) using instances of your class `IntDataflowVariable` in place of the integer fields. Call this class `FourThreadsWithDataflow`. Running this class's main method should always give the same output as the Oz code.

14. (10 points) [Concepts] [UseModels]

    Do problem 5 in section 4.11 of the textbook [RH04] (The Wait Operation).

15. (20 points) [Concepts]

    Do problem 8 in section 4.11 of the textbook [RH04] (Dataflow behavior in a concurrent setting).

Streams and Lazy Functional Programming

The following problems, many of which are due to John Hughes, relate to modularization of numeric code using streams and lazy execution. In particular, we will explore the Newton-Raphson algorithm. This algorithm computes better and better approximations to the square root of a floating point number `N` from a previous approximation `X` by using the following curried function.

```
% $Id: Next.oz,v 1.2 2007/10/31 00:42:36 leavens Exp leavens $
declare
% Next: <fun {$ <Float>}: <fun {$ <Float>}: Float>>
fun {Next N}
   fun {$ X}
      (X + N / X) / 2.0
   end
end
```

In the following the type `<IStream T>` means infinite lists of type T. Note that `nil` never occurs in a `<IStream T>`, which has the following grammar:

⟨IStream T⟩ ::= `T '|'` ⟨IStream T⟩

As an aid to writing code for this section, and for testing that code, we provide a library file containing predicates for approximate comparisons of floating point numbers and for testing with approximate comparisons. The floating point approximate comparison code is shown in Figure 1 on the next page. The testing code for floating point numbers is shown in Figure 2 on page 6.

16.

(a) (10 points) [UseModels] Write a lazy function

`IStreamIterate: <fun lazy {$ <fun {$ T}: T> T}: <IStream T>>`

such that `{IStreamIterate` F X`}` takes a function F and a value X and returns the stream

$$X \mid \{FX\} \mid \{F\{FX\}\} \mid \{F\{F\{FX\}\}\} \mid \ldots,$$

that is, the stream whose i^{th} item, counting from 1, is F^{i-1} applied to X.

The examples in Figure 3 on page 7 are written using the `WithinTest` procedure from Figure 2 on page 6. Notice also that, since `Next` is curried, we don't pass `Next` itself to `IStreamIterate`, but instead pass the value of applying `Next` to some number.

(b) (5 points) [Concepts]

Why does `IStreamIterate` have to be declared to be lazy? Give a brief answer.

```
% $Id: FloatPredicates.oz,v 1.3 2007/10/23 02:14:21 leavens Exp leavens $
%% Some functions to do approximate equality of floating point numbers.
%% AUTHOR: Gary T. Leavens

declare
%% Return true iff the difference between X and Y
%% is no larger than Epsilon
fun {Within Epsilon X Y} {Abs X-Y} =< Epsilon end

%% Partly curried version of Within
fun {WithinMaker Epsilon} fun {$ X Y} {Within Epsilon X Y} end end

%% Return true iff the corresponding lists are
%% equal relative to the given predicate
fun {CompareLists Pred Xs Ys}
   case Xs#Ys of
      nil#nil then true
   [] (X|Xr)#(Y|Yr) then {Pred X Y} andthen {CompareLists Pred Xr Yr}
   else false
   end
end

%% Return true iff the lists are equal
%% in the sense that the corresponging elements
%% are equal to within Epsilon
fun {WithinLists Epsilon Xs Ys}
   {CompareLists {WithinMaker Epsilon} Xs Ys}
end

%% Return true iff the ratio of X-Y to Y is within Epsilon
fun {Relative Epsilon X Y} {Abs X-Y} =< Epsilon*{Abs Y} end

%% Partly curried version of Relative
fun {RelativeMaker Epsilon} fun {$ X Y} {Relative Epsilon X Y} end end

%% Return true iff the lists are equal
%% in the sense that the corresponging elements
%% are relatively equal to within Epsilon
fun {RelativeLists Epsilon Xs Ys}
   {CompareLists {RelativeMaker Epsilon} Xs Ys}
end

%% A useful tolerance for testing
StandardTolerance = 1.0e~3

%% A convenience for testing, relative equality with a fixed Epsilon
ApproxEqual = {RelativeMaker StandardTolerance}
```

Figure 1: Comparisons for floating point numbers. This code is available in the course `lib` directory.

```
% $Id: FloatTesting.oz,v 1.5 2008/03/24 15:43:04 leavens Exp leavens $
% Testing for floating point numbers.
% AUTHOR: Gary T. Leavens

\insert 'FloatPredicates.oz'
\insert 'TestingNoStop.oz'

declare
%% TestMaker returns a procedure P such that {P Actual '=' Expected}
%% is true if {FloatCompare Epsilon Actual Expected} (for Floats)
%% or if {FloatListCompare Epsilon Actual Expected} (for lists of Floats)
%% If so, print a message, otherwise throw an exception.
fun {TestMaker FloatCompare FloatListCompare Epsilon}
   fun {Compare Actual Expected}
      if {IsFloat Actual} andthen {IsFloat Expected}
      then {FloatCompare Epsilon Actual Expected}
      elseif {IsList Actual} andthen {IsList Expected}
      then {FloatListCompare Epsilon Actual Expected}
      else false
      end
   end
in
   proc {$ Actual Connective Expected}
      if {Compare Actual Expected}
      then {System.showInfo
            {Value.toVirtualString Actual 5 20}
            # ' ' # Connective # ' '
            # {Value.toVirtualString Expected 5 20}}
      else {System.showInfo
            'TEST FAILURE: '
            # {Value.toVirtualString Actual 5 20}
            # ' ' # Connective # ' '
            # {Value.toVirtualString Expected 5 20}
           }
      end
   end
end

WithinTest = {TestMaker Within WithinLists StandardTolerance}
RelativeTest = {TestMaker Relative RelativeLists StandardTolerance}
```

Figure 2: Testing code for floating point. This puts output on standard output (the `*Oz Emulator*` window). The file `FloatPredicates` is shown in Figure 1 on the preceding page. This file is available in the course `lib` directory. To use it, copy the files from the course directory to your own directory and then put `\insert 'FloatTesting.oz'` in your file.

```
% $Id: IStreamIterateTest.oz,v 1.1 2009/02/28 20:52:54 leavens Exp leavens $
\insert 'IStreamIterate.oz'
\insert 'Next.oz'
\insert 'FloatTesting.oz'
{StartTesting 'IStreamIterate...'}
{WithinTest {List.take {IStreamIterate {Next 1.0} 1.0} 7}
 '~=~' [1.0 1.0 1.0 1.0 1.0 1.0 1.0]}
{WithinTest {List.take {IStreamIterate {Next 9.0} 1.0} 7}
 '~=~' [1.0 5.0 3.4 3.0235 3.0001 3.0 3.0]}
{WithinTest {List.take {IStreamIterate {Next 200.0} 1.0} 7}
 '~=~' [1.0 100.5 51.245 27.574 17.414 14.449 14.145]}
{RelativeTest {List.take {IStreamIterate {Next 0.144} 7.0} 9}
 '~=~' [7.0 3.5103 1.7757 0.92838 0.54174 0.40378 0.3802 0.37947 0.37947]}
{RelativeTest {List.take {IStreamIterate fun {$ X} X*X end 2.0} 9}
 '~=~' [2.0 4.0 16.0 256.0 65536.0 4.295e009 1.8447e019 3.4028e038
        1.1579e077]}
{RelativeTest {List.take {IStreamIterate fun {$ X} X/3.0 end 10.0} 8}
 '~=~' [10.0 3.3333 1.1111 0.37037 0.12346 0.041152 0.013717 0.0045725]}
{StartTesting 'done'}
```

Figure 3: Tests for Problem 16 on page 4.

17.

(a) (10 points) [UseModels]
Write a function

```
Approximations: <fun {$ Float Float}: <IStream Float>>
```

such that `{Approximations N A0}` returns the infinite list of approximations to the square root of `N`, starting with `A0`, according to the Newton-Raphson method, that is by iterating `{Next N}`. For example, the call `{Approximations 2.0 1.0}` should compute the infinite stream:

```
1.0 | {{Next 2.0} 1.0}
    | {{Next 2.0} {{Next 2.0} 1.0}}
    | {{Next 2.0} {{Next 2.0} {{Next 2.0} 1.0}}}
    | {{Next 2.0} {{Next 2.0} {{Next 2.0} {{Next 2.0} 1.0}}}}
    | ...
```

More examples appear in Figure 4 on the following page.

(b) (5 points) [Concepts]
Does your function `Approximations` have to be declared to be lazy? Briefly explain.

18. (10 points) [UseModels]

Write a function

```
ConvergesTo: <fun {$ <IStream T> <fun {$ T T}: Bool>}: T>
```

such that `{ConvergesTo Xs Pred}` looks down the stream `Xs` to find the first two consecutive elements of `Xs` that satisfy `Pred`, and it returns the second of these consecutive elements. (It will never return if there is no such pair of consecutive elements.) Figure 5 on the following page gives some examples.

19. (10 points) [UseModels]

Using `WithinMaker` from Figure 1 on page 5 and other functions given above, in particular `ConvergesTo`, write a function

```
SquareRoot: <fun {$ Float Float Float}: Float>
```

such that `{SquareRoot A0 Epsilon N}` returns an approximation to the square root of `N` that is within `Epsilon`. The parameter `A0` is used as an initial guess in the Newton-Raphson method. Examples are given in Figure 6 on the next page.

20. (10 points) [UseModels]

Using `RelativeMaker` from Figure 1 on page 5 and the other pieces given above, in particular `ConvergesTo`, write a function

```
RelativeSquareRoot: <fun {$ Float Float Float}: Float>
```

such that `{RelativeSquareRoot A0 Epsilon N}` returns an approximation to the square root of `N` that only has a relative error of `Epsilon`. The parameter `A0` is used as an initial guess in the Newton-Raphson method. When iterating, keep going until the ratio of the difference between the last and the previous approximation to the last approximation approaches 0, instead of waiting for the differences between the approximations themselves to approach zero. (This is equivalent to iterating until the ratio of the last two approximations approaches 1.) This is test for convergence is better for square roots of very large numbers, and for square roots of very small numbers. Examples are given in Figure 7 on page 10.

```
% $Id: ApproximationsTest.oz,v 1.1 2009/02/28 20:52:54 leavens Exp leavens $
\insert 'Approximations.oz'
\insert 'FloatTesting.oz'
{StartTesting 'Approximations'}
{WithinTest {List.take {Approximations 1.0 1.0} 7}
 '~=~' [1.0 1.0 1.0 1.0 1.0 1.0 1.0]}
{RelativeTest {List.take {Approximations 2.0 1.0} 5}
 '~=~' [1.0 1.5 1.41667 1.41422 1.41421]}
{RelativeTest {List.take {Approximations 64.0 1.0} 5}
 '~=~' [1.0 32.5 17.2346 10.474 8.29219]}
{StartTesting 'done'}
```

Figure 4: Tests for Problem 17 on the previous page.

```
% $Id: ConvergesToTest.oz,v 1.1 2009/02/28 20:52:54 leavens Exp leavens $
\insert 'ConvergesTo.oz'
\insert 'FloatTesting.oz'

fun lazy {Repeat X} X|{Repeat X} end

{StartTesting 'ConvergesTo'}
{WithinTest {ConvergesTo
             {Append [1.0 3.5 4.5] {Repeat 7.0}}
             {WithinMaker 1.01}
            }
 '~=~' 4.5}
{WithinTest {ConvergesTo
             {Append [1.0 32.5 17.2346 10.474 8.29219 8.00515] {Repeat 8.0}}
             {WithinMaker 0.5}
            }
 '~=~' 8.00515}
{StartTesting 'done'}
```

Figure 5: Tests for Problem 18 on the previous page.

```
% $Id: SquareRootTest.oz,v 1.2 2009/02/28 20:52:54 leavens Exp leavens $
\insert 'FloatTesting.oz'
\insert 'SquareRoot.oz'
{StartTesting 'SquareRoot'}
Millionth = 0.0000001
WithinMillionth = {TestMaker
                   Within
                   WithinLists
                   Millionth}
{WithinMillionth {SquareRoot 1.0 Millionth 2.0} '~=~' 1.41421356}
{WithinMillionth {SquareRoot 1.0 Millionth 4.0} '~=~' 2.0}
{WithinMillionth {SquareRoot 1.0 Millionth 64.0} '~=~' 8.0}
{WithinMillionth {SquareRoot 1.0 Millionth 3.14159} '~=~' 1.7724531}
{StartTesting 'done'}
```

Figure 6: Tests for Problem 19 on the preceding page.

```
% $Id: RelativeSquareRootTest.oz,v 1.2 2009/02/28 20:52:54 leavens Exp leavens $
\insert 'FloatTesting.oz'
\insert 'RelativeSquareRoot.oz'
{StartTesting 'RelativeSquareRoot'}
TenThousandth = 0.000001
RelativeTenThousandth = {TestMaker
                  Relative
                  RelativeLists
                  TenThousandth}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 2.0}
                       '~=~' 1.41421356}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 4.0}
                        '~=~' 2.0}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 64.0}
                       '~=~' 8.0}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 3.14159}
                       '~=~' 1.7724531}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 9.0e24}
                       '~=~' 3.0e12}
{RelativeTenThousandth {RelativeSquareRoot 1.0 TenThousandth 9.0e~40}
                       '~=~' 3.0e~20}
{StartTesting 'done'}
```

Figure 7: Tests for Problem 20 on page 8.

21. (15 points) [UseModels]

    You may recall that the derivative of a function `F` at a point `X` can be approximated by the following function.

```
% $Id: EasyDiff.oz,v 1.3 2007/10/31 00:44:27 leavens Exp leavens $
declare
fun {EasyDiff F X Delta} ({F (X+Delta)} - {F X}) / Delta end
```

    Good approximations are given by small values of `Delta`, but if `Delta` is too small, then rounding errors may swamp the result. One way to choose `Delta` is to compute a sequence of approximations, starting with a reasonably large one. (In this way, if `{WithinMaker Epsilon}` is used to select the first approximation that is accurate enough, this can reduce the risk of a rounding error affecting the result, as we will show in the next problem.)

    In this problem your task is to write a function

```
DiffApproxims: <fun {$ Float <fun {$ Float}: Float> Float}:
                    <IStream Float>>
```

    such that `{DiffApproxims Delta0 F X}` returns an infinite list of approximations to the derivative of `F` at `X`, where at each step, the current `Delta` is halved. Examples are given in Figure 8 on the following page.

    Hint: do you need to make something lazy to make this work?

22. (15 points) [UseModels]

    Using the pieces given above, in particular `ConvergesTo` and `DiffApproxims`, write a function

```
Differentiate: <fun {$ Float Float  <fun {$ Float}: Float> Float}:
                    Float>
```

    such that `{Differentiate Epsilon Delta0 F N}` returns an approximation that is accurate to within `Epsilon` to the derivative of `F` at `N`. Use the previous problem (Problem 21) with `Delta0` as the initial value for `Delta`. Examples are given in Figure 9 on the next page.

Laziness Problems

The following problems explore more about laziness and its utility.

23. (10 points) [EvaluateModels]

    Do problem 12 in section 4.11 of the textbook [RH04] (Laziness and incrementality).

24. (10 points) [EvaluateModels]

    Do problem 13 in section 4.11 of the textbook [RH04] (Laziness and monolithic functions).

25. (20 points) [Concepts] [UseModels]

    Do problem 16 in section 4.11 of the textbook [RH04] (By-need execution).

    To explain this problem and provide a test, call your procedure that solves this problem `RequestCalc`. Note that it should be a procedure, not a function. Then when run as in Figure 10 on the next page it should show in the Browser window first `Z`, then it should show `requestin`, then the `Z` should change to `0|_`, then you should see `request2in`, and then the first line should change to `0|1|_`.

    Hints: think about what actions in Oz request (demand) calculation of a variable identifier's value. And think about what new features we have in this chapter that can be used to prevent a thread from waiting for something.

26. (15 points) [Concepts] [EvaluateModels]

    Do problem 19 in section 4.11 of the textbook [RH04] (Limitations of Declarative Concurrency). (Note: there is a typo in the book, the `Merge` function has only two input streams, not three.)

```
% $Id: DiffApproximsTest.oz,v 1.1 2009/02/28 20:52:54 leavens Exp leavens $
\insert 'DiffApproxims.oz'
\insert 'FloatTesting.oz'
{StartTesting 'DiffApproxims'}
{WithinTest {List.take {DiffApproxims 500.0 fun {$ X} X*X end 20.0} 9}
 '~=~' [540.0 290.0 165.0 102.5 71.25 55.625 47.8125 43.9062 41.9531]}
{WithinTest {List.take {DiffApproxims 100.0 fun {$ X} X*X*X end 10.0} 8}
 '~=~' [13300.0 4300.0 1675.0 831.25 526.562 403.516 349.316 324.048]}
{StartTesting 'done'}
```

Figure 8: Tests for Problem 21 on the previous page.

```
% $Id: DifferentiateTest.oz,v 1.2 2009/02/28 20:52:54 leavens Exp leavens $
\insert 'FloatTesting.oz'
\insert 'Differentiate.oz'
{StartTesting 'Differentiate'}
Millionth = 0.0000001
WithinMillionth = {TestMaker
                   Within
                   WithinLists
                   Millionth}
{WithinMillionth {Differentiate Millionth 500.0 fun {$ X} X*X end 20.0}
 '~=~' 40.0}
{WithinMillionth {Differentiate Millionth 100.0 fun {$ X} X*X*X end 10.0}
 '~=~' 300.0}
{StartTesting 'done'}
```

Figure 9: Tests for Problem 22 on the preceding page.

```
% $Id: RequestCalcTest.oz,v 1.1 2009/02/28 20:52:54 leavens Exp leavens $
\insert 'TestingNoStop.oz'
\insert 'RequestCalc.oz'

% Testing
declare Z
fun lazy {SGen Y} {Delay 5000} Y|{SGen Y+1} end
Z = {SGen 0}
{Browse Z}

{Delay 2000} {RequestCalc Z} {Browse requestin}
{Delay 2000} {RequestCalc Z.2} {Browse request2in}
```

Figure 10: Tests for Problem 25 on the preceding page.

27. (20 points) [EvaluateModels]

    Make a table listing all the different programming techniques, the characteristics of problems that are best solved with these techniques (i.e., when to use the techniques), and the name of at least one example of that technique.

programming technique	problem characteristics	example(s)
recursion (over grammars)		
higher-order functions		
stream programming		
lazy functions		

Extra Credit Problems

Extra credit problems are entirely optional. See the course grading policy for details.

28. (10 points; extra credit) (10 points; extra credit) [Concepts]

    Do problem 2 in section 4.11 of the textbook [RH04] (Threads and garbage collection).

    Note that the code in the problem uses _ instead of a variable name as the argument to `B` and to `Wait`.

29. (20 points; extra credit) [Concepts]

    Do problem 7 in section 4.11 of the textbook [RH04] (Programmed triggers using higher-order programming). Note that the example mentioned in the problem is the example with `DGenerate` and `DSum` that is the first example shown in section 4.3.3.1.

30. (15 points; extra credit) [Concepts]

    Do problem 18 in section 4.11 of the textbook [RH04] (Concurrency and Exceptions). (Hint: the "abstract machine semantics" is the operational semantics of Oz, which is described in the textbook's sections 2.4, 2.7.2, 4.1, and 4.9.1. You don't have to do anything formal or especially mathematical with the operational semantics.)

31. (20 points; extra credit) [Concepts]

    The instructions for this homework state that you are not to use Oz's built-in function `IsDet` in writing your code, since that is not part of the declarative model. Show why the use of `IsDet` takes you outside the declarative model by using it to simulate a Boolean-valued cell. (Of course, you are to do this only using the declarative model and `IsDet`. In particular you must not use cells in your solution!) Let's call the ADT that you are programming `CellWithIsDet`. Then your code should have the following interface.

```
NewCellWithIsDet: <fun {$ Bool}: CellWithIsDet>
CWIDAccess: <fun {$ CellWithIsDet}: Bool>
CWIDAssign: <proc {$ CellWithIsDet Bool}>
```

    Testing should show that you can change the value of a `CellWithIsDet` object from true to false and back again, as shown in Figure 11 on the next page.

Points

This homework's total points: 305. Total extra credit points: 65.

```
% $Id: CellWithIsDetTest.oz,v 1.1 2009/02/28 20:52:54 leavens Exp leavens $
\insert 'TestingNoStop.oz'
\insert 'CellWithIsDet.oz'

{StartTesting 'CellWithIsDet'}
declare
MyCWID = {NewCellWithIsDet true}
Alias = MyCWID
MyCWID2 = {NewCellWithIsDet false}
{Test {CWIDAccess MyCWID} '==' true}
{Test {CWIDAccess Alias} '==' true}
{Test {CWIDAccess MyCWID2} '==' false}
{CWIDAssign MyCWID false}
{Test {CWIDAccess MyCWID} '==' false}
{Test {CWIDAccess Alias} '==' false}
{Test {CWIDAccess MyCWID2} '==' false}
{CWIDAssign MyCWID2 true}
{Test {CWIDAccess MyCWID} '==' false}
{Test {CWIDAccess Alias} '==' false}
{Test {CWIDAccess MyCWID2} '==' true}
{CWIDAssign MyCWID2 true}
{Test {CWIDAccess MyCWID} '==' false}
{Test {CWIDAccess Alias} '==' false}
{Test {CWIDAccess MyCWID2} '==' true}
{CWIDAssign MyCWID2 false}
{Test {CWIDAccess MyCWID2} '==' false}
{CWIDAssign MyCWID2 false}
{Test {CWIDAccess MyCWID2} '==' false}
{CWIDAssign MyCWID2 true}
{Test {CWIDAccess MyCWID2} '==' true}
{StartTesting 'done'}
```

Figure 11: Testing for Problem 31 on the preceding page.

References

[CW98] Mary Campione and Kathy Walrath. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[DKS06] Denys Duchier, Leif Kornstaedt, and Christian Schulte. *The Oz Base Environment*. mozart-oz.org, June 2006. Version 1.3.2.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, Cambridge, Mass., 2004.