Homework 4: Declarative Concurrency

See Webcourses and the syllabus for due dates.
Note that you can't do quizzes on webcourses after their due date, so be sure to take any webcourses quizzes by the date they are due!
In this homework you will learn about the declarative concurrent model and basic techniques of programming in this model. 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 should look at Problem 23 on page 19 right now so you can be thinking about it while you do the other problems.
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 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, except where we explicitly allow them.) But please use all linguistic abstractions and syntactic sugars that are helpful.
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 concurrent model from the Oz library (base environment), especially functions like `Map` and `FoldR`.
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.
Read Chapter 4 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 4, through section 4.1 of the textbook [VH04] and answer the following questions.

1. (2 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.

   These different execution orders allow observable nondeterminism in a language like Java or C#. The way that happens is that a program can use mutable state (such as cells) to record information that depends on the exact interleaving of each thread, and this can be used to give different outputs.

   What property of Oz's declarative concurrent model makes this reasoning problem *not* possible in Oz?

2. (3 points) [Concepts] Read sections 4.1-4.2 in the textbook [VH04] and then take the quiz titled "Sections 4.1-4.2 quiz" on webcourses.
3. (3 points) [Concepts] [MapToLanguages] Read sections 4.3-4.4 in the textbook [VH04] and then take the quiz titled "Sections 4.3-4.4 quiz" on webcourses.
4. (6 points) [Concepts] Read section 4.5 in the textbook [VH04] and then take the quiz titled "Section 4.5 quiz" on webcourses.
5. (2 points) Read chapter 4, section 4.8 of the textbook [VH04] (you can skip section 4.6 and section 4.7.) and then take the quiz titled "Section 4.8 quiz" on webcourses.
6. (1 point) Read chapter 4, section 4.9 of the textbook [VH04] and then take the quiz titled "Section 4.9 quiz" on webcourses.

Regular Problems

We expect you'll do the problems in this section after reading various parts of the chapter.

Some of the following problems are from the textbook [VH04, section 4.11].

Thread and Dataflow Behavior Semantics

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

7. [Concepts]

   Consider the code in the first part of problem 4 of section 4.11 of the textbook [VH04] (Order-determining concurrency). Answer the following questions:

   (a) (3 points) How does the Oz runtime system determine the order of execution of the assignment statements in this example?

   (b) (2 points) State the order in which the additions executed.

8. (0 points) [Concepts] [UseModels] (suggested practice)

   Do problem 5 in section 4.11 of the textbook [VH04] (The Wait Operation).

9. (0 points) [Concepts] (suggested practice)

   Do problem 8 in section 4.11 of the textbook [VH04] (Dataflow behavior in a concurrent setting).

Streams and Lazy Functional Programming

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⟩

10. (10 points) [UseModels]

    Using the declarative concurrent model, write an incremental function,

    `IMerge : <fun {$ <IStream Int> <IStream Int>}: <IStream Int>>`

that takes two infinite stream of Ints (i.e., an <IStream Int>), both of which are in non-decreasing order, and returns an infinite stream of Ints that is similarly in non-decreasing order. The order of the Ints in the result is such that each Int, `I`, in one of the input streams appears in the result before any larger Int appears in the result.

Be sure that `IMerge` is incremental. That is, if it suspends, it should have already produced some output, and if the dataflow variable that it is suspended on is determined later, then it will be able to produce more output.

There are examples in Figure 1.

```
% $Id: IMergeTest.oz,v 1.2 2012/04/03 21:53:49 leavens Exp leavens $
\insert 'IMerge.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'IMergeTest $Revision: 1.2 $'}
local OddEnd Odds EvenEnd Evens in
   Odds = 1|3|5|7|9|OddEnd
   Evens = 2|4|6|8|10|EvenEnd
   {Test {List.take thread {IMerge Odds Evens} end 8} '==' [1 2 3 4 5 6 7 8]}
   local OddEnd2 EvenEnd2 in
      OddEnd = 21|25|29|OddEnd2
      EvenEnd = 12|18|22|30|50|60|70|80|EvenEnd2
      {Test {List.take thread {IMerge Odds Evens} end 16}
       '==' [1 2 3 4 5 6 7 8 9 10 12 18 21 22 25 29]}
   end
   local First3End First3 Second2 Second2End in
      First3 = 1|1|1|2|2|2|3|3|3|First3End
      Second2 = 1|1|2|2|3|3|Second2End
      {Test {List.take thread {IMerge First3 Second2} end 12}
       '==' [1 1 1 1 1 2 2 2 2 2 3 3]}
   end
end
{DoneTesting}
```

Figure 1: Tests for problem Problem 10 on the preceding page.

11. (10 points) [UseModels]

    Using the demand-driven concurrent model, write an incremental function

    `Averages : <fun lazy {$ <IStream <List Float>>}: <IStream Float> >`

    that takes an infinite stream of non-empty, finite Lists of Floats, `ISF`, and lazily returns an infinite stream of Floats, with the ith Float in the result being the average of all the Floats in the ith list in `ISF`. Remember that in Oz you cannot mix Floats and Ints in arithmetic expressions!

    Figure 2 on the next page has examples, which use the `WithinTest` function from Figure 3 on page 5.

12. (10 points) [UseModels] Write a lazy function

    `RepeatingListOf : <fun lazy {$ <List T>}: <IStream T> >`

    that, for some type T takes a non-empty finite list `Elements` of elements of type T, and lazily returns an infinite stream (i.e., an <IStream T>) whose elements repeat the individual elements of `Elements` endlessly. See Figure 4 on page 6 for examples.

```
% $Id: AveragesTest.oz,v 1.1 2012/03/27 15:42:08 leavens Exp $
\insert 'FloatTesting.oz'
\insert 'Averages.oz'
declare
fun lazy {RepeatingListOf Elements} % A helper for testing only
   {Append Elements {RepeatingListOf Elements}}
end
fun {FromToBy From To By}  % A helper for testing only
   %% REQUIRES: By \= 0.0 and none of From, To, or By is NaN.
   %% ENSURES: result is the finite list [From From+By From+2*By ... To]
   if (By > 0.0 andthen From > To) orelse (By < 0.0 andthen From < To)
   then nil
   else From|{FromToBy From+By To By}
   end
end
{StartTesting 'AveragesTest $Revision: 1.1 $'}
{WithinTest {List.take {Averages
                        {RepeatingListOf [[1.0] [1.0 2.0] [1.0 2.0 3.0] [1.0 2.0 3.0 4.0]
                                          {FromToBy 1.0 100.0 1.0} {FromToBy 1.0 1000.0 1.0}
                                         ]}}
             7} '~=~' [1.0 1.5 2.0 2.5 50.5 500.5 1.0]}
{WithinTest {List.take {Averages
                        {RepeatingListOf [[~1.0] [~1.0 ~2.0] [~1.0 ~2.0 3.0] [~1.0 ~2.0 3.0 4.0]
                                          {FromToBy ~1.0 ~100.0 ~1.0} {FromToBy ~1.0 ~1000.0 ~2.0}
                                         ]}}
             7} '~=~' [~1.0 ~1.5 0.0 1.0 ~50.5 ~500.0 ~1.0]}
{WithinTest {List.take {Averages
                        {RepeatingListOf [{FromToBy 0.0 6000.0 3.0} {FromToBy 0.0 3.0e3 100.0}
                                         ]}}
             4} '~=~' [3000.0 1500.0 3000.0 1500.0]}
{WithinTest {List.take {Averages
                        {RepeatingListOf {Map {FromToBy 1.0 10.0 1.0} fun {$ N} [0.0 N 2.0*N] end}}}
             10} '~=~' {FromToBy 1.0 10.0 1.0}}
{DoneTesting}
```

Figure 2: Tests for Problem 11 on the previous page.

```
% $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 3: Comparisons for floating point numbers. This code is available the this homework's zip file and also in the course `lib` directory.

```
\insert 'RepeatingListOf.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'RepeatingListOfTest $Revision: 1.5 $'}
{Test {Nth {RepeatingListOf [1]} 1} '==' 1}
{Test {Nth {RepeatingListOf [7]} 500} '==' 7}
{Test {Nth {RepeatingListOf [2]} 999999} '==' 2}
{Test {List.take {RepeatingListOf [2 3 4]} 7} '==' [2 3 4 2 3 4 2]}
{Test {List.take {RepeatingListOf [a b c d e]} 11} '==' [a b c d e a b c d e a]}
{Test {List.take {RepeatingListOf [home work]} 5} '==' [home work home work home]}
{TestString {List.take {RepeatingListOf "homework!"} 11} '==' "homework!ho"}
{DoneTesting}
```

Figure 4: Tests for Problem 12 on page 3.

13. (15 points) [UseModels]

    Write a lazy function

    BlockIStream: <**fun lazy** {$ <IStream <Char>> <Int>}: <IStream <List <Char> > >

    that takes an infinite stream of characters, `IStrm`, and an integer, `BlockSize`, and which lazily returns an infinite stream of lists of characters, where each list in the result has exactly `BlockSize` elements, consisting of the first `BlockSize` elements of `IStrm`, followed by a list containing the next `BlockSize` elements of `IStrm`, and so on. That is, a call to `BlockIStream` chunks the characters in `IStrm` into strings of length `BlockSize`. (Hint: in your solution, you may want to use `List.take` and `List.drop`. Note that `{List.drop L N}` returns the list `L` without the first `N` elements.)

    See Figure 5 for our tests.

```
% $Id: BlockIStreamTest.oz,v 1.5 2011/10/31 01:18:11 leavens Exp $
\insert 'BlockIStream.oz'
\insert 'RepeatingListOf.oz' % from problem above, used here to make the tests
\insert 'TestingNoStop.oz'
declare
{StartTesting 'BlockIStreamTest $Revision: 1.5 $'}
fun lazy {From Char} Char|{From Char+1} end % for testing only
{TestLOS {List.take {BlockIStream {From &a} 3} % &a is the character a
       8}
 '==' ["abc" "def" "ghi" "jkl" "mno" "pqr" "stu" "vwx"]}
{TestLOS {List.take {BlockIStream {RepeatingListOf "Now is the time for..."} 1}
       12}
 '==' ["N" "o" "w" " " "i" "s" " " "t" "h" "e" " " "t"]}
{TestLOS {List.take {BlockIStream {RepeatingListOf "Now is the time for..."} 3}
       4}
 '==' ["Now" " is" " th" "e t"]}
{TestLOS {List.take {BlockIStream {RepeatingListOf "Now is the time for..."} 3}
       9}
 '==' ["Now" " is" " th" "e t" "ime" " fo" "r.." ".No" "w i"]}
{TestLOS {List.take {BlockIStream {RepeatingListOf "Now is the time for..."} 6}
       7}
 '==' ["Now is" " the t" "ime fo" "r...No" "w is t" "he tim" "e for."]}
{DoneTesting}
```

Figure 5: Tests for Problem 13.

14. (15 points) [UseModels]

    Using `BlockIStream`, write a lazy function

```
EncryptIStream: <fun lazy {$ <IStream <Char>> <Int> <fun {$ <List <Char> >}: <List <Char> >}
                   : <IStream <List <Char> > >
```

    that takes 3 arguments: an infinite stream of characters, `IStrm`, an integer, `BlockSize`, and a function, `Encrypt`. The `EncryptIStream` function lazily returns an infinite stream of lists of characters, where each list in the result is the result of applying `Encrypt` to a list containing `BlockSize` elements of `IStrm`. The first list in the result is the result of applying `Encrypt` to the first `BlockSize` elements of `IStrm`, and this is followed by the result of applying `Encrypt` to the next `BlockSize` elements of `IStrm`, and so on. That is, a call to `EncryptIStream` first chunks the characters in `IStrm` into strings of length `BlockSize`, and then applies `Encrypt` to each resulting list of strings. Figure 6 shows some examples, written using various (cryptographically poor) `Encrypt` functions.

```
\insert 'EncryptIStream.oz'
\insert 'RepeatingListOf.oz' % from problem above, used here to make the tests
\insert 'TestingNoStop.oz'
declare
{StartTesting 'EncryptIStreamTest $Revision: 1.4 $'}
fun {NoEncryption Str} Str end
fun {ReverseNoEncryption Str} {Reverse {NoEncryption Str}} end
fun {CeaserCypher Str} {Map Str fun {$ C}(C+1) mod 512 end} end
fun {ReverseCeaserCypher Str} {Reverse {CeaserCypher Str}} end
{TestLOS {List.take {EncryptIStream {RepeatingListOf "Now is the time for..."} 12 NoEncryption}
       3}
 '==' ["Now is the t" "ime for...No" "w is the tim" ]}
{TestLOS {List.take {EncryptIStream {RepeatingListOf "We're off to see the Wizard, the Wonderful Wizard of Oz"}
                  3 ReverseNoEncryption}
       18}
 '==' ["'eW" " er" "ffo" "ot " "es " "t e" " eh" "ziW" "dra" "t ," " eh"
       "noW" "red" "luf" "iW " "raz" "o d" "O f"]}
{TestLOS {List.take {EncryptIStream {RepeatingListOf "Now is the time for..."} 5 CeaserCypher}
       7}
 '==' ["Opx!j" "t!uif" "!ujnf" "!gps/" "//Opx" "!jt!u" "if!uj"]}
{TestLOS {List.take {EncryptIStream {RepeatingListOf "Now is the time for..."} 5 ReverseCeaserCypher}
       7}
 '==' ["j!xpO" "fiu!t" "fnju!" "/spg!" "xpO//" "u!tj!" "ju!fi"]}
{DoneTesting}
```

Figure 6: Tests for Problem 14.

15. (3 points) [Concepts] Does your function `EncryptIstream` in question 14 have to be lazy itself? Answer “yes” or “no” and give a brief explanation.

16. (10 points) [Concepts] [UseModels] Infinite data structures don't have to just be lists. For example, the type ⟨ITree ⟨S⟩ ⟨T⟩⟩ below is a type of infinite binary trees.

    ⟨ITree ⟨S⟩ ⟨T⟩⟩ ::= tree(key: ⟨S⟩ answer: ⟨T⟩ left: ⟨ITree ⟨S⟩ ⟨T⟩⟩ right: ⟨ITree ⟨S⟩ ⟨T⟩⟩)

    Contrast this with the similar grammar for finite binary trees

    ⟨Tree ⟨S⟩ ⟨T⟩⟩ ::=leaf
    | tree(key: ⟨S⟩ answer: ⟨T⟩ left: ⟨Tree ⟨S⟩ ⟨T⟩⟩ right: ⟨Tree ⟨S⟩ ⟨T⟩⟩)

    Your task is to write a lazy function

    ```
    ITreeMap : <fun lazy {$ <ITree <S> <T>> <fun {$ <S>}:<T>>}: <ITree <S> <T>>>
    ```

    which for some types `S` and `T` takes an ⟨ITree ⟨S⟩ ⟨T⟩⟩, `Tr`, and a function, `F`, and which returns an ITree with that is the same as `Tr`, except that each answer field contains the result of applying `F` to the corresponding key field's value. Figure 7 on the following page contains examples.

17. (20 points) [Concepts] [UseModels]

    Do problem 16 in section 4.11 of the textbook [VH04] (By-need execution).

    To explain this problem and provide a test, call your procedure that solves this problem `RequestCalc`.

    Note that `RequestCalc` should be a procedure, not a function.

    When you run the test Figure 8 on page 11 it should show in the Browser window immediately several lines, with almost no delay, including the line `'reached the end'`. That indicates that the caller of RequestCalc is not waiting, which is what the problem specifies. Then after the delay specified in the test (about 10 seconds) you should see `Z` change to `0|_`, and `X` change to 3 and `A` change to `an_atom`. Then after another delay, you should see the line where `Z` was change to `0|1|_`. Shortly after that you should see another line of output appear, summarizing this.

    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. If your code for `RequestCalc` waits for the calculation, you should see messages about assertions failing. If your code for `RequestCalc` does not request the calculations, then you won't see the variables in the Browser's output changing.

    For this problem, turn in your code. If the tests work as described above, you don't have to paste any testing output into the answer box. However, if your tests have errors, then paste the `*Oz Emulator*` buffer output showing the error into the answer box. In all cases, upload your code for `RequestCalc` in your file `RequestCalc.oz`.

The following problems, inspired by a paper written by John Hughes, relate to modularization of numeric code using streams and lazy execution. In particular, we will explore numerical differentiation.

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 3 on page 5. The testing code for floating point numbers is shown in Figure 9 on page 12.

```
% $Id: ITreeMapTest.oz,v 1.2 2012/03/27 16:07:37 leavens Exp leavens $
\insert 'ITreeMap.oz'
\insert 'TestingNoStop.oz'

local
   % Just for testing, you don't have to implement ITreeTake
   % ITreeTake: <fun {$ <ITree S T> <Int>}: <Tree S T>>
   fun {ITreeTake T Level}
      if Level == 0
      then leaf
      else case T of
              tree(key: K answer: A left: L right: R) then
              tree(key: K answer: A left: {ITreeTake L Level-1} right: {ITreeTake R Level-1})
           end
      end
   end
   % Just for testing, you don't have to implement ITreeGenerate
   % This is lazy because that's the easiest way to generate
   % new dataflow variables for subtrees, and control it.
   % ITreeGenerate: <fun lazy {$ <T> <fun {$ <S>}: T> <fun {$ <T>}: <T>}: <ITree <T> <T>>>
   fun lazy {ITreeGenerate InitialK LeftFun RightFun}
      tree(key: InitialK answer: _ % an undetermined dataflow variable
           left: {ITreeGenerate {LeftFun InitialK} LeftFun RightFun}
           right: {ITreeGenerate {RightFun InitialK} LeftFun RightFun})
   end
in
   {StartTesting 'ITreeMapTest.oz $Revision: 1.2 $'}
   {Test {ITreeTake {ITreeMap {ITreeGenerate 1 fun {$ K} K+2 end fun {$ K} K+3 end} fun {$ K} K*10 end}
          3}
    '==' tree(key: 1 answer: 10
              left: tree(key: 3 answer: 30
                         left: tree(key: 5 answer: 50 left: leaf right: leaf)
                         right: tree(key: 6 answer: 60 left: leaf right: leaf))
              right: tree(key: 4 answer: 40
                          left: tree(key: 6 answer: 60 left: leaf right: leaf)
                          right: tree(key: 7 answer: 70 left: leaf right: leaf)))}
   {Test {ITreeTake {ITreeMap {ITreeGenerate 5 fun {$ K} K*2 end fun {$ K} K*3 end} fun {$ K} K*100 end}
          4}
    '==' tree(key: 5 answer: 500
              left: tree(key: 10 answer: 1000
                         left: tree(key: 20 answer: 2000
                                    left: tree(key: 40 answer: 4000 left: leaf right: leaf)
                                    right: tree(key: 60 answer: 6000 left: leaf right: leaf))
                         right: tree(key: 30 answer: 3000
                                     left: tree(key: 60 answer: 6000 left: leaf right: leaf)
                                     right: tree(key: 90 answer: 9000 left: leaf right: leaf)))
              right: tree(key: 15 answer: 1500
                          left: tree(key: 30 answer: 3000
                                     left: tree(key: 60 answer: 6000 left: leaf right: leaf)
                                     right: tree(key: 90 answer: 9000 left: leaf right: leaf))
                          right: tree(key: 45 answer: 4500
                                      left: tree(key: 90 answer: 9000 left: leaf right: leaf)
                                      right: tree(key: 135 answer: 13500 left: leaf right: leaf))))}
   {DoneTesting}
end
```

Figure 7: Tests for Problem 16 on the previous page.

```
% $Id: RequestCalcTest.oz,v 1.6 2011/11/11 14:06:40 leavens Exp $
\insert 'TestingNoStop.oz'
\insert 'RequestCalc.oz'

% Testing
declare
% A testing helper
fun {TimeProc P}
   %% Ensures: Result is (slightly more than)
   %%          the time in milliseconds used to execute P
   T1 = {Property.get 'time'}
   {P}
   T2 = {Property.get 'time'}
in
   T2.total - T1.total
end
% Another testing helper
fun lazy {LTail Ls} Ls.2 end
DelayTime = 10000 % 10 seconds

% Make reporting go to the Browser and the Emulator (both)!
{SetReporter BothReporter}

{StartTesting 'RequestCalcTest $Revision: 1.6 $'}
fun lazy {SGen Y} {Delay DelayTime} Y|{SGen Y+1} end
Z = {SGen 0}
{Browse Z}

{Assert {TimeProc proc {$} {RequestCalc Z} {Browse requestin} end} < 1000}
Q = {LTail Z}
{Browse Q}
{Assert {TimeProc proc {$} {RequestCalc Q} {Browse requestQin} end} < 1000}

fun lazy {LThree} {Delay DelayTime} 3 end
X = {LThree}
{Browse X}

{Assert {TimeProc proc {$} {RequestCalc X} {Browse requestXin} end} < 1000}

fun lazy {LAtom} {Delay DelayTime} an_atom end
A = {LAtom}
{Browse A}
{Assert {TimeProc proc {$} {RequestCalc A} {Browse requestAin} end} < 1000}
{Browse 'reached the end'}
{Delay 2*DelayTime+1000}
{Browse 'by now the browser should show values for Z, Q, X, and A'}
```

Figure 8: Tests for Problem 17 on page 9.

```
% $Id: FloatTesting.oz,v 1.2 2011/11/07 23:50:24 leavens Exp leavens $
% Testing for floating point numbers.
% AUTHOR: Gary T. Leavens

\insert 'TestingNoStop.oz'
\insert 'FloatPredicates.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 note the failure.
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 {ReportAbout
            {Value.toVirtualString Actual 5 20}
            # ' ' # Connective # ' '
            # {Value.toVirtualString Expected 5 20}}
      else {NotifyAbout
            '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 9: Testing code for floating point numbers. This sends output to where the `ReportAbout` and `NotifyAbout` hooks send the output. (By default this is the `*Oz Emulator*` window.) The file `FloatPredicates` is shown in Figure 3 on page 5. This file is available in this homework's zip file and also in the course `lib` directory.

18. (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 lazy infinite 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 10 are written using the `WithinTest` procedure from Figure 9 on the previous page. Notice also that, since the function `Next` in the testing file is curried, we don't pass `Next` itself to `IStreamIterate`, but instead pass the value of applying `Next` to some number.

```
% $Id: IStreamIterateTest.oz,v 1.3 2010/11/01 00:58:00 leavens Exp $
\insert 'IStreamIterate.oz'
\insert 'FloatTesting.oz'
declare
% Next: <fun {$ <Float>}: <fun {$ <Float>}: Float>>
fun {Next N}
   fun {$ X}
      (X + N / X) / 2.0
   end
end
{StartTesting 'IStreamIterateTest $Revision: 1.3 $'}
{WithinTest {List.take {IStreamIterate {Next 1.0} 1.0} 7}
 '~=~' [1.0 1.0 1.0 1.0 1.0 1.0 1.0]}
{WithinTest {List.take {IStreamIterate {Next 9.0} 1.0} 7}
 '~=~' [1.0 5.0 3.4 3.0235 3.0001 3.0 3.0]}
{WithinTest {List.take {IStreamIterate {Next 200.0} 1.0} 7}
 '~=~' [1.0 100.5 51.245 27.574 17.414 14.449 14.145]}
{RelativeTest {List.take {IStreamIterate {Next 0.144} 7.0} 9}
 '~=~' [7.0 3.5103 1.7757 0.92838 0.54174 0.40378 0.3802 0.37947 0.37947]}
{RelativeTest {List.take {IStreamIterate fun {$ 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]}
{DoneTesting}
```

Figure 10: Tests for Problem 18.

19. (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 11 gives some examples.

```
% $Id: ConvergesToTest.oz,v 1.3 2011/10/31 01:18:11 leavens Exp $
\insert 'ConvergesTo.oz'
\insert 'FloatTesting.oz'

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

{StartTesting 'ConvergesToTest $Revision: 1.3 $'}
{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}
{DoneTesting}
```

Figure 11: Tests for Problem 19.

You may know that the definite integral of a function `F` between two values `X` and `X+Delta` can be approximated by the function in Figure 12. See Figure 13 for an explanation.

```
% $Id: Trapezoid.oz,v 1.1 2011/10/30 03:00:00 leavens Exp $
declare
% Trapezoid: <fun {$ <fun {$ <Float}: Float> <Float> <Float>}: Float>
fun {Trapezoid F X Delta}
   % REQUIRES: Delta >= 0.0
   % ENSURES: result is the area of the trapezoid inside the points:
   %          (X, {F X}),    (X+Delta, {F X+Delta}),
   %          (X, 0.0), and (X+Delta, 0.0).
   local
      Y1 = {F X}
      Y2 = {F X + Delta}
      Min_Height = {Min Y1 Y2}
      Max_Height = {Max Y1 Y2}
   in
      if Y1 >= 0.0 orelse Y2 >= 0.0
      then
         Delta * Min_Height  % area of base rectangle
         + 0.5 * Delta * (Max_Height - Min_Height) % area of top triangle
      else
         Delta * Max_Height % area of base rectangle below x axis
         + 0.5 * Delta * (Min_Height - Max_Height) % area of bottom triangle
      end
   end
end
```

Figure 12: The `Trapezoid` function found in `Trapezoid.oz`.

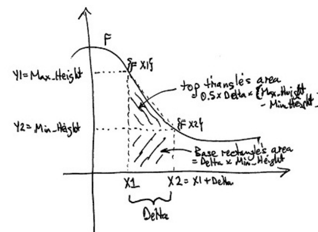

Figure 13: Illustration for the Trapezoid function.

Since `F` may not be linear, good approximations are obtained when `Delta` is small. However, if `Delta` is too small, then floating point errors may swamp the result. One way to choose `Delta` is to work with smaller and smaller values of `Delta`, until the integral converges. In the following problems, you will use `Trapezoid` as the basis for a numerical integration algorithm.

20. (15 points) [UseModels] To compute a definite integral between two values `X1` and `X2`, where `X2 > X1`, we divide the interval between `X1` and `X2` into `N` subintervals. Then we add the areas of the `N` trapezoids of width `Delta` = `(X2 - X1)/{IntToFloat N}` together. See Figure 14 on the next page.

    Your task in this problem is to write, in Oz, a function

```
TrapezoidSum: <fun {$ <fun {$ <Float>}: <Float>> <Float> <Float> <Int>}: <Float>>
```

    that takes a function, `F`, a Float `X`, a positive Float `Delta`, and an Int `N`. This function returns the sum of areas of `N` trapezoids. For `I` between 1 and `N`, the `I`th such trapezoid is between the points:

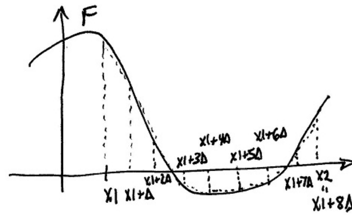

Figure 14: Illustration for the TrapezoidSum function, showing the case where `N` is 8.

(`X+(Delta*{IntToFloat I-1})`,0.0), (`X+(Delta*{IntToFloat I})`,0.0), (`X+(Delta*{IntToFloat I-1})`, `{F X+(Delta*{IntToFloat I-1})}`), and (`X+(Delta*{IntToFloat I})`, `{F X+(Delta*{IntToFloat I})}`). You should use `Trapezoid` in your solution. Figure 15 gives test cases.

```
% $Id: TrapezoidSumTest.oz,v 1.1 2011/10/30 03:00:00 leavens Exp $
\insert 'TrapezoidSum.oz'
\insert 'FloatTesting.oz'
{StartTesting 'TrapezoidSumTest $Revision: 1.1 $'}
local Id = fun {$ X} X end in
   {WithinTest {TrapezoidSum Id 0.0 1.0 1} '~=~' {Trapezoid Id 0.0 1.0}}
   {WithinTest {TrapezoidSum Id 0.0 1.0 2}
    '~=~' {Trapezoid Id 0.0 1.0} + {Trapezoid Id 1.0 1.0}}
   {WithinTest {TrapezoidSum Id 0.0 1.0 100} '~=~' 100.0*100.0/2.0}
end
local Square = fun {$ X} X*X end in
   {WithinTest {TrapezoidSum Square 0.0 1.0 1} '~=~' {Trapezoid Square 0.0 1.0}}
   {WithinTest {TrapezoidSum Square 0.0 1.0 4}
    '~=~' {Trapezoid Square 0.0 1.0} + {Trapezoid Square 1.0 1.0}
          + {Trapezoid Square 2.0 1.0} + {Trapezoid Square 3.0 1.0}}
   {WithinTest {TrapezoidSum Square 0.0 1.0 100} '~=~' 3.3335e5}
end
{WithinTest {TrapezoidSum fun {$ X} X*X*X end ~10.0 1.0 12}
 '~=~' ~2520.0}
{WithinTest {TrapezoidSum fun {$ X} ~X*X*X - 3.0*X*X - 20.0 end 0.0 1.0 15}
 '~=~' ~16395.0}
{DoneTesting}
```

Figure 15: Tests for Problem 20 on the previous page.

21. (15 points) [UseModels] In this problem your task is to write a function

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

such that `{IntegApproxims F X1 X2}` returns an infinite lazy list of approximations to the definite integral of `F` between `X1` and `X2`, starting with 2 intervals (so `Delta` = `(X2 - X1)/2.0`). Each subsequent approximation doubles the previous number of intervals, halving their size, and thus yielding better and better approximations to the definite integral $\int_{X1}^{X2} F(X)dX$. Examples are given in Figure 16 on the following page.

Hint: Use `TrapezoidSum` and `IStreamIterate`.

```
% $Id: IntegApproximsTest.oz,v 1.3 2011/10/31 00:51:32 leavens Exp $
\insert 'IntegApproxims.oz'
\insert 'FloatTesting.oz'
{StartTesting 'IntegApproximsTest $Revision: 1.3 $'}
% Expected results checked with Wolfram Alpha (http://www.wolframalpha.com)
{WithinTest {List.take {IntegApproxims fun {$ X} X end 0.0 20.0} 5}
 '~=~' [200.0 200.0 200.0 200.0 200.0]}
{WithinTest {List.take {IntegApproxims fun {$ X} X end ~1.0 1.0} 5}
 '~=~' [0.0 0.0 0.0 0.0 0.0]}
{RelativeTest {List.take {IntegApproxims fun {$ X} X*X end 0.0 20.0} 9}
 '~=~' [3000.0 2750.0 2687.5 2671.9 2668.0 2667.0 2666.7 2666.7 2666.7]}
{RelativeTest {List.take {IntegApproxims fun {$ X} X*X*X end 0.0 128.0} 9}
 '~=~' [8.3886e007 7.1303e007 6.8157e007 6.7371e007 6.7174e007 6.7125e007
        6.7113e007 6.711e007 6.7109e007]}
{RelativeTest {List.take {IntegApproxims fun {$ X} 2.0*X*X*X*X + 5.0*X*X*X end
                          ~5.0 5.0} 12}
 '~=~' [6250.0 3515.6 2758.8 2565.0 2516.3 2504.1 2501.0 2500.3 2500.1 2500.0
        2500.0 2500.0]}
{WithinTest {List.take {IntegApproxims fun {$ X} {Cos X*X} + {Sin X} end
                          0.0 0.125} 5}
 '~=~' [0.1328 0.1328 0.1328 0.1328 0.1328]}
{DoneTesting}
```

Figure 16: Tests for Problem 21 on the previous page.

22. (15 points) [UseModels] Using the pieces given above, in particular `ConvergesTo` and `IntegApproxims`, write a function

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

    such that `{Integrate F X1 X2 Epsilon}` returns an approximation to the definite integral $\int_{X1}^{X2} F(X)dX$ that is accurate to within `Epsilon`. Use the previous problem (Problem 21 on page 16) to obtain an approximation that converges to within `Epsilon`. Examples are given in Figure 17.

    Hint, you can use `WithinMaker` from our `FloatTesting` file if you wish (see Figure 9 on page 12).

```
% $Id: IntegrateTest.oz,v 1.2 2011/10/31 00:51:32 leavens Exp $
\insert 'Integrate.oz'
\insert 'FloatTesting.oz'
declare
fun {Int F X1 X2} % Integrate with fixed Epsilon, to avoid redundancy in tests
   {Integrate F X1 X2 StandardTolerance/2.0}
end
{StartTesting 'IntegrateTest $Revision: 1.2 $'}
% Expected results from Wolfram Alpha (http://www.wolframalpha.com)
{WithinTest {Integrate fun {$ X} X end 0.0 20.0 1.0e~5} '~=~' 200.0}
{WithinTest {Integrate fun {$ X} X end ~1.0 1.0 1.0e~5} '~=~' 0.0}
{WithinTest {Int fun {$ X} ~X end 3.0 4.0} '~=~' ~3.5}
{WithinTest {Integrate fun {$ X} X*X end 0.0 0.5 1.0e~7} '~=~' 0.04166667}
{WithinTest {Integrate Cos 0.0 3.141592653589793 1.0e~19} '~=~' 1.96193e~16}
{WithinTest {Int Sin 0.0 1.0} '~=~' 0.459698}
{WithinTest {Int fun {$ X} {Cos {Sin X}} + 3.0*X*X end 0.0 0.5}
 '~=~' 0.605414}
{WithinTest {Int fun {$ X} {Cos {Sin X*X}} end ~0.5 0.5} '~=~' 0.993839}
{WithinTest {Integrate fun {$ X} {Pow X 33.0} end 0.4 0.45 1.0e~19}
 '~=~' 4.6747220817552974e~14}
{WithinTest {Int fun {$ X} {Log X} end 0.1 0.25} '~=~' ~0.266315}
{WithinTest {Int fun {$ X} {Sqrt 1.0+4.0*X} end 0.0 0.25} '~=~' 0.304738}
{DoneTesting}
```

Figure 17: Tests for Problem 22.

Problems and Programming Models

The following problems ask you to compare different programming models and the problems they are good at solving.

23. (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		

    Please put your answer into the answer box on webcourses. You can use HTML to make a table by checking the "use HTML" box. (Or if you want to use plain text use HTML and wrap `<pre>` and `</pre>` around your answer to preserve the formatting.)

Points

This homework's total points: 200.

References

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