Homework 3: Declarative Programming

See Webcourses and the syllabus for due dates.

Don't start these problems at the last minute! These are mostly programming problems that you will need time to complete.

In this homework you will learn basic techniques of recursive programming over various types of (recursively-structured) data, and more advanced functional programming techniques such as using higher-order functions to abstracting from programming patterns, and using higher-order functions to model infinite data [UseModels] [Concepts]. Many of the problems exhibit polymorphism [UseModels] [Concepts]. The problems as a whole illustrate how functional languages work without hidden side-effects [EvaluateModels].

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 or code that is excessively hard to follow. Avoid duplicating code by using helping functions or by using syntactic sugars and local definitions.

Code for programming problems should be written in Oz's declarative model, so do not use either cells or cell 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 in the declarative programming model 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 model from the Oz library (base environment), especially functions like `Map` and `FoldR`.

For all Oz programing exercises, you must run your code using the Mozart/Oz system. For programming exercises in your favorite language, you should use the most standard version of the language you can find. 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: Turn in (on Webcourses) your code and output of your testing for each problem that requires code. Please upload code as a plain (text) file with the name given in the problem or testing file and with the suffix `.oz`. Please upload test output and English answers by pasting them into the answer box in the assignment on Webcourses. If you have a mix of code and English for a problem, please use the answer box for the English and upload a `.oz` file for the code. (In any case, don't put spaces or tabs 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.

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

For background, you should read Chapter 3 of the textbook [VH04]. Also read "Following the Grammar" [Lea07] and follow its suggestions for organizing your code. You may also want to read a tutorial on the concepts of functional programming languages, such as Hudak's computing survey article mentioned in the syllabus. See also the course code examples page (and the course resources page).

Reading Problems

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

Read chapter 3, through section 3.1 of the textbook [VH04] and answer the following questions.

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

   How does one write deterministic programs in C, C++, or Java? (Give a brief explanation.)

Read section 3.2 of the textbook and answer the following questions.

2. (5 points) [UseModels]

   In Oz, write an iterative function

   ```
   ListMin: <fun {$ <List T>}: T>
   ```

   that for some type T takes a non-empty list of elements of type T Ls, and returns the least element in Ls. The following are examples that you can find in our test file LisMinTest.oz.

   ```
   {Test {ListMin a|a|a|a|a|nil} '==' a}
   {Test {ListMin z|nil} '==' z}
   {Test {ListMin a|nil} '==' a}
   {Test {ListMin a|z|nil} '==' a}
   {Test {ListMin z|a|nil} '==' a}
   {Test {ListMin a|b|c|d|a|c|e|z|nil} '==' a}
   {Test {ListMin 4|0|2|0|nil} '==' 0}
   {Test {ListMin 99|86|12|~3|nil} '==' ~3}
   {Test {ListMin 100000|8600000|12222|~999999999999999999999999999999|nil}
             '==' ~999999999999999999999999999999}
   {Test {ListMin [now is the time 'for' change]} '==' change}
   {Test {ListMin [the code examples page gives access to the code examples
   ```

   You can assume that ListMin is called with a non-empty list as an argument.

   Your code must have iterative behavior. (So it must use tail recursion!)

   Put your code in a file ListMin.oz. After doing your own testing, run our tests in ListMinTest.oz. For this and all coding problems, be sure to hand in both your code and the output of running our tests (see the instructions above).

Skim section 3.3 and read section 3.4 through 3.4.1 of the textbook and answer the following questions.

3. [Concepts]

   (a) (5 points) Give an example Oz expression, other than leaf, that defines a value of the type ⟨BTree Int⟩. Recall that a ⟨Literal⟩ can be an atom such as atm and that ⟨Int⟩ contains both integer literals, such as 42.

   (b) (2 points) Which of the following do *not* match the grammar for <List Literal>?

      1. nil
      2. a|b|c|d|e
      3. a|b|c|d|e|nil
      4. [a b c]

Read section 3.4.2 up to and including section 3.4.2.6 of the textbook, and read the "Following the Grammar" handout.

4. (0 points) [UseModels]

   (Try the self-test on "following the grammar" on Webcourses.)

Read section 3.4.2.7, skim over sections 3.4.4 and 3.4.5, read section 3.4.6, and skim over 3.4.7 and 3.4.8 of the textbook and answer the following questions.

5. [Concepts]

   Some students take a liking to the `Flatten` function described in section 3.4.4 (on page 143) of the textbook. But consider this: is calling Flatten on the list argument a useful first step in the solution of the following problems? (For each answer "yes" or "no" and give a brief explanation; note that you are *not* being asked to program these!)

   (a) (2 points) A function `Has3List` that takes a list of lists `LL` and returns true just when `LL` contains a list with exactly three elements? For example, `{Has3List [[a b] [c] [d e f] [g] [h]]}` should return **`true`**, but , and `{Has3List [[a [] b]]}` should return **`true`**, but `{Has3List [[a b] [c] [d e] [] [g] [h]]}` should return **`false`**

   (b) (2 points) A function `InsertAfter` that takes a list of lists `LL` and two atoms: `AfterThis` and `What`; this function inserts `What` after each occurrence of `AfterThis` throughout `LL`. For example:
   `{InsertAfter [[a b c] [c p a]] a z}` returns `[a z b c] [c p a z]]` and
   `{InsertAfter [[[[a] [[b c] []] [[[[c p [a]]]]]]]] a q}` returns `[[[[a q] [[b c] []] [[[[c p [a q]]]]]]]]`.

Read section 3.5 of the textbook (skimming 3.5.3 and 3.5.4) and answer the following questions.

6. (0 points) (suggested practice) [Concepts] Ignoring statement sequences, which kernel language statements might get into infinite loops?

Read section 3.6 of the textbook and answer the following questions.

7. (5 points) [Concepts]

   Briefly describe what the function `All`, whose code is given below, does.

```
declare
fun {All Ls}
   {FoldR Ls fun {$ X Y} X andthen Y end true}
end
```

Read section 3.7 of the textbook (you can just skim 3.7.3) and answer the following questions.

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

   Why should one use private (or protected) visibility to hide the internal representation of an abstract datatype in C++, C#, or Java?

Read section 3.8 of the textbook (you can skim 3.8.1 through 3.8.3) and answer the following questions.

9. (5 points) [Concepts] [MapToLanguages] If one were to add input from files or direct input from a user as part of the declarative programming model, what would change about the model?

Regular Problems

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

Iteration

Material on iteration and tail recursion is found in section 3.2 and 3.4.2.3 and 3.4.3.

10. (10 points) [UseModels]

Do problem 5 in section 3.10 of the textbook [VH04] (iterative SumList).

Put your code in a file `SumList.oz`. After doing your own testing, run our tests in `SumListTest.oz` (see below).

```
% $Id: SumListTest.oz,v 1.3 2011/10/04 02:02:08 leavens Exp $
\insert 'SumList.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'SumList $Revision: 1.3 $'}
{Test {SumList nil} '==' 0}
{Test {SumList 3|nil} '==' 3}
{Test {SumList ~2|3|nil} '==' 1}
{Test {SumList [7 8 ~2 3]} '==' 16}
{Test {SumList [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0]} '==' 24}
{Test {SumList [4 4 2 1 99 105 3004 999999]} '==' 1003218}
{DoneTesting}
```

11. [MapToLanguages]

  (a) (5 points) Carefully record the time you spend on the next part of this problem, and write in the answer box on webcourses: (i) programming language you used and (ii) the number of minutes you spent on writing the code for `SumList` itself in your favorite language, and (iii) the number of minutes you spent in writing tests, testing, and debugging your code.

  (b) (10 points) Using your favorite programming language, write `SumList`, from problem10 above, and test it. If you are using Java, C++, or C for this problem, then you may use our implementation of Oz's lists in which you can findare in the subdirectories `Java`, `C++`, and `C` of the tests zip file that you can get from webcourses (in the zip file attached to problem 1).

  Hand in your code, including code for testing, as attachments. Paste the test output into the answer box. Your testing code should be approximately what we did for testing in Oz (and thus should be separated from the code for `SumList`.

Following the Grammar

Material on following the grammar is found in section 3.4, especially section 3.4.2, and in detail with many examples in the "Following the Grammar" handout.

12. (10 points) [UseModels]

  Write a function

```
Hep: <fun {$ <List Atom>}: <List Atom> >
```

  that takes a list of atoms, `Txt`, and returns a list just like `Txt` but with the following substitutions made each time they appear in `Txt`:

  - `you` is replaced by `u`,
  - `are` is replaced by `r`,
  - `your` is replaced by `ur`,
  - `by_the_way` is replaced by `btw`,
  - `for_your_information` is replaced by `fyi`
  - `boyfriend` is replaced by `bf`,
  - `girlfriend` is replaced by `gf`,
  - `be_right_back` is replaced by `brb`,

- see_you is replaced by cya, and
- great is replaced by gr8.

This list is complete (for this problem).

The examples in Figure 1 are written using the Test procedure from the course library. They r also found in our testing file HepTest.oz which u can get from webcourses (in the zip file attached to problem 1). Be sure to turn in both ur code and the output of our tests on webcourses.

```
% $Id: HepTest.oz,v 1.1 2011/10/04 02:02:08 leavens Exp $
\insert 'Hep.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'HepTest $Revision: 1.1 $'}
{Test {Hep nil} '==' nil}
{Test {Hep [you you you you]} '==' [u u u u]}
{Test {Hep [you know i will see_you soon]}
      '==' [u know i will cya soon]}
{Test {Hep [by_the_way you must see my girlfriend she is great]}
      '==' [btw u must see my gf she is gr8]}
{Test {Hep [for_your_information you are a pig see_you later when you find me a boyfriend]}
      '==' [fyi u r a pig cya later when u find me a bf]}
{Test {Hep [by_the_way i will be_right_back]} '==' [btw i will brb]}
{DoneTesting}
```

Figure 1: Tests for problem 12.

Put ur code in a file Hep.oz and test using our tests. BTW, we will take some number of points off if u have repeated code in ur solution. U can avoid repeated code by using a helping function or a case-expression. A case-expression would be used in a larger expression to form the result list, like: **case** ... **end** | ....

13. (10 points) [UseModels]

    Write a function

    ```
    IsAList: <fun {$ <Value>}: <Bool> >
    ```

    that takes an Oz Value, Val, and returns true just when Val is an association list (which in this problem we call an "AList"). That is, it returns true if and only if Val is a list of #-pairs, as in Figure 2.

⟨AList S T⟩ ::= nil
  | ⟨List ⟨#-Pair S T⟩⟩
⟨#-Pair S T⟩ ::= ⟨S⟩ # ⟨T⟩

Figure 2: The grammar for the type ⟨AList S T⟩, and ⟨#-Pair S T⟩. Here S and T are arbitrary types, and ⟨List U⟩ denotes the usual type of lists of some type (U).

Note that ⟨Value⟩ is the type of all Oz values [DKS06, Section 2].

Figure Figure 3 on the following page shows the tests from our test file IsAListTest.oz.

Put your code in a file IsAList.oz and test using our tests.

14. (10 points) [UseModels]

    In Oz, write a function

    ```
    Invert: <fun {$ <AList S T>}: <AList T S> >
    ```

```
% $Id: IsAListTest.oz,v 1.1 2011/10/06 02:14:35 leavens Exp $
\insert 'IsAList.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'IsAList $Revision: 1.1 $'}
{Test {IsAList nil} '==' true}
{Test {IsAList [a#1 b#2 c#3 a#4 b#5]} '==' true}
{Test {IsAList [a#1 a#4 a#1]} '==' true}
{Test {IsAList fun {$ X} X end} '==' false}
{Test {IsAList not_an_alist} '==' false}
{Test {IsAList 3} '==' false}
{Test {IsAList a#b} '==' false}
{Test {IsAList [a]} '==' false}
{Test {IsAList (x#24)|(y#25)|(z#26)|nil} '==' true}
{Test {IsAList (x#24)|(y#25)|oops|(z#26)|nil} '==' false}
{Test {IsAList (x#24)|(y#25)|(z#26)} '==' false}
{Test {IsAList [i#fun {$ X} X end k4#fun {$ X} 4 end]} '==' true}
{Test {IsAList [b#true c#3 d#"four"]} '==' true}
local BigAList = {fun {$ Len}
                    for I in 1 .. Len collect: C do {C I#[I I+1]} end
                 end
                 10000}
in
   {Test {IsAList BigAList} '==' true}
   {Test {IsAList atom|BigAList} '==' false}
end
{DoneTesting}
```

Figure 3: Tests for 13.

that takes an <AList S T> and returns an <AList T S> with the same number of elements, but with each ⟨#-Pair⟩ being reversed. (See Figure 2 on the previous page for the grammar of ALists.) Thus, if one thinks of an <AList S T> as a binary relation between values of type S and T, the result is the mathematical inverse (or transpose) of the relation. See Figure 4 for examples from our test file InvertTest.oz.

```
% $Id: InvertTest.oz,v 1.1 2011/10/06 02:14:35 leavens Exp $
\insert 'Invert.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'Invert $Revision: 1.1 $'}
{Test {Invert nil} '==' nil}
{Test {Invert [a#1 b#2 c#3 a#4 b#5]} '==' [1#a 2#b 3#c 4#a 5#b]}
{Test {Invert [a#1 a#4 a#1]} '==' [1#a 4#a 1#a]}
{Test {Invert (x#24)|(y#25)|(z#26)|nil} '==' (24#x)|(25#y)|(26#z)|nil}
{Test {Invert [b#true c#3 d#four]} '==' [true#b 3#c four#d]}
local BigAList = {fun {$ Len}
                    for I in 1 .. Len collect: C do {C I#[I I+1]} end
                 end
                 10000}
   InvertedBigAList = {fun {$ Len}
                          for I in 1 .. Len collect: C do {C [I I+1]#I} end
                       end
                       10000}
in
   {Test {Invert BigAList} '==' InvertedBigAList}
   {Test {Invert InvertedBigAList} '==' BigAList}
   {Test {Invert {Invert BigAList}} '==' BigAList}
end
{DoneTesting}
```

Figure 4: Tests for problem 14.

Put your code in a file Invert.oz and test it using our tests. (As described on the first page, you are to hand in both the code and the output of our tests.)

15. [MapToLanguages]

(a) (5 points) Carefully record the time you spend on the next part of this problem, and write in the answer box on webcourses: (i) programming language you used and (ii) the number of minutes you spent on writing the code for `Invert` itself in your favorite language, and (iii) the number of minutes you spent in writing tests, testing, and debugging your code.

(b) (10 points) Using your favorite programming language, write `Invert`, from problem14 above, and test it. If you are using Java, C++, or C for this problem, then you may use our implementation of Oz's lists. You can find these in the subdirectories `Java`, `C++`, and `C` of the tests zip file that you can get from webcourses (in the zip file attached to problem 1). You can represent the type ⟨#-Pair S T⟩ in Java or C# or C++ by using a simple class with two public fields and a constructor. In C, you can represent the type ⟨#-Pair S T⟩ as a struct (and to work with our lists, it may be best to manipulated pointers to such structs).

Hand in your code, including code for testing, as attachments. Paste the test output into the answer box. Your testing code should be approximately what we did for testing in Oz (and thus should be separated from the code for `Invert`.

16. (15 points) [UseModels]

In Oz, implement the function

```
AppendMap: <fun {$ <List T> <fun {$ T}: <List S> >}: <List S> >
```

that takes a list `Lst` of elements of some type `T`, and a function `F` that takes an element of type `T` and returns a list of some type `S`, and which acts as follows. A call of the form `{AppendMap Lst F}` returns a list that is the concatenation (using `Append`) of all the lists that result from applying `F` to teach of the elements in `Lst`. File `AppendMapTest.oz` contains various examples (see Figure 5).

```
% $Id: AppendMapTest.oz,v 1.5 2011/10/06 02:14:35 leavens Exp $
\insert 'AppendMap.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'AppendMap $Revision: 1.5 $'}
{Test {AppendMap nil fun {$ X} [X+1] end} '==' nil}
{Test {AppendMap [7 6 5 4 7] fun {$ X} [X+1] end} '==' [8 7 6 5 8]}
{Test {AppendMap [7 6 5 4 7] fun {$ X} [X X+1] end} '==' [7 8 6 7 5 6 4 5 7 8]}
{Test {AppendMap [a b c] fun {$ X} if X == b then nil else [X] end end}
 '==' [a c]}
{Test {AppendMap [a b c] fun {$ X} if X \= b then nil else [X] end end}
 '==' [b]}
{Test {AppendMap [a b c] fun {$ X} [X X X] end}
 '==' [a a a b b b c c c]}
{Test {AppendMap [a b c] fun {$ X} [[X] [X]] end}
 '==' [[a] [a] [b] [b] [c] [c]]}
{DoneTesting}
```

Figure 5: Tests for Problem 16.

17. (50 points) [UseModels]

This is a problem about recursion over lists, where the elements are #-pairs. In this problem you will write several functions that operate on the abstract data type, `<BRel S T>`, which is a type of binary relations between types `S` and `T`. We represent this type of binary relations as the type `<List <#-Pair S T»`, that is lists whose elements are #-pairs of elements of type `S` and `T`.

In this problem, we give you a file `BRel.oz` containing some of the code for implementing the type `<BRel S T>`. Your task is to fill in the remaining code, as indicated in the file. Our provided code is available from the Webcourses assignment for this problem and in the zip file for the homework. You need to read the code for the operations we provide to understand it. This code assumes that binary relations are represented by lists of #-pairs of keys and values. The code assumes that a given #-pair occurs only once in a representation's list. The code considers that values of types `S` and `T` can be compared using `==`.

Your task is to write each of the following functions on sets (given with their types below).

```
BRelApply: <fun {$ <BRel S T> <S>}: <List T> >
BRelUnion: <fun {$ <BRel S T> <BRel S T>}: <BRel S T> >
BRelDiff: <fun {$ <BRel S T> <BRel S T>}: <BRel S T> >
BRelCompose: <fun {$ <BRel S T> <BRel T U>}: <BRel S U> >
ProductBRel: <fun {$ <List S> <List T>}: <BRel S T> >
```

All these functions return new binary relations; none modify or mutate their arguments. (This is declarative programming!) The function `BRelApply` is given a `<BRel S T>` and a key `K` of type `S`; it returns a list of all the values (of type `T`) that the relation associates `K` to. (If the relation does not relate `K` to anything, then the result is `nil`.) `BRelUnion` returns the union of its two argument relations, so that every relationship in the two arguments is represented in the result. `BRelDiff` returns the binary relation in the first argument, without any relationships that occur in the second argument. `BRelCompose` returns the composition of the given relations, so that if the first argument relates x to y, and the second relation relates y to z, then the result relates x to z. `ProductBRel` takes two lists and returns the Cartesian product of the two lists; that is, it returns the relation that relates each element of the first list to each element of the second list.

Figure 6 on the next page and Figure 7 on page 11 give tests that use these functions from our file `BRelTest.oz`.

To start solving this problem, download the file `BRel.oz` from Webcourses to your directory. Note that you must keep the name as `BRel.oz`. Then add your own code as indicated in the file. (This file is also included in our testing zip file, so if you have already downloaded that, then you have it already.)

In your solution you may not modify any of the provided functions.

Hint: these are really just a bunch of problems about recursion over flat lists.

Hint: To save yourself time, you should write and test each of your functions one by one. It really will save time to test your code yourself; just trying to run our test cases may be frustrating, because you won't have much idea of what went wrong (due to the way our tests are written, using `Assert`).

After doing your own testing, then run our test cases from `BRelTest.oz`, and turn in your source code in `BRel.oz`. Paste the output of our tests into the answer box on webcourses.

```
% $Id: BRelTest.oz,v 1.1 2011/10/06 02:14:35 leavens Exp $
\insert 'BRel.oz'
\insert 'TestingNoStop.oz'
{System.showInfo ""}
{Show 'Since these tests use Assert, you will only see'}
{Show 'messages about what is being tested and failure messages if tests fail.'}
{StartTesting 'BRelTest $Revision: 1.1 $'}
{StartTesting 'Provided code'}
{Assert {BRelEqual {AsBRel nil} {EmptyBRel}}}
{Assert {BRelEqual {AsBRel [3#c 1#a 2#b 3#c]} {AsBRel [1#a 2#b 3#c]}}}
{Assert {BRelHas {AsBRel [3#c 1#a 2#b 3#c]} 2 b}}
{Assert {Not {BRelHas {AsBRel [3#c 1#a 2#b 3#c]} 2 c}}}
{Assert {Not {BRelHasKey {AsBRel [3#c 1#a 2#b 3#c]} a}}}
{Assert {BRelHasKey {AsBRel [3#c 1#a 2#b 3#c]} 3}}
{Assert {IsBRel {AsBRel [3#c 1#a 2#b 3#c]}}}
{Assert {BRelSubset {EmptyBRel} {AsBRel [3#c 1#a 2#b 3#c]}}}
{Assert {Not {BRelSubset {AsBRel [1#c]} {AsBRel [3#c 1#a 2#b 3#c]}}}}
{StartTesting 'BRelAdd'}
{Assert {BRelEqual {BRelAdd {EmptyBRel} 1 1} {AsBRel [1#1]}}}
{Assert {BRelEqual {BRelAdd {EmptyBRel} c 3} {AsBRel [c#3]}}}
{Assert {BRelEqual {BRelAdd {AsBRel [a#b b#c c#d]} d e}
                   {AsBRel [a#b b#c c#d d#e]}}}
{Assert {BRelEqual {BRelAdd {EmptyBRel} c 3} {AsBRel [c#3]}}}
{Assert {BRelEqual {BRelAdd {BRelAdd {EmptyBRel} 2 funky} 3 town}
                   {AsBRel [2#funky 3#town]}}}
{Assert {BRelEqual {BRelAdd {AsBRel [2#3 1#2]} 9 10}
                   {BRelAdd {BRelAdd {BRelAdd {EmptyBRel} 9  10} 2 3} 1 2}}}

{StartTesting 'BRelApply'}
{Assert {BRelApply {AsBRel [a#b b#c c#d d#e]} c} == [d]}
{Assert {BRelApply {AsBRel [a#b a#c a#d a#g]} a} == [b c d g]}
{Assert {BRelApply {AsBRel [a#c a#d a#g]} a} == [c d g]}
{Assert {BRelApply {AsBRel [3#c 1#a 2#b 3#d]} a} == nil}
{Assert {BRelApply {AsBRel [3#c 1#a 2#b 3#d]} 3} == [c d]}
{Assert {BRelApply {AsBRel [1#a 2#b 3#d]} 3} == [d]}
{Assert {BRelApply {AsBRel [3#c 1#a 2#b 3#d]} 4} == nil}
{Assert {BRelApply {AsBRel [3#c 1#a 2#b 3#d]} 1} == [a]}
{StartTesting 'BRelUnion'}
{Assert {BRelEqual {BRelUnion {EmptyBRel} {EmptyBRel}} {EmptyBRel}}}
{Assert {BRelEqual {BRelUnion {EmptyBRel} {AsBRel [3#c 1#a 2#b 3#d]}}
                    {AsBRel [3#c 1#a 2#b 3#d]}}}
{Assert {BRelEqual {BRelUnion {AsBRel [3#c 1#a 2#b 3#d]}
                              {AsBRel [1#a 3#c 2#b 3#d]}}
                              {AsBRel [3#c 1#a 2#b 3#d]}}}
{Assert {BRelEqual {BRelUnion {AsBRel [3#c 1#a 2#b 3#d]}
                              {AsBRel [7#f 8#g 9#i 10#k]}}
                   {AsBRel [3#c 1#a 2#b 3#d 7#f 8#g 9#i 10#k]}}}
```

Figure 6: Tests for problem 17, part 1 of 2.

```
{StartTesting 'BRelDiff'}
{Assert {BRelEqual {BRelDiff {AsBRel [3#c 1#a 2#b 3#d]} {EmptyBRel}}
                   {AsBRel [3#c 1#a 2#b 3#d]}}}
{Assert {BRelEqual {BRelDiff {AsBRel [3#c 1#a 2#b 3#d]}
                             {AsBRel [3#c 1#a 2#b 3#d]}}
                   {EmptyBRel}}}
{Assert {BRelEqual {BRelDiff {AsBRel [3#c 1#a 2#b 3#d]} {AsBRel [3#d 2#b]}}
                   {AsBRel [3#c 1#a]}}}
{Assert {BRelEqual {BRelDiff {AsBRel [3#c 1#a 2#b 3#d]} {AsBRel [3#e 2#c]}}
                   {AsBRel [3#c 1#a 2#b 3#d]}}}
{Assert {BRelEqual {BRelDiff {AsBRel [1#2 2#3 5#6 3#4 7#8]}
                              {AsBRel [7#8 2#3]}}
                    {AsBRel [1#2 5#6 3#4]}}}
{Assert {BRelEqual {BRelDiff {AsBRel [3#c 1#a 2#b 3#d]}
                              {AsBRel [7#f 8#g 9#i 10#k]}}
                   {AsBRel [3#c 1#a 2#b 3#d]}}}

{StartTesting 'BRelCompose'}
{Assert {BRelEqual {BRelCompose {EmptyBRel} {AsBRel [2#b 3#c]}} {EmptyBRel}}}
{Assert {BRelEqual {BRelCompose {EmptyBRel} {EmptyBRel}} {EmptyBRel}}}
{Assert {BRelEqual {BRelCompose {AsBRel [1#2 2#3]} {AsBRel [2#b 3#c]}}
                   {AsBRel [1#b 2#c]}}}
{Assert {BRelEqual {BRelCompose {AsBRel [1#3 2#3]} {AsBRel [3#b 3#c]}}
                   {AsBRel [1#b 1#c 2#b 2#c]}}}

{StartTesting 'ProductBRel'}
{Assert {BRelEqual {ProductBRel nil [b c]} {EmptyBRel}}}
{Assert {BRelEqual {ProductBRel [b c] nil} {EmptyBRel}}}
{Assert {BRelEqual {ProductBRel [b c] [1 2 3]}
                   {AsBRel [b#1 b#2 b#3 c#1 c#2 c#3]}}}
{Assert {BRelEqual {ProductBRel [1 2 3] [b c]}
                   {AsBRel [1#b 2#b 3#b 1#c 2#c 3#c]}}}
{Assert {BRelEqual {ProductBRel [1] [a]}
                   {AsBRel [1#a]}}}
{DoneTesting}
```

Figure 7: Tests for problem 17, continued, part 2 of 2.

18. (20 points) [UseModels]

This is a problem about the window layouts discussed in the “Following the Grammar” handout, section 5.2.

Write a function

```
ChangeChannel: <fun {$ <WindowLayout> <Atom> <Atom>}: <WindowLayout>
```

that takes a window layout `WL`, two atoms `New` and `Old`, and returns a window layout that is just like `WL` except that all windows whose name field’s value is (== to) `Old` in the argument `WL` are changed to `New` in the result.

You can assume that the input has been constructed according to the grammar. So you should not check for arguments that do not conform to this grammar. However, that we will take points off if you don’t follow the grammar in your solution!

Figure 8 on the next page shows examples.

```
\insert 'ChangeChannel.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'ChangeChannel $Revision: 1.6 $'}
{Test {ChangeChannel vertical(nil) cnn simpsons} '==' vertical(nil)}
{Test {ChangeChannel horizontal(nil) cnn simpsons} '==' horizontal(nil)}
{Test {ChangeChannel window(name: simpsons width: 30 height: 40) cnn simpsons}
                '==' window(name: cnn width: 30 height: 40)}
{Test {ChangeChannel
       horizontal([window(name: simpsons width: 30 height: 40)]) simpsons snl}
  '==' horizontal([window(name: simpsons width: 30 height: 40)])}
{Test {ChangeChannel
       vertical([window(name: snl width: 90 height: 50)
                 window(name: snl width: 180 height: 120)])
       futurama snl}
 '=='  vertical([window(name: futurama width: 90 height: 50)
                window(name: futurama width: 180 height: 120)])}
{Test {ChangeChannel
       horizontal([window(name: cbs width: 30 height: 15)
                   vertical([window(name: cnn width: 89 height: 55)
                             window(name: cbs width: 101 height: 45)])
                   horizontal([window(name: cbs width: 92 height: 150)])])
       dailyshow cbs}
  '==' horizontal([window(name: dailyshow width: 30 height: 15)
                   vertical([window(name: cnn width: 89 height: 55)
                             window(name: dailyshow width: 101 height: 45)])
                   horizontal([window(name: dailyshow width: 92 height: 150)])
                   ])}
 {Test {ChangeChannel
        vertical(
           [vertical([window(name: simpsons width: 30 height: 40)])
            horizontal([horizontal([window(name: news width: 5 height: 5)])])
            horizontal([window(name: simpsons width: 30 height: 15)
                        window(name: futurama width: 89 height: 55)])])
       nbc simpsons}
  '==' vertical(
           [vertical([window(name: nbc width: 30 height: 40)])
            horizontal([horizontal([window(name: news width: 5 height: 5)])])
            horizontal([window(name: nbc width: 30 height: 15)
                        window(name: futurama width: 89 height: 55)])])}
{DoneTesting}
```

Figure 8: Tests for problem 18.

19. (25 points) [UseModels]

This is a problem about the statement and expression grammar from the "Following the Grammar" handout, section 5.5.

Write a function

```
NegateIfs: <fun {$ <Statement>}: <Statement>
```

that takes a statement `Stmt`, and returns a statement that is just like `Stmt` except that all `ifStmt` statements of the form `ifStmt(`E S`)` that occur anywhere within `Stmt` are replaced by `ifStmt(equalsExp(`E `varExp(false))` S`)`. This process occurs recursively for all subparts of `Stmt`, even within E and S. Figure 9 shows various examples.

```
\insert 'NegateIfs.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'NegateIfs $Revision: 1.5 $'}
{Test {NegateIfs expStmt(numExp(3))} '==' expStmt(numExp(3))}
{Test {NegateIfs expStmt(varExp(y))} '==' expStmt(varExp(y))}
{Test {NegateIfs expStmt(equalsExp(varExp(y) varExp(z)))}
 '==' expStmt(equalsExp(varExp(y) varExp(z)))}
{Test {NegateIfs assignStmt(x numExp(3))} '==' assignStmt(x numExp(3))}
{Test {NegateIfs ifStmt(varExp(true) assignStmt(x numExp(3)))}
 '==' ifStmt(equalsExp(varExp(true) varExp(false)) assignStmt(x numExp(3)))}
{Test {NegateIfs expStmt(beginExp(nil numExp(3)))}
 '==' expStmt(beginExp(nil numExp(3)))}
{Test {NegateIfs
       expStmt(beginExp([ifStmt(varExp(true) assignStmt(x numExp(3)))
                         assignStmt(y numExp(4))]
                        varExp(y)))}
 '==' expStmt(beginExp([ifStmt(equalsExp(varExp(true) varExp(false))
                                       assignStmt(x numExp(3)))
                                assignStmt(y numExp(4))]
                               varExp(y)))}
{Test {NegateIfs
       ifStmt(beginExp([ifStmt(varExp(true) assignStmt(x numExp(3)))
                        assignStmt(y numExp(4))]
                       varExp(y))
              assignStmt(q beginExp([ifStmt(varExp(m) expStmt(numExp(7)))]
                                    varExp(m))))}
 '==' ifStmt(equalsExp(beginExp([ifStmt(equalsExp(varExp(true) varExp(false))
                                       assignStmt(x numExp(3)))
                                assignStmt(y numExp(4))]
                               varExp(y))
                      varExp(false))
             assignStmt(q beginExp([ifStmt(equalsExp(varExp(m) varExp(false))
                                          expStmt(numExp(7)))]
                                   varExp(m))))}
{DoneTesting}
```

Figure 9: Tests for Problem 19.

Be sure to use a helping function for expressions, so that your code follows the grammar! We will take points off if your code does not follow the grammar.

Using Libraries and Higher-Order Functions

Material on higher-order functions is found in section 3.6 of the textbook. See also the course's code examples page.

20. [UseModels]

    In Oz, write a function

```
Capitalize: <fun {$ <List <String> >}: <List <String> > >
```

    that takes a list of non-empty strings and returns a list of strings such that {Capitalize Strings} is the same as Strings, but with the first character in each String within Strings changed from lower to upper case.

    You can use the Oz built-in function Char.toUpper to convert a character from lower to upper case. (This function leaves characters that are not lower case characters unchanged.)

    In this problem you will implement Capitalize twice:

    (a) (5 points) by using the **for** loop with collect: in Oz (see the Oz documentation or section 3.6.3 of the text [VH04]), and

    (b) (5 points) by using Oz's built in list function Map. (see the code examples page and also Section 6.3 of "The Oz Base Environment" [DKS06]).

    Name your 2 solutions: CapitalizeFor, CapitalizeMap, and put them both in a file named Capitalize.oz.

    For the **for** loop, be sure to use the form with collect:, as only that form of the **for** loop is an expression.

    Hint: since you can assume that each of the strings in Strings is non-empty, you may find it convenient to use pattern matching in the declarations of the **for** loop and in the function passed to Map.

    You can test each of your solution functions by passing it as an argument to the higher-order procedure CapitalizeTest in the file CapitalizeTest.oz (see Figure 10 on the following page).

    Figure 10 on the next page also shows how to use the procedure CapitalizeTest in a way that will work if you name each of your solutions as indicated, and put them all in a file named Capitalize.oz.

21. (10 points) [UseModels] [Concepts]

    Write a function

```
Curry: <fun {$ <fun {$ S T}: U>}: <fun {$ S}: <fun {$ T}: U> > >
```

    that takes a two-argument function, F, and returns curried version of F. Figure 11 on the following page gives some examples, found in the file CurryTest.oz.

    Hint: Note that a 2-argument function named F is equivalent to **fun** {$ X Y} {F X Y} **end**.

```
% $Id: CapitalizeTest.oz,v 1.5 2011/10/06 02:14:35 leavens Exp $
\insert 'Capitalize.oz'
\insert 'TestingNoStop.oz'
declare
{StartTesting 'CapitalizeTest $Revision: 1.5 $'}
proc {CapitalizeTest CapitalizeFun}
   {TestLOS {CapitalizeFun nil} '==' nil}
   {TestLOS {CapitalizeFun ["the" "computer" "science" "way" "of" "the" "world"]}
    '==' ["The" "Computer" "Science" "Way" "Of" "The" "World"]}
   {TestLOS {CapitalizeFun ["a" "tale" "of" "two" "cities" "by" "charles" "dickens"]}
    '==' ["A" "Tale" "Of" "Two" "Cities" "By" "Charles" "Dickens"]}
   {TestLOS {CapitalizeFun ["z"]} '==' ["Z"]}
   {TestLOS {CapitalizeFun ["hand" "in" "test" "output!"]}
    '==' ["Hand" "In" "Test" "Output!"]}
end

{StartTesting 'Part A'}
{CapitalizeTest CapitalizeFor}
{StartTesting 'Part B'}
{CapitalizeTest CapitalizeMap}
{DoneTesting}
```

Figure 10: Test procedure for Problem 20 and its use.

```
% $Id: CurryTest.oz,v 1.2 2011/10/06 02:14:35 leavens Exp $
\insert 'Curry.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'Curry'}
{Test {{{Curry fun {$ X Y} 2*X*X+3*Y end} 10} 5} '==' 2*10*10+3*5}
{Test {{{Curry fun {$ X Y} X#Y end} 10} 5} '==' 10#5}
{Test {{{Curry Number.'+'} 3} 6} '==' 9}
{Test {{{Curry Number.'+'} 5} 6} '==' 11}
local CA = {Curry Append}
in
   {Test {{CA [1 2 3]} [4 5 6]} '==' [1 2 3 4 5 6]}
   {Test {{CA [a good time]} [was had by all]}
         '==' [a good time was had by all]}
end
{DoneTesting}
```

Figure 11: Examples for problem 21.

22. (5 points) [UseModels] [Concepts]

Define a function

```
SearchForZero: <fun {$ <fun {$ <Int>}:<Int> >}: <Int> >
```

that takes an integer-valued function `F` as an argument, and returns the least natural number `N` such that `{F N} == 0`. (In this problem "natural numbers" means non-negative Ints, i.e., 0, 1, 2, ...) Test the examples below by using the file `SearchForZeroTest.oz` (Figure 12), which inserts the actual examples from the file and `SearchForZeroBodyTest.oz` (Figure 13).

```
% $Id: SearchForZeroTest.oz,v 1.1 2010/02/16 22:09:04 leavens Exp $
\insert 'TestingNoStop.oz'
\insert 'SearchForZero.oz'
\insert 'SearchForZeroBodyTest.oz'
```

Figure 12: The file `SearchForZeroTest.oz`.

```
% $Id: SearchForZeroBodyTest.oz,v 1.3 2011/10/06 02:14:35 leavens Exp $
{StartTesting 'SearchForZeroBodyTest $Revision: 1.3 $'}
{Test {SearchForZero fun {$ X} if X == 3 then 0 else 5 end end} '==' 3}
{Test {SearchForZero fun {$ X} 5*X - 10 end} '==' 2}
{Test {SearchForZero fun {$ N} N*N - 36 end} '==' 6}
{DoneTesting}
```

Figure 13: The file `SearchForZeroBodyTest.oz`.

23. (5 points) [UseModels] [Concepts]

    Without using `SearchForZero`, define a function

```
SearchForFixedPoint: <fun {$ <fun {$ <Int>}: <Int> >}: <Int> >
```

    that takes an integer-valued function `F` and returns the least fixed point of `F` in the non-negative integers. That is, `{SearchForFixedPoint F}` returns the least non-negative integer `N` such that `{F N} == N`.

    Test the examples below by feeding the file `SearchForFixedPointTest.oz` (Figure 14) which inserts the actual examples from the file `SearchForFixedPointBodyTest.oz` (Figure 15).

```
% $Id: SearchForFixedPointTest.oz,v 1.1 2010/02/16 22:08:49 leavens Exp $
\insert 'TestingNoStop.oz'
\insert 'SearchForFixedPoint.oz'
\insert 'SearchForFixedPointBodyTest.oz'
```

Figure 14: The file `SearchForFixedPointTest.oz`.

```
% $Id: SearchForFixedPointBodyTest.oz,v 1.3 2011/10/06 02:14:35 leavens Exp $
{StartTesting 'SearchForFixedPointBodyTest $Revision: 1.3 $'}
{Test {SearchForFixedPoint fun {$ X} X end} '==' 0}
{Test {SearchForFixedPoint fun {$ X} if X == 3 then 3 else 7 end end} '==' 3}
{Test {SearchForFixedPoint fun {$ N} {Nth [8 7 6 5 4 3 2 1 0] N+1} end} '==' 4}
{Test {SearchForFixedPoint fun {$ N} N*N - 42 end} '==' 7}
{DoneTesting}
```

Figure 15: The file `SearchForFixedPointBodyTest.oz`.

24. (20 points) [UseModels] [Concepts]

    Define a curried function SearchForMaker that is a generalization of SearchForZero and SearchForFixedPoint. (The exact type of SearchForMaker is for you to decide.) Put your code in a file SearchForMaker.oz.

    Then write a testing file SearchForMakerTesting.oz that shows how to use your definition of the function SearchForMaker to define both functions SearchForZero and SearchForFixedPoint. Your testing file should continue to runs the tests in both SearchForZeroBodyTest.oz and SearchForFixedPointBodyTest.oz, to test these definitions. Your function SearchForMaker should be able to be instantiated (by passing it a function argument) to produce both of these other functions. That is, you should have in your file SearchForMakerTesting.oz something like the code in Figure 16, where you have to fill in appropriate function arguments for SearchForMaker.

```
\insert 'SearchForMaker.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'SearchForZero'}
SearchForZero = {SearchForMaker fun ... end}
\insert 'SearchForZeroBodyTest.oz'
{StartTesting 'SearchForFixedPoint'}
SearchForFixedPoint = {SearchForMaker fun ... end}
\insert 'SearchForFixedPointBodyTest.oz'
```

Figure 16: Outline of the code for your file SearchForMakerTesting.oz.

    Turn in both your code and the file SearchForMakerTesting.oz that you wrote, as well as the output from running the tests in SearchForMakerTesting.oz.

25. (15 points) [UseModels]

    Using FoldR define

    Count: <**fun** {$ <List T> <T>}: <Int> >

    that, for some type T, takes two arguments: Lst, which is a list of values of type T, and Elem, which is a value of type T. The function you are to write, Count, returns an integer that is equal to the number of times that an element equal to Elem is found in Lst. Use the == operator to tell whether an element of Lst is equal to Elem. See Figure 17 for examples.

```
\insert 'Count.oz'
{StartTesting 'CountTest $Revision: 1.1 $'}
{Test {Count nil 7} '==' 0}
{Test {Count [7 2 1 7 3] 7} '==' 2}
{Test {Count [2 1 7 3] 7} '==' 1}
{Test {Count [a g o o d t i m e] o} '==' 2}
{Test {Count [a g o o d t i m e] e} '==' 1}
{Test {Count [e e e k s a i d m i n e e] e} '==' 5}
{DoneTesting}
```

Figure 17: Tests for Problem 25.

    To properly use FoldR to define your solution, make sure that your code for this problem looks like the following outline. (You can also use helping functions.)

```
fun {Count Lst Elem}
   {FoldR
    ...
```

```
        ...
    }
end
```

The next three problems work with the type "Music," as defined by the following grammar. Note that all the ⟨Int⟩s that occur in a ⟨Music⟩ are guaranteed to be non-negative.

```
⟨Music⟩ ::=
      pitch(⟨Int⟩)
    | chord(⟨List Music⟩)
    | sequence(⟨List Music⟩)
```

26. (10 points) [UseModels] Define a function

```
HighestNote: <fun {$ <Music>}: <Number> >
```

that takes a ⟨Music⟩ and returns the largest ⟨Int⟩ that occurs within it. Note that you can use the built-in Oz function `Max` in your solution, as well as functions such as `Map` and `FoldR` to deal with the lists. For this problem we guarantee that the ⟨Music⟩ arguments passed to `HighestNote` will not contain empty lists.

Do not pass lists directly to `HighestNote`, as that will not follow the grammar! We will take points off if you do not follow the grammar by using separate helping functions (or built-in functions such as `FoldR` or `Map`) to deal with lists.

You can test your definition of `HighestNote` using the code given in `HighestNoteTest.oz` (see Figure 18 on the following page), which uses the examples from the file `HighestNoteBodyTest` (also in the figure). The latter gives some examples. Note that in this problem some of the lists may be empty.

27. (15 points) [UseModels] Define a function

```
Transpose: <fun {$ <Music> <Int>}: <Music> >
```

that takes a music value, `Song`, and a number, `Delta`, and produces a music value that is just like `Song`, but in which each integer has been replaced by that integer plus `Delta`. (This is what musicians call transposition, hence the name.)

There are tests for `Transpose` in two files. The file `TransposeTest.oz` (Figure 19 on the next page) is the driver that you use to run the tests. The file `TransposeBodyTest.oz` (Figure 20 on page 23) contains the actual test cases.

28. (30 points) [Concepts] [UseModels] By generalizing your answers to the above problems, define an Oz function

```
FoldMusic: <fun {$ <Music> <fun {$ <Int>}: T>
                   <fun {$ <List Music>}: T> <fun {$ <List Music>}: T>}: T>
```

that is analogous to `FoldR` for lists. The arguments to `FoldMusic` are a ⟨Music⟩, `Song`, a function `PFun` that works on the ⟨Int⟩ in a `pitch` record, a function `CFun` that works on the ⟨List Music⟩ in a `chord` record, and a function `SFun` that works on the ⟨List Music⟩ in a `sequence` record.

Figure 21 on page 24 has testing code, in `FoldMusicTest.oz`, tests that your definition of `FoldMusic` can be used to define `HighestNote`, and `Transpose`.

```
% $Id: HighestNoteTest.oz,v 1.1 2010/02/17 01:35:57 leavens Exp $
\insert 'TestingNoStop.oz'
\insert 'HighestNote.oz'
\insert 'HighestNoteBodyTest.oz'

% $Id: HighestNoteBodyTest.oz,v 1.5 2011/10/06 02:15:37 leavens Exp leavens $
{StartTesting 'HighestNoteBodyTest $Revision: 1.5 $'}
{Test {HighestNote pitch(3)} '==' 3}
{Test {HighestNote chord([pitch(1) pitch(3) pitch(5) pitch(8)])} '==' 8}
{Test {HighestNote chord([pitch(3) sequence([pitch(3) pitch(5) pitch(8)])])}
 '==' 8}
{Test {HighestNote sequence([pitch(3)
                             chord([pitch(1) pitch(3) pitch(5) pitch(8)])
                             chord([pitch(2) sequence([pitch(1) pitch(3)])])
                             sequence([chord([pitch(5) pitch(9)])
                                       chord([pitch(6) pitch(8)])
                                       pitch(1)])
                            ])}
    '==' 9}
{DoneTesting}
```

Figure 18: Testing for problem 26.

```
% $Id: TransposeTest.oz,v 1.1 2010/02/17 01:57:10 leavens Exp $
\insert 'TestingNoStop.oz'
\insert 'Transpose.oz'
\insert 'TransposeBodyTest.oz'
```

Figure 19: Testing for problem 27.

```
% $Id: TransposeBodyTest.oz,v 1.3 2011/10/06 02:14:35 leavens Exp $
{StartTesting 'TransposeBodyTest $Revision: 1.3 $'}
{Test {Transpose pitch(3) 7} '==' pitch(10)}
{Test {Transpose pitch(10) 5} '==' pitch(15)}
{Test {Transpose chord(nil) ~3} '==' chord(nil)}
{Test {Transpose chord([pitch(1) pitch(5) pitch(8)]) 2}
 '==' chord([pitch(3) pitch(7) pitch(10)])}
{Test {Transpose sequence(nil) ~1} '==' sequence(nil)}
{Test {Transpose sequence([pitch(1) pitch(5) pitch(8)]) 2}
 '==' sequence([pitch(3) pitch(7) pitch(10)])}
{Test {Transpose
       sequence([chord([pitch(1) pitch(5) pitch(8)])
                 chord([pitch(3) pitch(7) pitch(0)])
                 chord([pitch(7) pitch(5) pitch(9)])])
       1}
 '==' sequence([chord([pitch(2) pitch(6) pitch(9)])
                 chord([pitch(4) pitch(8) pitch(1)])
                 chord([pitch(8) pitch(6) pitch(10)])])}
{Test {Transpose
       chord([sequence([chord([pitch(1) pitch(5) pitch(8)])
                        chord([pitch(3) pitch(7) pitch(0)])
                        chord([pitch(7) pitch(5) pitch(9)])])
              sequence([pitch(1) pitch(1)])
              chord([sequence(nil) sequence([pitch(3)])])])
       1}
 '==' chord([sequence([chord([pitch(2) pitch(6) pitch(9)])
                       chord([pitch(4) pitch(8) pitch(1)])
                       chord([pitch(8) pitch(6) pitch(10)])])
             sequence([pitch(2) pitch(2)])
             chord([sequence(nil) sequence([pitch(4)])])])}
{Test {Transpose
       sequence([chord([sequence([chord([pitch(1) pitch(5) pitch(8)])
                                  chord([pitch(3) pitch(7) pitch(0)])
                                  chord([pitch(7) pitch(5) pitch(9)])])
                        sequence([pitch(1) pitch(1)])
                        chord([sequence(nil) sequence([pitch(3)])])])
                 chord([pitch(1) pitch(9)])])
       1}
 '==' sequence([chord([sequence([chord([pitch(2) pitch(6) pitch(9)])
                                 chord([pitch(4) pitch(8) pitch(1)])
                                 chord([pitch(8) pitch(6) pitch(10)])])
                       sequence([pitch(2) pitch(2)])
                       chord([sequence(nil) sequence([pitch(4)])])])
                chord([pitch(2) pitch(10)])])}
{DoneTesting}
```

Figure 20: Body of tests for problem 27.

```
% $Id: FoldMusicTest.oz,v 1.1 2010/02/17 02:36:37 leavens Exp $
\insert 'FoldMusic.oz'
\insert 'TestingNoStop.oz'
declare
fun {HighestNote Song}
   fun {HighestInList LOM} % TYPE: <fun {$ <List Music>}: <Int>>
      {FoldR {Map LOM HighestNote} Max ~1}
   end
in
   {FoldMusic Song fun {$ N} N end HighestInList HighestInList}
end
fun {Transpose Song Delta}
   fun {TransposeList LOM} % TYPE: <fun {$ <List Music>}: <List Music>>
      {Map LOM fun {$ M} {Transpose M Delta} end}
   end
in
   {FoldMusic Song
    fun {$ N} pitch(N+Delta) end
    fun {$ Lst} chord({TransposeList Lst}) end
    fun {$ Lst} sequence({TransposeList Lst}) end}
end
\insert 'HighestNoteBodyTest.oz'
\insert 'TransposeBodyTest.oz'
```

Figure 21: Testing for Problem 28 on page 21.

29. (30 points) [UseModels] [Concepts]

A potentially infinite bag (or PIBag) can be described by a "characteristic function" of type `<fun {$ <Value>}: <Int> >`, that determines the multiplicity of each value in the bag. For example, the function M such that

$$M(x) = x - 7, \text{if } x \text{ is an number and } x > 7$$

is the characteristic function for a potentially infinite bag containing all numbers strictly greater than 7, with 8 having multiplicity 1, 9 having multiplicity 2, 10 occuring 3 times, etc. Allowing the user to construct such a potentially infinite bag from a characteristic function gives them the power to construct potentially infinite bags like the one above, which contains an infinite number of elements. (In this example, the bag contains $i - 7$ copies of all numbers i that are strictly greater than 7.)

Your problem is to implement the following operations for the type `PIBag` of potentially infinite bags. (Hint: think about using a function type as the representation of PIBags.)

1. The function `PIBagSuchThat` takes a characteristic function, F and returns a potentially infinite bag such that each value X is in the resulting PIBag with multiplicity $\{F\ X\}$.
2. The function `PIBagUnion` takes two PIBags, with characteristic functions F and G, and returns a PIBag such that each value X is in the resulting PIBag with multiplicity $\{F\ X\} + \{G\ X\}$.
3. The function `PIBagIntersect` takes two PIBags, with characteristic functions F and G, and returns a PIBag such that each value X is in the resulting PIBag with a multiplicity that is the minimum of $\{F\ X\}$ and $\{G\ X\}$.
4. The function `PIBagMultiplicity` takes a PIBag B and a value X and returns an Int that tells how many times X is in B.
5. The function `PIBagAdd` takes a PIBag B, a value X, and a multiplicity N, and returns a PIBag that contains everything in B plus N more occurrences of X.

Note (hint, hint) that the equations in Figure 22 must hold, for all functions `F` and `G`, elements `X` and `Y` of appropriate types, and ⟨Int⟩s `N`.

```
{PIBagMultiplicity {PIBagUnion {PIBagSuchThat F} {PIBagSuchThat G}} X}
    == {F X} + {G X}
{PIBagMultiplicity {PIBagIntersect {PIBagSuchThat F} {PIBagSuchThat G}} X}
    == {Min {F X} {G X}}
{PIBagMultiplicity {PIBagSuchThat F} X} == {F X}
{PIBagMultiplicity {PIBagAdd {PIBagSuchThat F} Y N} X}
    == if X == Y then {F Y} + N else {F X} end
```

Figure 22: Equations that give hints for problem 29.

As examples, consider the tests in Figure 23 on the following page.

```
% $Id: PIBagTest.oz,v 1.4 2011/10/06 02:14:35 leavens Exp $
\insert 'PIBag.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'PIBagTest $Revision: 1.4 $'}
declare
fun {Cokes X} if X == coke then 6 else 0 end end
fun {Beers X} if X == beer then 12 else 0 end end
fun {GTMaker Y} fun {$ X} if {IsInt X} andthen X > Y then X else 0 end end end
GT5 = {GTMaker 5}
GT7 = {GTMaker 7}

{Test {PIBagMultiplicity {PIBagSuchThat Cokes} coke} '==' 6}
{Test {PIBagMultiplicity {PIBagSuchThat Cokes} pepsi} '==' 0}
{Test {PIBagMultiplicity {PIBagAdd {PIBagSuchThat Cokes} pepsi 2} coke} '==' 6}
{Test {PIBagMultiplicity {PIBagAdd {PIBagSuchThat Cokes} pepsi 2} pepsi} '==' 2}
{Test {PIBagMultiplicity {PIBagAdd {PIBagSuchThat Cokes} pepsi 2} sprite} '==' 0}
{Test {PIBagMultiplicity {PIBagUnion {PIBagSuchThat Cokes} {PIBagSuchThat Beers}}
       pepsi} '==' 0}
{Test {PIBagMultiplicity {PIBagUnion {PIBagSuchThat Cokes} {PIBagSuchThat Beers}}
       coke} '==' 6}
{Test {PIBagMultiplicity {PIBagUnion {PIBagSuchThat Cokes} {PIBagSuchThat Beers}}
       beer} '==' 12}
{Test {PIBagMultiplicity
         {PIBagIntersect {PIBagSuchThat Cokes} {PIBagSuchThat Beers}}
       coke} '==' 0}
{Test {PIBagMultiplicity {PIBagSuchThat GT5} coke} '==' 0}
{Test {PIBagMultiplicity {PIBagSuchThat GT7} coke} '==' 0}
{Test {PIBagMultiplicity {PIBagSuchThat GT7} 8} '==' 8}
{Test {PIBagMultiplicity {PIBagSuchThat GT7} 7} '==' 0}
{Test {PIBagMultiplicity {PIBagSuchThat GT7} 6} '==' 0}
{Test {PIBagMultiplicity {PIBagSuchThat GT7} 999092384084184} '==' 999092384084184}
{Test {PIBagMultiplicity {PIBagSuchThat GT5} 999092384084184} '==' 999092384084184}
{Test {PIBagMultiplicity {PIBagUnion {PIBagSuchThat GT5} {PIBagSuchThat GT7}} 6}
 '==' 6}
{Test {PIBagMultiplicity {PIBagUnion {PIBagSuchThat GT5} {PIBagSuchThat GT5}} 6}
 '==' 12}
{Test {PIBagMultiplicity {PIBagIntersect {PIBagSuchThat GT5} {PIBagSuchThat GT7}} 6}
 '==' 0}
{Test {PIBagMultiplicity {PIBagAdd {PIBagSuchThat GT5} 10 3} 10} '==' 13}
{DoneTesting}
```

Figure 23: Example tests for problem 29.

30. (25 points) [Concepts] [UseModels]

Consider the following data grammars.

```
<Exp> ::= boolLit( <Bool> )
      | intLit( <Int> )
      | charLit( <Char> )
      | subExp( <Exp> <Exp> )
      | equalExp( <Exp> <Exp> )
      | andExp( <Exp> <Exp> )
      | ifExp( <Exp> <Exp> <Exp> )
<OType> ::= obool | oint | ochar | owrong
```

In the grammar for expressions, `<Exp>`, the `boolLit`, `intLit`, and `charLit` records represent Boolean, Integer, and Character literals (respectively). As the grammar says, you can assume that inside `boolLit` is a `<Bool>`, and inside an `intLit` is an `<Int>`, and similarly for `charLit`. Records of the form `subExp(`E_1 E_2`)` represent subtractions ($E_1 - E_2$). Records of the form `equalExp(`E_1 E_2`)` represent equality tests, i.e., E_1 `==` E_2. Records of the form `andExp(`E_1 E_2`)` represent conjunctions, i.e., E_1 **andthen** E_2. Records of the form `ifExp(`E_1 E_2 E_3`)` represent if-then-else expressions, i.e., **if** E_1 **then** E_2 **else** E_3 **end**.

In the grammar for types, `<OType>`, the type `obool` is the type of the Booleans, `oint` is the type of the integers, and `ochar` is the type of the characters. The type `owrong` is used for the type of expressions that contain a type error.

Your task is to write a function

```
TypeOf: <fun {$ <Exp>}: OType>
```

that takes an `<Exp>` and returns its `OType`. The file `TypeOfTest.oz` (see Figure 24 on the next page) gives some examples and should be used for testing.

Your function should incorporate a reasonable notion of what the exact type rules are, but your rules should agree with our test cases in Figure 24 on the following page. (Exactly what "reasonable" is left up to you; explain any decisions you feel the need to make. However, note that this is static type checking, you will not be executing the programs and should not look at the values of subexpressions when deciding on types.)

The answer should not suppress `owrong` in any subexpression; that is, if a subexpression is wrong, the whole expression that contains it is wrong.

Points

This homework's total points: 391. Total extra credit points: 0.

References

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

[Lea07] Gary T. Leavens. Following the grammar. Technical Report CS-TR-07-10b, School of EECS, University of Central Florida, Orlando, FL, 32816-2362, November 2007.

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

```
% $Id: TypeOfTest.oz,v 1.5 2011/10/06 02:14:35 leavens Exp $
\insert 'TypeOf.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'TypeOfTest $Revision: 1.5 $'}
{Test {TypeOf boolLit(true)} '==' obool}
{Test {TypeOf boolLit(false)} '==' obool}
{Test {TypeOf intLit(4020)} '==' oint}
{Test {TypeOf charLit(&c)} '==' ochar}
{Test {TypeOf subExp(intLit(3) intLit(4))} '==' oint}
{Test {TypeOf subExp(subExp(intLit(3) intLit(4))
                     subExp(intLit(7) intLit(8)))} '==' oint}
{Test {TypeOf subExp(charLit(&a) intLit(4))} '==' owrong}
{Test {TypeOf subExp(intLit(4) charLit(&a))} '==' owrong}
{Test {TypeOf subExp(intLit(4) boolLit(true))} '==' owrong}
{Test {TypeOf subExp(boolLit(true) intLit(4))} '==' owrong}
{Test {TypeOf equalExp(intLit(3) intLit(4))} '==' obool}
{Test {TypeOf equalExp(charLit(&a) intLit(&b))} '==' owrong}
{Test {TypeOf equalExp(boolLit(true) boolLit(false))} '==' obool}
{Test {TypeOf equalExp(subExp(intLit(5) intLit(3)) intLit(4))} '==' obool}
{Test {TypeOf andExp(boolLit(true) boolLit(false))} '==' obool}
{Test {TypeOf andExp(ifExp(boolLit(true) boolLit(false) boolLit(true))
                    boolLit(false))} '==' obool}
{Test {TypeOf ifExp(boolLit(true) intLit(5) intLit(3))} '==' oint}
{Test {TypeOf ifExp(boolLit(false) boolLit(false) intLit(3))} '==' owrong}
{Test {TypeOf ifExp(boolLit(true) intLit(7) charLit(&c))} '==' owrong}
{Test {TypeOf equalExp(subExp(charLit(&a) intLit(3))
                       intLit(4))} '==' owrong}
{Test {TypeOf equalExp(ifExp(subExp(charLit(&a) intLit(&b))
                             boolLit(false)
                             intLit(4))
                       ifExp(boolLit(true) intLit(3) intLit(4)))}
 '==' owrong}
{Test {TypeOf ifExp(boolLit(true) intLit(4) intLit(5))} '==' oint}
{Test {TypeOf ifExp(boolLit(true) intLit(4) boolLit(true))} '==' owrong}
{Test {TypeOf ifExp(intLit(3) intLit(4) intLit(5))} '==' owrong}
{Test {TypeOf equalExp(subExp(charLit(&a) intLit(3))
                       ifExp(intLit(0) intLit(4) boolLit(true)))}
 '==' owrong}
{Test {TypeOf equalExp(subExp(charLit(&a) charLit(&b))
                       ifExp(boolLit(false)
                             ifExp(andExp(boolLit(true) boolLit(false))
                                   intLit(4)
                                   boolLit(false))
                             boolLit(true)))}
 '==' owrong}
{Test {TypeOf equalExp(equalExp(subExp(intLit(7) intLit(6))
                                subExp(intLit(5) intLit(4)))
                       ifExp(equalExp(intLit(3) intLit(3))
                             ifExp(boolLit(true)
                                   boolLit(true)
                                   boolLit(false))
                             equalExp(charLit(&y) charLit(&y))))}
 '==' obool}
{DoneTesting}
```

Figure 24: Examples for problem 30.