Homework 3: Declarative Programming

See Webcourses and the syllabus for due dates. Hint: don't start these problems at the last minute!

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

Your code should be written in the declarative model, so do not use cells and assignment in your Oz solutions. (Furthermore, note that the declarative model does *not* include the primitive `IsDet` or the library function `IsFree`; thus you are also prohibited from using either of these functions in your solutions.) But please use all linguistic abstractions and syntactic sugars that are helpful.

Feel free to use helping functions as well. Unless we specifically say how you are to solve a problem, feel free to use any functions from the Oz library (base environment), especially functions like `Map` and `FoldR`.

For all programing tasks, 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 Webcourses or from the course resources web page. Turn in (on Webcourses) your code and the output of your testing for all questions that require code.

If the tests don't pass, please try to say why they don't pass, as this enhances communication and makes commenting on the code easier and more specific to your problem.

Our tests use the functions in the course library's `TestingNoStop.oz`. The `Test` procedure in this file can be passed an actual value, a connective (which is used only in printing), and an expected value, as in the following statement.

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

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

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

Calls to `Assert` produce no output unless they are passed the argument **`false`**. Note that you would not use `Browse` or `Show` around a call to `Test` or `Assert`. If you're not sure how to use our testing code, ask us for help.

Turn in (on Webcourses) your both code and output of your testing for all problems that require code. Please upload code as text files with the name given in the problem or testing file and with the suffix `.oz`. Please use the name of the main function as the name of the file. Please upload test output and English answers as plain text files with suffix `.txt` or as entries in the webcourses answer box. If you have a mix of code and English, use a text file with a `.oz` file suffix, and put comments in the file for the English parts. (In any case, don't put any spaces in your file names!)

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

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" 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]

   Why is declarative programming useful?

2. (5 points) [Concepts] [MapToLanguages]

   Can you write declarative programs in C, C++, or Java? (Answer "yes" or "no" and briefly explain.)

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

3. (5 points) [UseModels]

   Write an iterative function

   ```
   FindIndex: <fun {$ <List <Atom>> <Atom>}: <Int>>
   ```

   that takes a list of atoms `LoA` and an atom `A`, and returns the index of the first occurrence of `A` in `LoA`, or `~1` if `A` does not occur in `LoA`. The following are examples that you can find in our test file `FindIndexTest.oz`.

   ```
   {Test {FindIndex a|b|c|d|a|nil a} '==' 1}
   {Test {FindIndex a|b|c|d|a|nil b} '==' 2}
   {Test {FindIndex a|b|c|d|a|nil c} '==' 3}
   {Test {FindIndex a|b|c|d|a|nil d} '==' 4}
   {Test {FindIndex a|nil e} '==' ~1}
   {Test {FindIndex nil hmmm} '==' ~1}
   {Test {FindIndex [now is the time 'for' change] 'for'} '==' 5}
   {Test {FindIndex [now is the time 'for' change] stasis} '==' ~1}
   {Test {FindIndex [the code examples page gives access to code examples
                     related to subjects covered 'in' 'COP' '4020' 'at' 'UCF'
                     as taught by 'Gary' 'T.' 'Leavens'] 'Gary'}
    '==' 22}
   ```

   Your code must have iterative behavior. (Hint: use tail recursion!)

   Put your code in a file `FindIndex.oz`. After doing your own testing, run our tests in `FindIndexTest.oz`.

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

4. (5 points) [Concepts] Give an example Oz expression, other than `leaf`, that defines a value in the type ⟨BTree Int⟩. Recall that a ⟨Literal⟩ can be an atom such as `a` or `'true'`.

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

5. (8 points) [UseModels]

   For each of the functions in Figure 1 on the following page, say whether (i) the function has a correct outline that follows the grammar for (finite) flat lists, or (ii) if it doesn't, then briefly explain what the problem is with that function (i.e., why it does not follow the outline for the flat list grammar).

   (Note: you don't have to judge whether these are correct or not, and you aren't expected to run them.)

6. (8 points) [UseModels]

   For each of the functions in Figure 2 on page 4, say whether (i) the function has a correct outline that follows the grammar for (finite) flat lists, or (ii) if it doesn't, then briefly explain what the problem is with that function (i.e., why it does not follow the outline for the flat list grammar).

1.
```
fun {TalentsOf People}
   case People of
      P|Ps then {Talent P}|{TalentsOf Ps}
   end
end
```

2.
```
fun {TalentsOf People}
   case People of
      nil then nil
   end
end
```

3.
```
fun {TalentsOf People}
   case People of
      P|Ps then {Talent P}|{TalentsOf Ps}
   else nil
   end
end
```

4.
```
fun {TalentsOf People}
   case People of
      hot then sweltering
   [] warm then happy
   [] cold then freezing
   end
end
```

5.
```
fun {TalentsOf People}
   if People == 0
   then {Talent People.1} + {TalentsOf People.2}
   else 0
   end
end
```

Figure 1: Problem 5.

1.
```
fun {RhymesWith Words Sought}
   case Words of
      orange then nil
   [] moon then [june croon swoon]
   [] love then [dove glove guv]
   end
end
```

2.
```
fun {RhymesWith Words Sought}
   case Words of
      W|Ws then
              {Append
               if {Not {Rhymes Sought W}} then nil else [W] end
               {RhymesWith Ws Sought}
              }
   else nil
   end
end
```

3.
```
fun {RhymesWith Words Sought}
   case Words of
      W|Ws then
              {Append
               if {Not {Rhymes Sought W}} then nil else [W] end
               {RhymesWith Ws Sought}
              }
   end
end
```

4.
```
fun {RhymesWith Words Sought}
   case Words of
      W|Ws then if {Not {Rhymes Sought W}}
                then {RhymesWith Ws Sought}
                else W|{RhymesWith Ws Sought}
                end
   else nil
   end
end
```

5.
```
fun {RhymesWith Words Sought}
   case Words of
      W|Ws andthen {Rhymes Sought W}
           then W|{RhymesWith Ws Sought}
   else {RhymesWith Ws Sought}
   end
end
```

Figure 2: Problem 6.

(Note: you don't have to judge whether these are correct or not, and you aren't expected to run them.)

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.

7. (5 points) [Concepts] [UseModels]

   In the declarative programming model, how can a function make a change to a data structure (for example, delete a key from a tree)?

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

8. [Concepts] [MapToLanguages]

   (a) (3 points) How does the textbook advocate using the kernel language to calculate the execution time and space used by an Oz program? (Give a brief answer without going into details.)

   (b) (2 points) Could same approach be used in Java with the Java Virtual Machine's bytecode language playing the role of Oz's kernel language? (Answer "yes" or "no".)

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

9. [Concepts] [MapToLanguages]

   (a) (3 points) Briefly describe what the `Filter` function is useful for.

   (b) (2 points) In object-oriented programming, one can make an object with one method that is treated as a function argument by other methods. Such objects are called *strategy objects* in the Strategy design pattern [GHJV95]. Using this idea, is it possible to write a method in Java or C++ that acts like `Filter` in Oz? (Answer "yes" or "no".)

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

10. (5 points) [Concepts] [MapToLanguages]

    How does one hide the internal representation of an abstract datatype in 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.

11. (5 points) [Concepts]

    Name one kind of standard programming language capability that is not supported by the declarative programming model, and that is useful in interfacing with the physical world.

Read section 3.9 of the textbook and answer the following questions.

12. (5 points) [Concepts] [MapToLanguages]

    In what way is a module in Oz like a class in C++ or Java?

Regular Problems

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

Iteration

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

13. (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`.

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.

14. (10 points) [UseModels]

    Write a function

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

    that a list of items of some type T, and an item of type T and returns a list just like the argument list, but with the each occurrence of the item (if any) removed. Use == to compare the item and the list elements. The following examples are written using the Test procedure from the course library. They are also found in our testing file DeleteAllTest.oz.

    ```
    % $Id: DeleteAllTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
    \insert 'TestingNoStop.oz'
    \insert 'DeleteAll.oz'
    {StartTesting 'DeleteAll'}
    {Test {DeleteAll nil 3} '==' nil}
    {Test {DeleteAll [1 1 2 3 2 1 2 3 2 1] 1} '==' [2 3 2 2 3 2]}
    {Test {DeleteAll [1 2 3 2 1 2 3 2 1] 1} '==' [2 3 2 2 3 2]}
    {Test {DeleteAll [1 1 2 3 2 1 2 3 2 1] 4} '==' [1 1 2 3 2 1 2 3 2 1]}
    {Test {DeleteAll [99 56 3] 3} '==' [99 56]}
    {StartTesting done}
    ```

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

15. (10 points) [UseModels]

    Write a function

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

    that takes a list of items of some type T and an item of type T, and returns a list just like the argument list, but with the second occurrence of the item (if any) removed. The following is our test file DeleteSecondTest.oz.

    ```
    % $Id: DeleteSecondTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
    \insert 'TestingNoStop.oz'
    \insert 'DeleteSecond.oz'
    {StartTesting 'DeleteSecond'}
    {Test {DeleteSecond nil 3} '==' nil}
    {Test {DeleteSecond [1 2 3 2 1 2 3 2 1] 1} '==' [1 2 3 2 2 3 2 1]}
    {Test {DeleteSecond [1 2 3 2 1 2 3 2 1] 4} '==' [1 2 3 2 1 2 3 2 1]}
    {Test {DeleteSecond [1 2 3] 3} '==' [1 2 3]}
    {Test {DeleteSecond [3 1 2 3] 3} '==' [3 1 2]}
    {StartTesting done}
    ```

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

    Hint: you may need a helping function.

16. (30 points) [UseModels]

This is a problem about recursion over flat lists. In this problem you will write several functions that operate on an abstract data type, <Set T> represented as the type <List T>, that is lists whose elements have type T. (In contrast to a later problem, in this problem, we will only consider finite sets.)

In this problem, we give you some of the code for implementing sets using lists, and ask you to fill in the remaining code. Our code is available from the Webcourses assignment for this problem. You need to read the code for the operations we provide to understand it. This code assumes that lists are represented *without* duplicate elements. The code considers that X is a duplicate of Y if and only if $X == Y$.

There is one other complication in the code that we have provided for you. This is that the functions won't work until you write some parts of your own code. In particular, our code for the function named AsSet uses the function Add, which you are to write; so AsSet won't work and can't be tested until you write a definition for Add.

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

```
Add: <fun {$ <Set T> T>}: <Set T>>
Remove: <fun {$ <Set T> T>}: <Set T>>
Union: <fun {$ <Set T> <Set T>>}: <Set T>>
Minus: <fun {$ <Set T> <Set T>>}: <Set T>>
Intersect: <fun {$ <Set T> <Set T>>}: <Set T>>
UnionList: <fun {$ <List <Set T>>}: <Set T>>
```

All these functions return new sets, none modify or mutate their arguments. (This is functional programming!) The function Add inserts an item into the set argument, returning a new set containing just the elements of the set argument and the item. Remove takes an item out of a set (or returns its set argument unchanged if the element argument was not in the set argument). Union returns the union of its two arguments as a set (i.e., without duplicates). Minus returns the set of all elements such that every element of the result is an element of the first set argument, but no element of the result is an element of the second set argument. Intersect returns the set of elements that are elements of both set arguments. UnionList returns the union of all the sets in its argument list.

Figure 3 on the following page gives tests that uses these functions.

To start solving this problem, download the file SetOps.oz from Webcourses to your directory. Note that you must keep the name as SetOps.oz. Then add your own code as indicated in the file. (This code 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 list recursion problems.

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 will be frustrating, because you won't have much idea of what went wrong (due to the way our tests are written, using Assert).

Hint: to incrementally develop the procedures, start by implementing Add. It may be helpful to "stub out" the other functions.

After doing your own testing, then run our test cases from SetOpsTest.oz, and turn in your source code in SetOps.oz and the output of our tests (as well as the output from any of your own tests).

```
% $Id: SetOpsTest.oz,v 1.2 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'SetOps.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'SetOps'}
{Assert {Equal {AsSet nil} {EmptySet}}}
{Assert {Equal {AsSet [1 2 3]} {AsSet [3 1 2]}}}
{Assert {Not {Equal {AsSet [1 2 3]} {AsSet [1 2]}}}}
{Assert {Not {Equal {AsSet [c b]} {AsSet [a b c]}}}}
{StartTesting 'Add'}
{Assert {Equal {Add {EmptySet} 1} {AsSet [1]}}}
{Assert {Equal {Add {AsSet [2 3]} 1} {AsSet [1 2 3]}}}
{Assert {Equal {Add {AsSet [2 3 1]} 1} {AsSet [2 1 3]}}}
{StartTesting 'Remove'}
{Assert {Equal {Remove {EmptySet} 7} {EmptySet}}}
{Assert {Equal {Remove {AsSet [2 3 1]} 1} {AsSet [3 2]}}}
{Assert {Equal {Remove {AsSet [2 3 1 5 7 4]} 5} {AsSet [3 2 1 7 4]}}}
{Assert {Equal {Remove {AsSet [2 3 4 8]} 1} {AsSet [2 3 4 8]}}}
{StartTesting 'Union'}
{Assert {Equal {Union {EmptySet} {AsSet [d e]}} {AsSet [d e]}}}
{Assert {Equal {Union {AsSet [a b c]} {EmptySet}} {AsSet [a b c]}}}
{Assert {Equal {Union {AsSet [a b c]} {AsSet [d e]}} {AsSet [a b c d e]}}}
{Assert {Equal {Union {AsSet [e a b c]} {AsSet [c d e a]}}
         {AsSet [a b c d e]}}}
{StartTesting 'Minus'}
{Assert {Equal {Minus {EmptySet} {AsSet [d e]}} {EmptySet}}}
{Assert {Equal {Minus {AsSet [d e]} {EmptySet}} {AsSet [d e]}}}
{Assert {Equal {Minus {AsSet [a b c]} {AsSet [d e]}} {AsSet [a b c]}}}
{Assert {Equal {Minus {AsSet [e a b c]} {AsSet [c d a e]}} {AsSet [b]}}}
{Assert {Equal {Minus {AsSet [e a b c]} {AsSet [c e d a f]}} {AsSet [b]}}}
{Assert {Equal {Minus {AsSet [a b]} {AsSet [b a]}} {AsSet nil}}}
{StartTesting 'Intersect'}
{Assert {Equal {Intersect {EmptySet} {AsSet [d e]}} {EmptySet}}}
{Assert {Equal {Intersect {AsSet [a b c]} {AsSet [d e]}} {EmptySet}}}
{Assert {Equal
         {Intersect {AsSet [e a b c]} {AsSet [c d a e]}} {AsSet [a e c]}}}
{Assert {Equal
         {Intersect {AsSet [e a b c]} {AsSet [c e d a f b]}}
         {AsSet [c b a e]}}}
{Assert {Equal {Intersect {AsSet [a b]} {AsSet [b a]}} {AsSet [a b]}}}
{StartTesting 'UnionList'}
{Assert {Equal {UnionList nil} {EmptySet}}}
{Assert {Equal
         {UnionList [{AsSet [a b c]} {AsSet nil} {AsSet [d e]}]}
         {AsSet [a b c d e]}}}
{Assert {Equal
         {UnionList [{AsSet [a]} {AsSet [b c]} {EmptySet} {AsSet [d e]}
                     {AsSet [f g h i j]} {AsSet [k l m a b e]}]}
         {AsSet [a b c d e f g h i j k l m]}}}
{StartTesting 'done'}
```

Figure 3: Tests for problem 16.

17. (20 points) [UseModels]

This is a problem about the window layouts discussed in section 5.2 of the "Following the Grammar" handout.

Write a function

```
ShrinkTo: <fun {$ <WindowLayout> <Number> <Number>}: <WindowLayout>>
```

such that `{ShrinkTo WL Width Height}` returns a window layout that is just like `WL`, except that each window in `WL` is made to have width W and height H, where W is the minimum of the window's current width and the `Width` parameter, and H is the minimum of the window's current height and the `Height` parameter.

You can assume that the input window layout has been constructed according to the grammar. That is, you don't have to check for errors in the input.

Figure 4 has some examples that are written using the `Test` procedure from the course library. Turn in your source code in a file `ShrinkTo.oz`, and output of testing that includes the tests in `ShrinkToTest.oz`.

```
% $Id: ShrinkToTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'ShrinkTo.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'ShrinkTo'}
{Test {ShrinkTo vertical(nil) 10 39} '==' vertical(nil)}
{Test {ShrinkTo horizontal(nil) 10 39} '==' horizontal(nil)}
{Test {ShrinkTo window(name: simpsons width: 30 height: 40) 10 39}
          '==' window(name: simpsons width: 10 height: 39)}
{Test {ShrinkTo window(name: simpsons width: 30 height: 11) 10 39}
          '==' window(name: simpsons width: 10 height: 11)}
{Test {ShrinkTo window(name: familyGuy width: 30 height: 11) 80 39}
          '==' window(name: familyGuy width: 30 height: 11)}
{Test {ShrinkTo window(name: familyGuy width: 30 height: 11) 80 5}
          '==' window(name: familyGuy width: 30 height: 5)}
{Test {ShrinkTo
       horizontal([window(name: familyGuy width: 30 height: 15)
                   window(name: futurama width: 89 height: 55)])
       20 30}
  '==' horizontal([window(name: familyGuy width: 20 height: 15)
                   window(name: futurama width: 20 height: 30)])}
 {Test {ShrinkTo
        vertical(
           [vertical([window(name: simpsons width: 30 height: 40)])
            horizontal([horizontal([window(name: news width: 5 height: 5)])])
            horizontal([window(name: familyGuy width: 30 height: 15)
                        window(name: futurama width: 89 height: 55)])])
       20 30}
  '==' vertical(
           [vertical([window(name: simpsons width: 20 height: 30)])
            horizontal([horizontal([window(name: news width: 5 height: 5)])])
            horizontal([window(name: familyGuy width: 20 height: 15)
                        window(name: futurama width: 20 height: 30)])])}
{StartTesting done}
```

Figure 4: Tests for problem 17.

18. (30 points) [UseModels]

    This is a problem about the Boolean expression grammar discussed in the "Following the Grammar" handout, section 5.4.

    Write a function

    ```
    BEval : <fun {$ <Bexp> <fun {$ <Atom>}: <Bool>>}: <Bool>>
    ```

    that takes 2 arguments: a `Bexp`, E, and a function from atoms to Booleans, F. Assume that F is defined on each atom that occurs in a $\langle$Varref$\rangle$. This function evaluates the expression E, using F to determine the values of all $\langle$Varref$\rangle$s that occur within it. Examples are shown in Figure 5 on the following page.

    You can assume that the input Bexp has been constructed according to the grammar. That is, you don't have to check for errors in the input.

    Turn in your source code in a file `BEval.oz`, and output of testing that includes the tests in `BEvalTest.oz`.

19. (35 points) [UseModels]

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

    Write a function

    ```
    AllIds : <fun {$ <Statement>} : <Set Atom>>
    ```

    such that `{AllIds Stmt}` returns a set of all Atoms that are used in the statement as identifiers. Such uses may occur in several places in the grammar, but the only base cases in which Atoms occur as identifiers are: the left side of an assignment statement (e.g., `id` in `assignStmt(id numExp(7))`) and variable reference expressions (e.g., `foo` in `varExp(foo)`).

    For the sets used in the problem, you should use your solution to problem 16 on page 7. (If you don't have a working solution to that problem, you can get a solution from the course staff, at the cost of losing the points for that problem.) Thus your file `AllIds.oz` would start as follows (after initial comments).

    ```
    \insert 'SetOps.oz'
    declare
    fun {AllIds Stmt}
    ```

    Figure 6 on page 12 gives some examples.

    Hint: Be sure to use a helping function, such as `AllIdsExp`, so that your code follows the grammar.

    After doing your own testing, run our tests and turn in the output from your tests and ours.

20. (35 points) [UseModels]

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

    Write a function

    ```
    SubstIdentifier : <fun {$ <Statement> <Atom> <Atom>}: <Statement>>
    ```

    that takes a statement `Stmt` and two atoms, `New` and `Old`, and returns a statement that is just like `Stmt`, except that all occurrences of `Old` in `Stmt` are replaced by `New`. Examples are shown in Figure 7 on page 12. Use that `SubstIdentifierTest.oz` file for your testing.

    Hint: Be sure to use a helping function, such as `SubstIdentifierExp`, so that your code follows the grammar.

```
% $Id: BEValTest.oz,v 1.1 2008/10/17 21:24:42 leavens Exp $
\insert 'BEval.oz'
\insert 'TestingNoStop.oz'
declare
fun {StdEnv A}
   case A of
      p then 1
   [] q then 2
   [] r then 4020
   [] x then 76
   [] y then 0
   else raise stdEnvIsUndefinedOn(A) end
   end
end

% All vars equal in Env2
fun {Env2 A} 1 end

{StartTesting 'BEval'}
{Assert {BEval comp(equals(q q)) StdEnv} == true}
{Assert {BEval comp(notequals(q q)) StdEnv} == false}
{Assert {BEval comp(equals(q r)) StdEnv} == false}
{Assert {BEval comp(equals(q r)) Env2} == true}
{Assert {BEval comp(notequals(p q)) StdEnv} == true}
{Assert {BEval comp(notequals(p q)) Env2} == false}
{Assert {BEval andExp(comp(notequals(p q))
                      comp(equals(x x))) StdEnv} == true}
{Assert {BEval andExp(comp(notequals(p q))
                      comp(equals(x x))) Env2} == false}
{Assert {BEval andExp(comp(notequals(p q))
                      comp(notequals(x x))) StdEnv} == false}
{Assert {BEval andExp(notExp(comp(equals(p p)))
                      comp(equals(x x))) StdEnv} == false}
{Assert {BEval notExp(andExp(notExp(comp(equals(p p)))
                             comp(equals(x x)))) StdEnv} == true}
{Assert {BEval orExp(notExp(andExp(notExp(comp(equals(p p)))
                                   comp(equals(x x))))
                     orExp(comp(equals(p q))
                           comp(equals(x x)))) StdEnv} == true}
{Assert {BEval orExp(andExp(notExp(comp(equals(p p)))
                            comp(equals(x x)))
                     orExp(comp(equals(p q))
                           comp(equals(x y)))) StdEnv} == false}
{StartTesting done}
```

Figure 5: Testing for BEval, problem 18.

```
% $Id: AllIdsTest.oz,v 1.1 2008/10/18 14:29:46 leavens Exp $
\insert 'AllIds.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'AllIds'}
{Assert {Equal {AllIds expStmt(varExp(q))} {AsSet [q]}}}
{Assert {Equal {AllIds expStmt(varExp(r))} {AsSet [r]}}}
{Assert {Equal {AllIds assignStmt(a varExp(b))} {AsSet [a b]}}}
{Assert {Equal {AllIds ifStmt(equalsExp(varExp(id) numExp(0))
                               assignStmt(id varExp(b)))}
         {AsSet [id b]}}}
{Assert {Equal {AllIds expStmt(beginExp(nil varExp(a)))}
         {AsSet [a]}}}
{Assert
 {Equal
  {AllIds
   expStmt(beginExp([ifStmt(equalsExp(varExp(a) numExp(0))
                               assignStmt(v varExp(q)))
                     assignStmt(v2 varExp(w))]
                    beginExp([expStmt(varExp(r))] varExp(a))))}
  {AsSet [a v q v2 w r]}}}
{StartTesting done}
```

Figure 6: Testing for `AllIds`, problem 19.

```
% $Id: SubstIdentifierTest.oz,v 1.1 2008/10/18 14:30:55 leavens Exp $
\insert 'SubstIdentifier.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'SubstIdentifier'}
{Assert {SubstIdentifier expStmt(varExp(q)) p q} == expStmt(varExp(p))}
{Assert {SubstIdentifier expStmt(varExp(r)) p q} == expStmt(varExp(r))}
{Assert {SubstIdentifier assignStmt(a varExp(a)) n a}
                     == assignStmt(n varExp(n))}
{Assert {SubstIdentifier
         ifStmt(equalsExp(varExp(id) numExp(0)) assignStmt(id varExp(b)))
         var id}
      == ifStmt(equalsExp(varExp(var) numExp(0)) assignStmt(var varExp(b)))}
{Assert {SubstIdentifier expStmt(beginExp(nil varExp(a))) n a}
                     == expStmt(beginExp(nil varExp(n)))}
{Assert {SubstIdentifier
         expStmt(beginExp([ifStmt(equalsExp(varExp(a) numExp(0))
                                  assignStmt(a varExp(b)))
                           assignStmt(a varExp(a))
                          ]
                          beginExp(nil varExp(a))))
         n a}
      == expStmt(beginExp([ifStmt(equalsExp(varExp(n) numExp(0))
                                  assignStmt(n varExp(b)))
                           assignStmt(n varExp(n))
                          ]
                          beginExp(nil varExp(n))))}
{StartTesting done}
```

Figure 7: Tests for the function `SubstIdentifier`, which is problem 20.

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.

21. [UseModels]

    In Oz, write a function

    ```
    Associated: <fun {$ <List <Pair Key Value>> Key}: <List Value>
    ```

    such that {Associated Pairs K} is the list, in order, of the second elements of pairs in Pairs, whose first element is equal (by ==) to the argument K.

    Do this:

    (a) (5 points) by writing out the recursion yourself,

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

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

    Name your 3 solutions: AssociatedPartA, AssociatedPartB, and AssociatedPartC.

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

    You can test by passing each of your functions as an argument to the higher-order procedure in Figure 8 on the following page. This is in the file AssociatedTest.oz.

    Figure 8 on the next page also shows how to use the procedure AssociatedTest in a way that will work if you name each of your solutions as indicated, and put them all in a file named Associated.oz.

22. [UseModels]

    This problem is due to Simon Thompson. It works with the database of a library. Consider the following types.

    ```
    <Database> ::= <List <Pair <Person> <Book>>>
    <Pair P B> ::= <P> # <B>
    <Person> ::= <Literal>
    <Book> ::= <Literal>
    ```

    A value of type <Database> records each borrowing by a person of a book.

    (a) (10 points) Write a function Borrowers that takes a <Database> and a <Book> and returns a list of all persons who have borrowed that book.

    (b) (10 points) Write a function Borrowed that takes a <Database> and a <Book> and returns true just when someone has borrowed it.

    (c) (10 points) Write a function NumBorrowed that takes a <Database> and a <Person> and returns the number of book that person has borrowed.

    Figure 9 on the following page gives examples of these. These tests are in the file BorrowedTest.oz (and require you to put all 3 functions in the file Borrowed.oz).

23. (15 points) [UseModels] [Concepts]

    Write a function

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

    that takes a list of functions, and returns a function which is their composition. Figure 10 on page 15 gives some examples. To test using these examples, use the file ComposeTest.oz.

    Hint: note that {Compose nil} is the identity function.

```
% $Id: AssociatedTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'Associated.oz'
\insert 'TestingNoStop.oz'

declare
proc {AssociatedTest Associated}
   {Test {Associated nil 3} '==' nil}
   {Test {Associated [(3#4) (5#7) (3#6) (9#3)] 3} '==' [4 6]}
   {Test {Associated [(1#a) (3#c) (2#b) (4#d)] 2} '==' [b]}
   {Test {Associated [(1#a) (3#c) (2#b) (4#d)] 0} '==' nil}
end

{StartTesting 'Part (a)'}
{AssociatedTest AssociatedPartA}
{StartTesting 'Part (b)'}
{AssociatedTest AssociatedPartB}
{StartTesting 'Part (c)'}
{AssociatedTest AssociatedPartC}
{StartTesting done}
```

Figure 8: Test procedure for Problem 21 and its use.

```
% $Id: BorrowedTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'Borrowed.oz'
\insert 'TestingNoStop.oz'
declare
ExampleBase = [ ('Alice' # 'Tintin') ('Anna' # 'Little Women')
                ('Alice' # 'Asterix') ('Rory' # 'Tintin') ]

{StartTesting 'Borrowers, part (a)'}
{Test {Borrowers ExampleBase 'Tintin'} '==' ['Alice' 'Rory']}
{Test {Borrowers ExampleBase 'Little Women'} '==' ['Anna']}
{Test {Borrowers ExampleBase 'Asterix'} '==' ['Alice']}
{Test {Borrowers ExampleBase 'The Wizard of Oz'} '==' nil}

{StartTesting 'Borrowed, part (b)'}
{Test {Borrowed ExampleBase 'Tintin'} '==' true}
{Test {Borrowed ExampleBase 'Little Women'} '==' true}
{Test {Borrowed ExampleBase 'Asterix'} '==' true}
{Test {Borrowed ExampleBase 'The Wizard of Oz'} '==' false}

{StartTesting 'NumBorrowed, part (c)'}
{Test {NumBorrowed ExampleBase 'Alice'} '==' 2}
{Test {NumBorrowed ExampleBase 'Anna'} '==' 1}
{Test {NumBorrowed ExampleBase 'Rory'} '==' 1}
{Test {NumBorrowed ExampleBase 'Ben'} '==' 0}

{StartTesting done}
```

Figure 9: Examples for problem 22.

```
% $Id: ComposeTest.oz,v 1.2 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'Compose.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'Compose'}
{Test {{Compose nil} [1 2 3]} '==' [1 2 3]}
local
   fun {Tail Ls} _|Rest = Ls in Rest end
in
   {Test {{Compose [Tail]} [1 2 3 4 5]}
         '==' [2 3 4 5]}
   {Test {{Compose [Tail Tail Tail]} [1 2 3 4 5]}
         '==' [4 5]}
   {Test {{Compose [fun {$ H|_} H end Tail Tail Tail]} [a b c d e f]}
         '==' d}
end
{Test {{Compose [fun {$ X} X + 1 end fun {$ X} X + 2 end]} 4}
      '==' 7}
{Test {{Compose [fun {$ X} {Append X [5 6]} end
                 fun {$ X} {Append X [2]} end]} [4]}
      '==' [4 2 5 6]}
{Test {{Compose [fun {$ X} 3|X end fun {$ Y} 4|Y end]} nil}
      '==' 3|(4|nil)}
{StartTesting done}
```

Figure 10: Examples for problem 23.

24. [UseModels] [Concepts]

Consider the following type as a representation of binary relations.

```
<BinaryRel A B> ::= <List <Pair A B>>
<Pair A B> ::= <A> # <B>
```

(a) (10 points) Using the built-in function `All` (see Section 6.3 of "The Oz Base Environment" [DKS06]), write a function

```
IsFunction: <fun {$ <BinaryRel A B>}: Bool>
```

that returns **true** just when its argument satisfies the standard definition of a function; that is, `{IsFunction R}` is **true** just when for each pair $x\#y$ in the list `R` there is no pair $x\#z$ in `R` such that $y \neq z$. The following examples are in the file `IsFunctionTest.oz`.

```
% $Id: IsFunctionTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'IsFunction.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'IsFunction'}
{Test {IsFunction nil} '==' true}
{Test {IsFunction [a#1 b#2 c#3 a#1]} '==' true}
{Test {IsFunction [d#4 a#1 b#2 c#3 a#1]} '==' true}
{Test {IsFunction [b#2 c#3 a#1]} '==' true}
{Test {IsFunction [b#2 c#3 b#41 a#1]} '==' false}
{Test {IsFunction [d#4 b#2 c#3 b#41 a#1]} '==' false}
{Test {IsFunction [b#2 c#3 d#2 e#2 f#2 g#3 a#1]} '==' true}
{Test {IsFunction [bush#shrub]} '==' true}
{Test {IsFunction [tree#arb bush#shrub]} '==' true}
{Test {IsFunction [tree#arb bush#hmmm bush#shrub]} '==' false}
{Test {IsFunction [plant#grow tree#arb bush#hmmm bush#shrub]} '==' false}
{StartTesting done}
```

(b) (10 points) Using the **for** loop in Oz (see the Oz documentation or section 3.6.3 of the text [VH04]), write a function

```
BRelCompose: <fun {$ <BinaryRel A B> <BinaryRel B C>}:
                    <BinaryRel A C>>
```

that returns the relational composition of its arguments. That is, a pair $x\#z$ is in the result if and only if there is a pair $x\#y$ in the first relation argument of the pair of arguments, and a pair $y\#z$ is in the second argument. The following examples are in the file `BRelComposeTest.oz`.

```
% $Id: BRelComposeTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'BRelCompose.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'BRelCompose'}
{Test {BRelCompose nil [2#b 3#c]} '==' nil}
{Test {BRelCompose nil nil} '==' nil}
{Test {BRelCompose [1#2 2#3]  [2#b 3#c]}
      '==' [1#b 2#c]}
{Test {BRelCompose [1#2 1#3]  [2#b 3#c]}
      '==' [1#b 1#c]}
{Test {BRelCompose [1#3 2#3] [3#b 3#c]}
      '==' [1#b 1#c 2#b 2#c]}
{StartTesting done}
```

25. (5 points) [UseModels] [Concepts]

Define a function

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

that takes a list of strings and returns a single string that contains the given strings in the order given, separated by `", "`. Test the examples below by using the file `CommaSeparateTest.oz`, which inserts the actual examples from the file and `CommaSeparateBodyTest.oz`.

```
% $Id: CommaSeparateTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'CommaSeparate.oz'
\insert 'TestingNoStop.oz'
\insert 'CommaSeparateBodyTest.oz'
```

```
% $Id: CommaSeparateBodyTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
{StartTesting 'CommaSeparate'}
{Test {CommaSeparate nil} '==' ""}
{Test {CommaSeparate ["a" "b"]} '==' "a, b"}
{Test {CommaSeparate ["Monday" "Tuesday" "Wednesday" "Thursday"]}
      '==' "Monday, Tuesday, Wednesday, Thursday"}
{StartTesting done}
```

26. (5 points) [UseModels] [Concepts]

Define a function

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

that takes a list of strings and returns a single string that, when printed, shows the strings on separate lines.

Test the examples below by feeding the file `OnSeparateLinesTest.oz` which inserts the actual examples from the file `OnSeparateLinesBodyTest.oz`.

```
% $Id: OnSeparateLinesTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'OnSeparateLines.oz'
\insert 'TestingNoStop.oz'
\insert 'OnSeparateLinesBodyTest.oz'
```

```
% $Id: OnSeparateLinesBodyTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
{StartTesting 'OnSeparateLines'}
{Test {OnSeparateLines nil} '==' ""}
{Test {OnSeparateLines ["a" "b"]} '==' "a\nb"}
{Test {OnSeparateLines ["Monday" "Tuesday" "Wednesday" "Thursday"]}
      '==' "Monday\nTuesday\nWednesday\nThursday"}
{StartTesting done}
```

27. (20 points) [UseModels] [Concepts]

Define a curried function

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

that is a generalization of `onSeparateLines` and `commaSeparated`. Put your code in a file `SeparatedBy.oz`.

Then write a testing file `SeparatedByTesting.oz` that shows how to use your definition of the function `SeparatedBy` to define both functions `CommaSeparate` and `OnSeparateLines`. Your testing file should continue to runs the tests in both `CommaSeparateBodyTest.oz` and `OnSeparateLinesBodyTest.oz`, to test these definitions.

28. (5 points) [UseModels]

Define the function `MyAppend` to be just like the standard `Append` function. However, your definition is to be written using `FoldR`, completing the following by adding arguments to the call of `FoldR`. (For a description of `FoldR`, see Section 6.3 of "The Oz Base Environment" [DKS06].)

```
fun {MyAppend Xs Ys}
  {FoldR  _________________________ }
end
```

Put your solution in a file MyAppend.oz and test it using the file MyAppendTest.oz that we provide.

29. (5 points) [UseModels]

Using FoldR in a way similar to the previous problem, define

```
DoubleAll: <fun {$ <List Number>}: <List Number>>
```

that takes a list of Numbers, and returns a list with each of its elements doubled. Testing the following examples can be done by running the file DoubleAllTest.oz, which inserts the file DoubleAllBodyTest.oz, which has the actual examples.

```
% $Id: DoubleAllTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'DoubleAll.oz'
\insert 'TestingNoStop.oz'
\insert 'DoubleAllBodyTest.oz'
```

```
% $Id: DoubleAllBodyTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
{StartTesting 'DoubleAll'}
{Test {DoubleAll nil} '==' nil}
{Test {DoubleAll [1 2 3]} '==' [2 4 6]}
{Test {DoubleAll [3 6 2 5 4 1]} '==' [6 12 4 10 8 2]}
{StartTesting done}
```

30. (15 points) [UseModels]

Define the function MyMap to be just like the standard Map function. Your definition is to be done by using FoldR.

Then you will have to write a testing file, MyMapTesting.oz, and hand that in to demonstrate that you know how to use MyMap. As part of your testing, use MyMap to (a) declare DoubleAll and test your definition (by inserting the file DoubleAllBodyTest.oz that we provide) and (b) to add 1 to all the elements of a list of Ints.

Turn in both your code in a file MyMap.oz the testing code in MyMapTesting.oz, and the test output.

31. [UseModels]

Consider the following type

```
<Tree T> ::= node(item:T subtrees:<List <Tree T>>)
```

for $n$ary-trees, which represents a Tree of elements of some type T as a node record, which contains a field item of type T and a list of subtrees.

(a) (10 points) Define a function

```
SumTree: <fun {$ <Tree Int>}: Int>
```

that adds together all the Ints in a Tree of Ints.
You can test your definition of SumTree using the code given in SumTreeTest.oz (see Figure 11 on page 20), which uses the examples from the file SumTreeBodyTest (also in the figure). The latter gives some examples.

(b) (15 points) Define a function

```
MapTree: <fun {$ <Tree S> <fun {$ S}: T>}: <Tree T>>
```

that takes a Tree t and a function f and returns a tree that has the same shape of t, but where each item x is replaced by the result of applying f to x.
You can test your definition of MapTree using the code given in MapTreeTest.oz (see Figure 12 on page 21), which uses the examples from the file MapTreeBodyTest (also in the figure). The latter gives some examples.

(c) (30 points) By generalizing your answers to the above problems, define an Oz function `FoldTree` that is analogous to `FoldR` for lists. This should take a tree, a function to replace the node constructor, a function to replace the `|` constructor for lists, and a value to replace the empty list.

The following testing code, in `FoldTreeTest.oz`, tests that your definition can define `SumTree`, and `MapTree` on Trees.

```
% $Id: FoldTreeTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'TestingNoStop.oz'
\insert 'FoldTree.oz'
declare
fun {Add X Y} X + Y end
fun {SumTree Tree} {FoldTree Tree Add Add 0} end
fun {MapTree Tree F}
   {FoldTree Tree
             fun {$ I Strs} node(item:{F I} subtrees:Strs) end
             fun {$ E Es} E|Es end
             nil}
end
\insert 'SumTreeBodyTest.oz'
\insert 'MapTreeBodyTest.oz'
```

```
% $Id: SumTreeTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'SumTree.oz'
\insert 'TestingNoStop.oz'
\insert 'SumTreeBodyTest.oz'

% $Id: SumTreeBodyTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
{StartTesting 'SumTree'}
{Test {SumTree node(item:4 subtrees:nil)} '==' 4}
{Test {SumTree
       node(item:3
            subtrees:[node(item:4 subtrees:nil)
                      node(item:7 subtrees:nil)])} '==' 14}
{Test {SumTree
       node(item:10
            subtrees:[node(item:3
                           subtrees:[node(item:4 subtrees:nil)
                                     node(item:7 subtrees:nil)])
                      node(item:10
                           subtrees:[node(item:20 subtrees: nil)
                                     node(item:30 subtrees: nil)
                                     node(item:40 subtrees: nil)]
                          )])}
  '==' 124}
{StartTesting done}
```

Figure 11: Testing for problem a.

```
% $Id: MapTreeTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'MapTree.oz'
\insert 'TestingNoStop.oz'
\insert 'MapTreeBodyTest.oz'

% $Id: MapTreeBodyTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
{StartTesting 'MapTree'}
local
   fun {Add1 X} X+1 end
   fun {Add3 X} X+3 end
in
   {Test {MapTree node(item:4 subtrees:nil) Add1}
         '==' node(item:5 subtrees:nil)}
   {Test {MapTree node(item:3
                   subtrees:[node(item:4 subtrees:nil)
                             node(item:7 subtrees:nil)])
                  Add3}
         '==' node(item:6
                   subtrees:[node(item:7 subtrees:nil)
                             node(item:10 subtrees:nil)])}
   {Test {MapTree
          node(item:10
               subtrees:[node(item:3
                              subtrees:[node(item:4 subtrees:nil)
                                        node(item:7 subtrees:nil)])
                         node(item:10
                              subtrees:[node(item:20 subtrees: nil)
                                        node(item:30 subtrees: nil)
                                        node(item:40 subtrees: nil)]
                             )])
          Add3}
    '==' node(item:13
             subtrees:[node(item:6
                            subtrees:[node(item:7 subtrees:nil)
                                      node(item:10 subtrees:nil)])
                       node(item:13
                            subtrees:[node(item:23 subtrees: nil)
                                      node(item:33 subtrees: nil)
                                      node(item:43 subtrees: nil)]
                           )])}
end
{StartTesting done}
```

Figure 12: Testing for problem b.

32. (30 points) [UseModels] [Concepts]

A potentially infinite set (or PISet) can be described by a "characteristic function" (whose range is the Booleans) that determines if an element occurs in the set. For example, the function P such that

$$P(x) = x \text{ is an number and } x > 7$$

is the characteristic function for a potentially infinite set containing all numbers strictly greater than 7. Allowing the user to construct such a potentially infinite set from a characteristic function gives them the power to construct potentially infinite sets like $\{x \mid P(x)\}$ that contains an infinite number of elements (in this example, the set contains all numbers strictly greater than 7).

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

1. The function `PISetSuchThat` takes a characteristic function, F and returns a potentially infinite set such that each value X is in the PISet returned just when $\{F\ X\}$ is **`true`**.
2. The function `PISetUnion` takes two PISets, with characteristic functions F and G, and returns a PISet such that each value X is in the resulting PISet just when either $\{F\ X\}$ or $\{G\ X\}$ is **`true`**.
3. The function `PISetIntersect` takes two PISets, with characteristic functions F and G, and returns a PISet such that each value X is in the resulting PISet just when both $\{F\ X\}$ and $\{G\ X\}$ are **`true`**.
4. The function `PISetMember` returns a Boolean that tells whether the second argument is a member of its PISet that is its first argument.
5. The function `PISetComplement` returns a PISet that contains everything that is not in its argument PISet.

Note (hint, hint) that the equations in Figure 13 must hold, for all functions `F` and `G`, and elements `X` of appropriate types.

```
{PISetMember {PISetUnion (PISetSuchThat F} {PISetSuchThat G}} X}
    == {F X} orelse {G X}
{PISetMember {PISetIntersect (PISetSuchThat F} {PISetSuchThat G}} X}
    == {F X} andthen {G X}
{PISetMember {PISetSuchThat F} X} == {F X}
{PISetMember {PISetComplement {PISetSuchThat F}} X} == {Not {F X}}
```

Figure 13: Equations that give hints for problem 32.

As examples, consider the tests in Figure 14 on the next page.

```
% $Id: PISetTest.oz,v 1.2 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'PISet.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'PISet'}
declare
fun {IsCoke X} X == coke end
fun {IsPepsi X} X == pepsi end
fun {GTMaker Y} fun {$ X} {IsNumber X} andthen X > Y end end
GT5 = {GTMaker 5}
GT7 = {GTMaker 7}

{Test {PISetMember {PISetSuchThat IsCoke} coke} '==' true}
{Test {PISetMember {PISetSuchThat IsCoke} pepsi} '==' false}
{Test {PISetMember {PISetComplement {PISetSuchThat IsCoke}} coke} '==' false}
{Test {PISetMember {PISetUnion {PISetSuchThat IsCoke} {PISetSuchThat IsPepsi}}
       pepsi} '==' true}
{Test {PISetMember {PISetUnion {PISetSuchThat IsCoke} {PISetSuchThat IsPepsi}}
       coke} '==' true}
{Test {PISetMember {PISetUnion {PISetSuchThat IsCoke} {PISetSuchThat IsPepsi}}
       sprite} '==' false}
{Test {PISetMember
         {PISetIntersect {PISetSuchThat IsCoke} {PISetSuchThat IsPepsi}}
       coke} '==' false}
{Test {PISetMember {PISetSuchThat GT5} coke} '==' false}
{Test {PISetMember {PISetSuchThat GT7} coke} '==' false}
{Test {PISetMember {PISetSuchThat GT7} 8} '==' true}
{Test {PISetMember {PISetSuchThat GT7} 7} '==' false}
{Test {PISetMember {PISetSuchThat GT7} 6} '==' false}
{Test {PISetMember {PISetSuchThat GT7} 999092384084184} '==' true}
{Test {PISetMember {PISetSuchThat GT5} 999092384084184} '==' true}
{Test {PISetMember {PISetSuchThat GT5} 7} '==' true}
{Test {PISetMember {PISetSuchThat GT5} 6} '==' true}
{Test {PISetMember {PISetSuchThat GT5} 5} '==' false}
{Test {PISetMember {PISetUnion {PISetSuchThat GT5} {PISetSuchThat GT7}} 6}
 '==' true}
{Test {PISetMember {PISetIntersect {PISetSuchThat GT5} {PISetSuchThat GT7}} 6}
 '==' false}
{Test {PISetMember {PISetComplement {PISetSuchThat GT5}} coke} '==' true}
{Test {PISetMember {PISetComplement {PISetSuchThat GT7}} coke} '==' true}
{Test {PISetMember {PISetComplement {PISetSuchThat GT7}} 8} '==' false}
{Test {PISetMember {PISetComplement {PISetSuchThat GT7}} 7} '==' true}
{Test {PISetMember {PISetComplement {PISetSuchThat GT7}} 6} '==' true}
{Test {PISetMember {PISetComplement {PISetSuchThat GT7}} 999092384084184}
 '==' false}
{Test {PISetMember {PISetComplement {PISetSuchThat GT5}} 999092384084184}
 '==' false}
{Test {PISetMember {PISetComplement {PISetSuchThat GT5}} 7} '==' false}
{Test {PISetMember {PISetComplement {PISetSuchThat GT5}} 6} '==' false}
{Test {PISetMember {PISetComplement {PISetSuchThat GT5}} 5} '==' true}
{Test {PISetMember {PISetComplement {PISetSuchThat GT5}} ~5} '==' true}
{StartTesting done}
```

Figure 14: Example tests for problem 32.

33. (25 points) [UseModels]

Consider the following data grammars.

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

In this grammar, `boolLit`, `intLit`, and `charLit` 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 `ifExp(`E_1 E_2 E_3`)` represent if-then-else expressions, i.e., **if** E_1 **then** E_2 **else** E_3 **end**.

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 15 on the following 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 15 on the next 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.)

Points

This homework's total points: 491. 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.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

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

```
% $Id: TypeOfTest.oz,v 1.1 2009/02/03 05:19:00 leavens Exp leavens $
\insert 'TypeOf.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'TypeOf'}
{Test {TypeOf equalExp(intLit(3) intLit(4))} '==' obool}
{Test {TypeOf subExp(intLit(3) intLit(4))} '==' oint}
{Test {TypeOf subExp(intLit(3) intLit(4))} '==' oint}
{Test {TypeOf subExp(charLit(&a) intLit(4))} '==' owrong}
{Test {TypeOf subExp(intLit(4) charLit(&a))} '==' 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(boolLit(true)
                                   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}
{StartTesting done}
```

Figure 15: Examples for problem 33.