COP 4020 — Programming Languages I February 26, 2012

# Homework 3: Declarative Programming

See Webcourses and the syllabus for due dates.
Note that you can't do quizzes on webcourses after their due date, so be sure to take any webcourses quizzes by the date they are due!
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, code with extra unnecessary cases, or code that is excessively hard to follow. You should always assume that the inputs given will follow the grammar for the types specified, and so your code should not have extra cases for inputs that do not follow the grammar. Avoid duplicating code by using helping functions or by using syntactic sugars and local definitions. It is a good idea to check your code for these problems before submitting.
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. You can find all the tests we provide in a zip file, which you can download from problem 1's assignment on Webcourses.
If the tests don't pass, please try to say why they don't pass, as this enhances communication and makes commenting on the code easier and more specific to your problem.
**What to Turn In:** For each problem that requires code, turn in (on Webcourses) your code and output of your testing. Please upload code as a plain (text) file with the name given in the problem or testing file and with the suffix `.oz`, and also paste the output from our tests into the answer box on webcourses. For English answers, please paste your answer into the answer box in the problem's "assignment" on Webcourses. For a problem with a mix of code and English, follow both of the above.
Your code should compile with Oz, if it doesn't you probably should keep working on it. Email the staff with your code file if you need help getting it to compile. If you don't have time, at least tell us that you didn't get it to compile.
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]

   Can one write deterministic programs in C, C++, C#, or Java? (a) pick your language, (b) answer "yes" or "no," and (c) give a brief explanation.

2. (18 points) Read section 3.1-3.4.4 in the textbook and then take the quiz titled "Section 3.1-3.4 quiz" on webcourses.

3. (7 points) Read chapter 3, sections 3.5-3.7 of the textbook [VH04] (you can just skim 3.7.3) and then take the quiz titled "Section 3.5-3.7 quiz" on webcourses.

4. (3 points) Read chapter 3, sections 3.8-3.9 of the textbook [VH04] and then take the quiz titled "Section 3.8-3.9 quiz" on webcourses.

## Regular Problems

### Iteration

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

5. (0 points) [UseModels] (suggested practice only; don't hand this in) For practice with iteration, 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 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'SumList.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'SumList $Revision: 1.2 $'}
{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}
```

6. (10 points) [UseModels] In Oz, write an iterative function

```
HasPrefix: <fun {$ <List T> <List T>}: <Bool>>
```

   that for some type `T` takes a list `Ls` and another list `SubLs`, and returns true just when all the elements of `SubLs` occur in order at the beginning of `Ls`. You are *not* allowed to use the Oz built-in function `List.isPrefix` in your solution. Figure 1 on the following page has examples that you can find in our test file `HasPrefixTest.oz`.

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

   Put your code in a file `HasPrefix.oz`. After doing your own testing, run our tests in `HasPrefixTest.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).

```
% $Id: HasPrefixTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'HasPrefix.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'HasPrefixTest.oz $Revision: 1.2 $'}
{Test {HasPrefix "" ""} '==' true}
{Test {HasPrefix "<-- there's an empty string here" ""} '==' true}
{Test {HasPrefix "we're off to see the wizard" "we"} '==' true}
{Test {HasPrefix "we're off to see the wizard" "we're off"} '==' true}
{Test {HasPrefix "we're off to see the wizard" "to see"} '==' false}
{Test {HasPrefix "we're off to see the wizard" "ew"} '==' false}
{Test {HasPrefix [a b c] [a]} '==' true}
{Test {HasPrefix [[a b c] [d e]] [[a b c]]} '==' true}
{Test {HasPrefix [[a b c] [d e]] [a b c]} '==' false}
{Test {HasPrefix [1 2 3 4 3 2 1 7] [1 2 3 4 3]} '==' true}
{Test {HasPrefix [x y z] [x y z a b c]} '==' false}
{DoneTesting}
```

Figure 1: Testing code for problem 6.

7. (15 points) [UseModels] In Oz, write an iterative function

   `OccursInList: <`**`fun`**` {$ <List T> <List T>}: <Int>>`

   that for some type `T` takes a list `Ls` and another list `SubLs`, and returns the first index of `Ls` at which all the elements of `SubLs` occur in order. If the elements of `SubLs` do not occur in `Ls` in some order, then the function returns ~1. Figure 2 has examples that you can find in our test file `OccursInListTest.oz`.

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

```
% $Id: OccursInListTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'OccursInList.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'OccursInListTest.oz $Revision: 1.2 $'}
{Test {OccursInList nil nil} '==' ~1}
{Test {OccursInList "<-- see the empty string here" ""} '==' 1}
{Test {OccursInList "now is the time" ""} '==' 1}
{Test {OccursInList "" "ball"} '==' ~1}
{Test {OccursInList "a balloon boonanza" "ball"} '==' 3}
{Test {OccursInList "a balloon boonanza" "oo"} '==' 7}
{Test {OccursInList "concepts, techniques, and models" "tech"} '==' 11}
{Test {OccursInList "a tale of two cities" "es"} '==' 19}
{Test {OccursInList "the trouble with that is this: the trouble with tribbles" "trouble"} '==' 5}
{Test {OccursInList "the trouble with that is this: the trouble with tribbles" "pizza"} '==' ~1}
{Test {OccursInList "the trouble with that is this: the trouble with tribbles" "the"} '==' 1}
{Test {OccursInList "the trouble with that is this: the trouble with tribbles" "with "} '==' 13}
{Test {OccursInList "calculus I" "easy stuff you don't need to worry about"} '==' ~1}
{Test {OccursInList [[a b c] [d e f] [h i j]] [[d e f] [h i j]]} '==' 2}
{Test {OccursInList [[a b c] [d e f] [h i j]] [d e f]} '==' ~1}
{DoneTesting}
```

Figure 2: Testing code for problem 7.

## 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.

(Also, you might try the self-test on “following the grammar” on Webcourses.)

8. (10 points) [UseModels] Write an Oz function

   ```
   Has3List : <fun {$ <List <List T>>}: <Bool>>
   ```

   takes a list of lists (of some type `T`), `LL`, and returns true just when `LL` contains a list with exactly three elements. There are examples in Figure 3.

   Note: to properly follow the grammar, you will need at least one helping function.

```
% $Id: Has3ListTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'Has3List.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'Has3ListTest $Revision: 1.2 $'}
{Test {Has3List nil} '==' false}
{Test {Has3List [[a b] [c] [d e f] [g] [h]]} '==' true}
{Test {Has3List [[a b] [c] [d e] [f] [g] [h]]} '==' false}
{Test {Has3List [[a b] [c] [d e] [f]]} '==' false}
{Test {Has3List [[a b c d] [d e] [f]]} '==' false}
{Test {Has3List [[nil nil nil]]} '==' true}
{Test {Has3List [[[a b c]]]} '==' false}
{Test {Has3List [[a nil c]]} '==' true}
{Test {Has3List [[a b] [c] [d e] nil [g] [h]]} '==' false}
{Test {Has3List [nil [1] [2 2] [3 3 3] [4 4 4 4]]} '==' true}
{DoneTesting}
```

Figure 3: Testing for problem 8.

9. (10 points) [UseModels] Write, in Oz, a function

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

   takes a list of atoms, `LoA`, and two atoms: `AfterThis` and `What`. This function returns a list that is just like `LoA`, except that in each place where `AfterThis`, it puts a copy of `What` following it. There are examples in Figure 4 on the next page

10. (10 points) [UseModels] Write, in Oz, a function

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

    takes a list of lists of atoms, `LL`, and two atoms: `AfterThis` and `What`; this function inserts `What` after each occurrence of `AfterThis` throughout each of the sublists in `LL`. There are examples in Figure 5 on the following page

```
% $Id: InsertAfterTest.oz,v 1.3 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'InsertAfter.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'InsertAfterTest $Revision: 1.3 $'}
{Test {InsertAfter nil a w} '==' nil}
{Test {InsertAfter a|b|nil a w} '==' a|w|b|nil}
{Test {InsertAfter [c x y c z c m q] c f} '==' [c f x y c f z c f m q]}
{Test {InsertAfter [m q] c f} '==' [m q]}
{Test {InsertAfter [c m q] c f} '==' [c f m q]}
{DoneTesting}
```

Figure 4: Testing for problem 9.

```
% $Id: InsertAfterInTest.oz,v 1.3 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'InsertAfterIn.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'InsertAfterInTest $Revision: 1.3 $'}
{Test {InsertAfterIn nil a w} '==' nil}
{Test {InsertAfterIn [nil nil] a w} '==' [nil nil]}
{Test {InsertAfterIn [[x y] [z z y] [x x o o]] x y} '==' [[x y y] [z z y] [x y x y o o]]}
{Test {InsertAfterIn [[a b c] [c] [c c] [c x y z m q] nil] c f}
                '==' [[a b c f] [c f] [c f c f] [c f x y z m q] nil]}
{Test {InsertAfterIn [[a good egg] [boiled] nil [is an egg]
                      [that is egg enough]] egg souffle}
 '==' [[a good egg souffle] [boiled] nil [is an egg souffle]
       [that is egg souffle enough]]}
{Test {InsertAfterIn [[a florida senator] [got florida 'in' trouble]
                      [with florida football]] florida state}
 '==' [[a florida state senator] [got florida state 'in' trouble]
       [with florida state football]]}
{DoneTesting}
```

Figure 5: Testing for problem 10.

11. (20 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`. First, the following substitutions take place when there are three consecutive atoms that match the pattern:

    - `by the way` is replaced by `btw`,
    - `for your information` is replaced by `fyi`, and
    - `be right back` is replaced by `brb`.

    Second, the following substitutions take place when there are two consecutive atoms that match the pattern:

    - `'I'see` is replaced by `ic`,
    - `see you` is replaced by `cya`, and
    - `voice mail` is replaced by `vm`.

    Finally, all of the following subtitutions are made on individual atoms, when not superceded by one of the above:

    - `you` is replaced by `u`,
    - `are` is replaced by `r`,
    - `your` is replaced by `ur`,
    - `boyfriend` is replaced by `bf`,
    - `girlfriend` is replaced by `gf`,
    - `great` is replaced by `gr8`.

    These lists r complete (for this problem).

    Hints: to handle the 3 element and 2 element substitutions, u may find it useful to have separate helping functions to look for 3 (and 2) elements that should be replaced. Arrange these functions in a "pipeline," with the function that looks for 3 element replacements processing the list argument first, and passing its result to the function that looks for 2 element replacements, which in turn passes its result to a function that looks for single atom replacements.

    The examples in Figure 6 on the next page 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.

    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 some repeated code by using helping functions or case-expressions. (Note that a case-expression can be used inside a larger expression.)

```
% $Id: HepTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'Hep.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'HepTest $Revision: 1.2 $'}
{Test {Hep nil} '==' nil}
{Test {Hep [you you you you]} '==' [u u u u]}
{Test {Hep ['I' see your voice mail]} '==' [ic ur vm]}
{Test {Hep [hey 'I' see your voice mail]} '==' [hey ic ur vm]}
{Test {Hep [by the way 'for' your information 'I' will be right back]}
      '==' [btw fyi 'I' will brb]}
{Test {Hep [hey by the way 'for' your information 'I' will be right back]}
      '==' [hey btw fyi 'I' will brb]}
{Test {Hep [hey you by the way 'for' your information 'I' will be right back]}
      '==' [hey u btw fyi 'I' will brb]}
{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]}
{Test {Hep [see you you are great by the way]} '==' [cya u r gr8 btw]}
{Test {Hep [you are your own 'for' your information a programmer]}
       '==' [u r ur own fyi a programmer]}
{DoneTesting}
```

Figure 6: Tests for problem 11.

12. (15 points) [UseModels]

    Write a function

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

    that takes an Oz Value, `Val`, and returns true just when `Val` is a relation. A value is a *relation* if and only if: (a) it is in the grammar Figure 7 for ⟨Relation⟩, and thus is a list of ⟨Record⟩s, (b) each relation record in the list has the same label and arity, and (c) the list of relation records has no duplicates (using == to compare elements).

⟨Relation⟩ ::= ⟨List ⟨Record⟩⟩

Figure 7: The grammar for the type ⟨Relation⟩. Here ⟨Record⟩ is the types of Oz records [DKS06, Section 2].

Hints: As usual, you can use Oz built-in functions to help with this. For example, you might want to use `IsList`, `IsRecord`, `Label`, `Arity`, `Not`, `Member`, and `All` [DKS06].

Figure Figure 8 shows the tests from our test file `IsAListTest.oz`. Put your code in a file `IsRelation.oz` and test using our tests.

```
% $Id: IsRelationTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'IsRelation.oz'
\insert 'RelationExamples.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'IsRelationTest.oz $Revision: 1.2 $'}
{Test {IsRelation nil} '==' true}
{Test {IsRelation 4020} '==' false}
{Test {IsRelation 4020|nil} '==' false}
{Test {IsRelation letters(a 1)|letters(b 2)|nil} '==' true}
{Test {IsRelation [letters(a 1) letters(b 2) letters(c 3)]} '==' true}
{Test {IsRelation [letters(a 1) letters(b 2) letters(a 1)]} '==' false}
{Test {IsRelation [letters(a 1) letters(b 2) letters(b 2)]} '==' false}
{Test {IsRelation [letters(a 1) letters(b 2) letters(b 2)]} '==' false}
{Test {IsRelation letters(a 1)#letters(b 2)} '==' false}
{Test {IsRelation [empty()]} '==' true}
{Test {IsRelation [lettersSchema(letter number)]} '==' true}
{Test {IsRelation [compmaker(who:dell where:texas kind:pc)
                   compmaker(who:apple where:california kind:mac)]} '==' true}
{Test {IsRelation [add(1 1 2) add(1 2 3) add(1 3 4) add(1 7 8)
                   add(2 1 3) add(2 2 4) add(2 3 5) add(2 4 6)]} '==' true}
{Test {IsRelation [rr(a b c d e f g h i j k) rr(l m n o p q r s t u v)]} '==' true}
{Test {IsRelation [rr(a b c d e f g h i j k) rr(l m n o p q r s t u v)
                   rr(w x y z)]} '==' false}
{Test {IsRelation Fish} '==' true}
{Test {IsRelation Cities} '==' true}
{Test {IsRelation Ports} '==' true}
{DoneTesting}
```

Figure 8: Tests for 12. The file `RelationExamples.oz` is shown in Figure 9 on the following page.

```
% $Id: RelationExamples.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
% Some example relations
declare
Fish = [fish(kind:tuna city:'Tokyo' country:'Japan')
        fish(kind:shrimp city:'Manchester' country:'United Kingdom')
        fish(kind:octopus city:'Seoul' country:'South Korea')
        fish(kind:abalone city:'Canton' country:'China')
        fish(kind:crab city:'Shanghai' country:'China')
        fish(kind:oyster city:'Singapore' country:'Singapore')
        fish(kind:squid city:'Rio de Janeiro' country:'Brazil')]
Cities = [city(name:'Tokyo' country:'Japan' population:34400000)
          city(name:'Canton' country:'China' population:25600000)
          city(name:'Seoul' country:'South Korea' population:25300000)
          city(name:'Shanghai' country:'China' population:25100000)
          city(name:'Mexico City' country:'Mexico' population:23100000)
          city(name:'Delhi' country:'India' population:22900000)]
Ports = [port(name:'Shanghai' country:'China' tons:537)
         port(name:'Singapore' country:'Singapore' tons:448)
         port(name:'Rotterdam' country:'Netherlands' tons:378)
         port(name:'Ningbo' country:'China' tons:307)
         port(name:'Guangzhou' country:'China' tons:302)
         port(name:'Tianjin' country:'China' tons:257)]
```

Figure 9: Some examples of Relations.

13. (10 points) [UseModels] In Oz, write a function

    `RelUnion: <fun {$ <Relation> <Relation>}: <Relation> >`

    that takes two `<Relation>`s, `R1` and `R2`, that have the same label and arity for all their records, and returns a relation containing all the records of both `R1` and `R2`, but without any duplicates. You may assume in that the argument relations have the records that all contain the same label and arity (field names). See Figure 10 for examples from our test file `RelUnionTest.oz`.

    Put your code in a file `RelUnion.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.)

```
% $Id: RelUnionTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'RelUnion.oz'
\insert 'RelationTesting.oz'

declare
{StartTesting 'RelUnionTest.oz $Revision: 1.2 $'}
{EqTest {RelUnion nil [spacecraft(series:mercury date:1962)]}
 '=set=' [spacecraft(series:mercury date:1962)]}
{EqTest {RelUnion [spacecraft(series:mercury date:1962)] nil}
 '=set=' [spacecraft(series:mercury date:1962)]}
{EqTest {RelUnion [spacecraft(series:mercury date:1962)] [spacecraft(series:mercury date:1962)]}
 '=set=' [spacecraft(series:mercury date:1962)]}
{EqTest {RelUnion [spacecraft(series:mercury date:1962)] [spacecraft(series:mercury date:1962)]}
 '=set=' [spacecraft(series:mercury date:1962)]}
{EqTest {RelUnion [spacecraft(series:apollo date:1969)
                   spacecraft(series:mercury date:1962)]
         [spacecraft(series:mercury date:1962)
          spacecraft(series:apollo date:1969)]}
 '=set=' [spacecraft(series:mercury date:1962)
       spacecraft(series:apollo date:1969)]}
{EqTest {RelUnion [add(3 4 7) add(1 2 3) add(1 3 4) add(1 4 5)]
         [add(1 4 5) add(1 5 6) add(1 2 3) add(1 3 4) add(2 6 8) add(3 7 10)]}
 '=set=' [add(3 4 7) add(1 4 5) add(1 5 6) add(1 2 3) add(1 3 4) add(2 6 8) add(3 7 10)]}
{DoneTesting}
```

Figure 10: Tests for problem 13. These use testing code from `RelationTesting.oz`.

14. (10 points) [UseModels] In Oz, write a function

    `RelSelect: <fun {$ <Relation> <fun {$ <RelationRecord>}: <Bool>>}: <Relation> >`

    that takes a `<Relation>`, `Rel`, and a predicate `Pred` (a function from relation records to Booleans), and returns a `<Relation>` that contains all records in `Rel` that satisfy `Pred`. You should assume that `{IsRelation Rel}` is true; that is, the argument `Rel` satisfies all the properties required of a relation. The result should also satisfy these properties. See Figure 11 on the following page for examples from our test file `RelSelectTest.oz`.

    Put your code in a file `RelSelect.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.)

```
% $Id: RelSelectTest.oz,v 1.4 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'RelSelect.oz'
\insert 'RelationExamples.oz'
\insert 'RelationTesting.oz'

{StartTesting 'RelSelectTest.oz $Revision: 1.4 $'}
{EqTest {RelSelect nil fun {$ RR} RR.1 > 7 end} '=set=' nil}
{EqTest {RelSelect Fish fun {$ RR} RR.country == 'China' end}
 '=set=' [fish(kind:abalone city:'Canton' country:'China')
       fish(kind:crab city:'Shanghai' country:'China')]}
{EqTest {RelSelect Fish fun {$ RR} RR.kind == tuna orelse RR.kind == squid end}
 '=set=' [fish(kind:tuna city:'Tokyo' country:'Japan')
       fish(kind:squid city:'Rio de Janeiro' country:'Brazil')]}
{EqTest {RelSelect Ports fun {$ R} R.country == 'China' end}
 '=set=' [port(name:'Shanghai' country:'China' tons:537)
       port(name:'Ningbo' country:'China' tons:307)
       port(name:'Guangzhou' country:'China' tons:302)
       port(name:'Tianjin' country:'China' tons:257)]}
{EqTest {RelSelect Cities fun {$ R} R.population > 25000000 end}
 '=set=' [city(name:'Tokyo' country:'Japan' population:34400000)
       city(name:'Canton' country:'China' population:25600000)
       city(name:'Seoul' country:'South Korea' population:25300000)
       city(name:'Shanghai' country:'China' population:25100000)]}
{EqTest {RelSelect Ports fun {$ R} R.country == 'China' andthen R.tons > 300 end}
 '=set=' [port(name:'Shanghai' country:'China' tons:537)
       port(name:'Ningbo' country:'China' tons:307)
       port(name:'Guangzhou' country:'China' tons:302)]}
{EqTest {RelSelect {RelSelect Ports fun {$ R} R.country == 'China' end}
                fun {$ R} R.tons > 500 end}
 '=set=' [port(name:'Shanghai' country:'China' tons:537)]}
{EqTest {RelSelect {RelSelect Ports fun {$ R} R.country == 'China' end}
                fun {$ R} R.tons > 800 end}
 '=set=' nil}
{DoneTesting}
```

Figure 11: Tests for problem 14. These use the examples in Figure 9 on page 9. They also use testing code from `RelationTesting.oz`.

15. (15 points) [UseModels]

In Oz, implement the function

`MapInner: <fun {$ <List <List T>> <fun {$ T}: <List S> >}: <List <ListS>> >`

that takes a list of lists `LL` 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 returns a list that is the result of applying `F` to each element in each inner list, and forming a list of the results. The list of results appears in the same order as the corresponding elements (at both levels of the lists). File `MapInnerTest.oz` contains various examples (see Figure 12).

```
% $Id: MapInnerTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'MapInner.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'MapInnerTest.oz $Revision: 1.2 $'}
{Test {MapInner nil Not} '==' nil}
{Test {MapInner [[true false] [false true] nil] Not} '==' [[false true] [true false] nil]}
{Test {MapInner [[1 2 3 4 5 6] [7] nil [8 9 10] [6 3 2 1 5 7] [4 4 4]]
      fun {$ I} I+1 end}
 '==' [[2 3 4 5 6 7] [8] nil [9 10 11] [7 4 3 2 6 8] [5 5 5]]}
{Test {MapInner [[[a b] [c d]] [[e f] [h i]]] fun {$ X|Y|nil} Y|X|nil end}
 '==' [[[b a] [d c]] [[f e] [i h]]]}
{DoneTesting}
```

Figure 12: Tests for Problem 15.

16. (10 points) [UseModels] Consider the following grammar (where Lists and Atoms are standard).

⟨Text⟩ ::= ⟨List Paragraph⟩
⟨Paragraph⟩ ::= `para(`⟨List Atom⟩`)`

In Oz, write a function,

`NumWords: <fun {$ <Text>}: <Int>>`

which takes a Text, `Txt`, and returns the total number of atoms in it.

Be sure that your code follows the grammar! 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!

There are examples in Figure 13 on the next page.

```
% $Id: NumWordsTest.oz,v 1.3 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'NumWords.oz'
\insert 'TestingNoStop.oz'

{StartTesting 'NumWords'}
{Test {NumWords nil} '==' 0}
{Test {NumWords [para([and along came java]) para([cpp was miffed with java])]}  '==' 9}
{Test {NumWords [para([dum dum dum dum]) para([dum bum]) para([rum scum])]} '==' 8}
{Test {NumWords [para([it was the best 'of' times it was the worst 'of' times])]} '==' 12}
{Test {NumWords [para(nil)]} '==' 0}
{Test {NumWords [para([it was clear that the day would 'not' 'end' well])
                 para([the time had come to demolish the university])
                 para([however colorless green frogs still sat atop the stadium])]} '==' 27}
{DoneTesting}
```

Figure 13: Examples for 16.

17. (15 points) [UseModels] Using the same grammar as in the previous question,

$\langle$Text$\rangle$ ::= $\langle$List Paragraph$\rangle$
$\langle$Paragraph$\rangle$ ::= `para(`$\langle$List Atom$\rangle$`)`

in Oz, write a function,

`SubstAll: <`**fun** `{$ <Text> <Atom> <Atom>}: <Text>>`

which takes a Text, `Txt`, and two atoms `Old` and `New`, and returns a Text that is just like `Txt`, except that each occurrence of the atom `Old` is replaced by the value of `New`. The following are examples.

```
\insert 'SubstAll.oz'
{StartTesting 'SubstAll'}
{Test {SubstAll nil java csharp} '==' nil}
{Test {SubstAll [para([and along came java]) para([cpp was miffed with java])]
                java csharp}
  '==' [para([and along came csharp]) para([cpp was miffed with csharp])]}
{Test {SubstAll [para([dum dum dum dum]) para([dum bum]) para([rum scum])]
                dum ah}
 '==' [para([ah ah ah ah]) para([ah bum]) para([rum scum])]}
{Test {SubstAll [para([it was the best 'of' times it was the worst 'of' times])]
                times tests}
 '==' [para([it was the best 'of' tests it was the worst 'of' tests])]}
{DoneTesting}
```

18. (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 14 shows various examples.

```
\insert 'NegateIfs.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'NegateIfs $Revision: 1.2 $'}
{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 14: Tests for Problem 18.

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.

19. [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 15).

```
% $Id: CapitalizeTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'Capitalize.oz'
\insert 'TestingNoStop.oz'
declare
{StartTesting 'CapitalizeTest $Revision: 1.2 $'}
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 15: Test procedure for Problem 19 and its use.

Figure 15 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`.

20. (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 a curried version of `F`. Figure 16 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**. Also note that `Curry` cannot know anything about the function `F` that is its argument; in other words, `Curry` must operate in the same way, regardless of what `F` is.

```
% $Id: CurryTest.oz,v 1.3 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'Curry.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'Curry'}
{Test {{{Curry Number.'+'} 3} 6} '==' 9}
{Test {{{Curry Number.'+'} 5} 6} '==' 11}
{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}
for I in 300 .. 310
do {Test {{{Curry fun {$ X Y} X+Y+I end} I*20} I*500} '==' (I*20)+(I*500)+I}
end
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]}
   {Test {{CA [color less]} [green ideas]} '==' [color less green ideas]}
end
{DoneTesting}
```

Figure 16: Examples for problem 20.

21. (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 17), which inserts the actual examples from the file and `SearchForZeroBodyTest.oz` (Figure 18).

```
% $Id: SearchForZeroTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'TestingNoStop.oz'
\insert 'SearchForZero.oz'
\insert 'SearchForZeroBodyTest.oz'
```

Figure 17: The file `SearchForZeroTest.oz`.

```
% $Id: SearchForZeroBodyTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
{StartTesting 'SearchForZeroBodyTest $Revision: 1.2 $'}
{Test {SearchForZero fun {$ 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}
{Test {SearchForZero fun {$ N} N*N - 10000 end} '==' 100}
{Test {SearchForZero fun {$ N} N*N - 1000000 end} '==' 1000}
{DoneTesting}
```

Figure 18: The file `SearchForZeroBodyTest.oz`.

22. (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 19) which inserts the actual examples from the file SearchForFixedPointBodyTest.oz (Figure 20).

```
% $Id: SearchForFixedPointTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'TestingNoStop.oz'
\insert 'SearchForFixedPoint.oz'
\insert 'SearchForFixedPointBodyTest.oz'
```

Figure 19: The file SearchForFixedPointTest.oz.

```
% $Id: SearchForFixedPointBodyTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
{StartTesting 'SearchForFixedPointBodyTest $Revision: 1.2 $'}
{Test {SearchForFixedPoint fun {$ 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 20: The file SearchForFixedPointBodyTest.oz.

23. (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 21, 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 21: 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`.

24. (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 22 on the following page for examples.

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
```

```
% $Id: CountTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'TestingNoStop.oz'
\insert 'Count.oz'
{StartTesting 'CountTest $Revision: 1.2 $'}
{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 22: Tests for Problem 24 on the previous page.

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⟩)
```

25. (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 23), 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.

```
% $Id: HighestNoteTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'TestingNoStop.oz'
\insert 'HighestNote.oz'
\insert 'HighestNoteBodyTest.oz'

% $Id: HighestNoteBodyTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
{StartTesting 'HighestNoteBodyTest $Revision: 1.2 $'}
{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 23: Testing for problem 25.

26. (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 24) is the driver that you use to run the tests. The file `TransposeBodyTest.oz` (Figure 25 on the following page) contains the actual test cases.

```
% $Id: TransposeTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'TestingNoStop.oz'
\insert 'Transpose.oz'
\insert 'TransposeBodyTest.oz'
```

Figure 24: Testing for problem 26.

27. (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 26 on page 23 has testing code, in `FoldMusicTest.oz`, tests that your definition of `FoldMusic` can be used to define `HighestNote`, and `Transpose`.

```
% $Id: TransposeBodyTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
{StartTesting 'TransposeBodyTest $Revision: 1.2 $'}
{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 25: Body of tests for problem 26.

```
% $Id: FoldMusicTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\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 26: Testing for Problem 27 on page 21.

28. (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 27 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 27: Equations that give hints for problem 28.

As examples, consider the tests in Figure 28 on the following page.

```
% $Id: PIBagTest.oz,v 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'PIBag.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'PIBagTest $Revision: 1.2 $'}
declare
fun {Cokes X} if X == coke then 6 else 0 end end
fun {Beers X} if X == beer then 12 else 0 end end
fun {GTMaker Y} fun {$ 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 28: Example tests for problem 28.

29. (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 29 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 29 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: 383. 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 1.2 2012/02/26 22:02:15 leavens Exp leavens $
\insert 'TypeOf.oz'
\insert 'TestingNoStop.oz'
{StartTesting 'TypeOfTest $Revision: 1.2 $'}
{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 29: Examples for problem 29.