Homework 3: Declarative Programming

Due: problems 1–6, Friday, September 28, 2007; problems 7–11, Friday, October 5, 2007; problems 12–24, Friday, October 12, 2007.

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].

Don't use side effects (assignment and cells) in your solutions.

You should use helping functions whenever you you find that helpful. 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 these you must also provide evidence that your program is correct (for example, test cases). Oz code with tests for various problems is available in a zip file, which you can download from the course resources web page. These tests make use of our code in the file `Test.oz`, shown in Figure 2 on page 3, and also available for download from WebCT and the course resources web page. Turn in (on WebCT) your code and the output of your testing for all questions that require code.

Be sure to clearly label what problem each area of code solves with a comment.

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

Read Chapter 3 of the textbook [RH04]. Also read our handout "Following the Grammar." 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.

Iteration

1. (5 points) [UseModels]

   Write a function

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

   that computes the factorial of its argument iteratively. The program described in the book's section 3.3 is not iterative. Your task is to write and test an iterative version.

2. (10 points) [UseModels]

   Do problem 5 in section 3.10 of the textbook [RH04] (iterative SumList). You should make up your own tests for this.

Following the Grammar

3. (20 points) [UseModels]

   For each of the functions in Figure 3 on page 3, 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.)

4. (20 points) [UseModels]

   For each of the functions in Figure 4 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).

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

```
% $Id: Testing.oz,v 1.5 2007/08/19 18:19:10 leavens Exp leavens $
%
% Assertion and testing procedures for Oz.
%
% AUTHOR: Gary T. Leavens

functor $
import
   System(showInfo)
export
   assert: Assert
   assume: Assume
   start: StartTesting
   test: Test
define
   %% Assert that the argument is true.
   proc {Assert B}
      if {Not B}
      then {Exception.raiseError assertionFailed}
      end
   end

   %% Mark an assumption that the argument is true.
   proc {Assume B}
      if {Not B}
      then {Exception.raiseError assumptionFailed}
      end
   end

   %% Print a newline and a message that testing is beginning.
   proc {StartTesting Name}
      {System.showInfo ""}
      {System.showInfo 'Testing ' # Name # '...'}
   end

   %% Test if Actual == Expected.
   %% If so, print a message, otherwise throw an exception.
   proc {Test Actual Connective Expected}
      if Actual == Expected
      then {System.showInfo
            {Value.toVirtualString Actual 5 10}
            # ' ' # Connective # ' '
            # {Value.toVirtualString Expected 5 10}}
      else {Exception.raiseError
                testFailed(actual:Actual
                           connective:Connective
                           expected:Expected
                           debug:unit)
           }
      end
   end
end
```

Figure 1: Testing code that puts output on standard output (the `*Oz Emulator*` window). This functor is available in the course `lib` directory. This can be used in other functors by importing `Testing`.

```
% $Id: Test.oz,v 1.6 2006/09/26 08:37:27 leavens Exp $
% AUTHOR: Gary T. Leavens

declare
local [Testing] = {Module.link ['Testing.ozf']}
in
   StartTesting = Testing.start
   Test = Testing.test
end
```

Figure 2: Testing code that works in the Mozart system's Oz Programming Interface. The module linked is shown in Figure 1 on the preceding page. This file is available in the course `lib` directory To use it, copy the files from the course directory to your own directory and then put `\insert 'Test.oz'` in your file.

(a)
```
fun {TalentsOf People}
   case People of
      P|Ps then {Talent P}|{TalentsOf Ps}
   end
end
```

(b)
```
fun {TalentsOf People}
   case People of
      nil then nil
   end
end
```

(c)
```
fun {TalentsOf People}
   case People of
      P|Ps then {Talent P}|{TalentsOf Ps}
   else nil
   end
end
```

(d)
```
fun {TalentsOf People}
   case People of
      hot then sweltering
   [] warm then happy
   [] cold then freezing
   end
end
```

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

Figure 3: Problem 3.

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

(b)
```
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
```

(c)
```
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
```

(d)
```
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
```

(e)
```
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 4: Problem 4.

5. (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 Figure 2 on page 3.

   ```
   \insert 'Test.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]}
   ```

6. (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 examples are written using the `Test` procedure from Figure 2 on page 3.

   ```
   \insert 'Test.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]}
   ```

   Hint: you may need a helping function.

7. (30 points) [UseModels]

   This is a problem about recursion over flat lists. In this problem you will write procedures to 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 lists.)

   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 WebCT 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 are two complications in the code that we have provided for you. The first is that it the file we give you `SetOps.oz` is written as a functor. See section 3.9.3 in our textbook [RH04] for more about functors. For your initial debugging and testing, you may want to take out all the functor syntax. To do this, comment out everything from the line containing **`functor`** `$` up to and including the line containing **`define`**, and also comment out the last **`end`**; finally, add a **`declare`** at the beginning. After you have done your own debugging and testing, you will have to make the `SetOps.oz` file back into a functor, since our tests assume that your file is a functor. To make it a functor again, comment out the **`declare`** you added, and uncomment the functor syntax you commented out previously. When the file is not a functor, you can feed it to Oz as always, using the "Feed Buffer" item of the Oz menu (and you can't compile it). But when the file is a functor, you must compile it using the "Compile File"

item of the Oz menu or using the command `ozc -c SetOps.oz`. When the file is a functor, you cannot feed it as usual.

The second complication is that the functions won't work until you write something for your 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 a procedure for 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 5 on the next page gives tests that uses these functions.

To start solving this problem, download the file `SetOps.oz` from WebCT to your directory. Note that you must keep the name as `SetOps.oz`. Then add your own code as indicated in the file.

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.

Hint: to incrementally develop the procedures, start with by implementing `Add`. It may be helpful to "stub out" the other functions. See also the notes above for how to make the `SetOps.oz` file not be a functor while doing this development.

Hint: when testing, don't forget to recompile `SetOps.oz` as described above. Also when running our tests, you compile (not feed) `SetOps.oz`, but you feed (but don't compile) `SetOpsTest.oz`.

After doing your own testing, turn your file back into a functor if necessary, and then run our test cases, and include their output (along with output from any of your own tests) and your source code in what you turn in.

```
% $Id: SetOpsTest.oz,v 1.2 2007/09/24 20:53:45 leavens Exp leavens $
local [Testing SetOps] = {Module.link ['Testing.ozf' 'SetOps.ozf']}
   StartTesting = Testing.start   Assert = Testing.assert
   EmptySet = SetOps.emptySet     AsSet = SetOps.asSet
   Size = SetOps.size             Choose = SetOps.choose
   IsMember = SetOps.isMember     IsSubset = SetOps.isSubset
   Equal = SetOps.equal           Add = SetOps.add
   Remove = SetOps.remove         Union = SetOps.union
   Minus = SetOps.minus           Intersect = SetOps.intersect
   UnionList = SetOps.unionList
in
   {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 {AsSet [2 3 1]} 1} {AsSet [3 2]}}}
   {Assert {Equal {Remove {AsSet [2 3 4 8]} 1} {AsSet [2 3 4 8]}}}
   {StartTesting 'Union'}
   {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 {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 {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]}}}
end
```

Figure 5: Tests for exercise 7.

8. (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 6 has some examples that are written using the `Test` procedure from Figure 2 on page 3. Turn in your source code along with output of testing that includes these tests.

```
\insert 'Test.oz'
\insert 'ShrinkTo.oz'
{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)])])}
```

Figure 6: Tests for exercise 8.

9. (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 ⟨Varref⟩. This function evaluates the expression E, using F to determine the values of all ⟨Varref⟩s that occur within it. Examples are shown in Figure 7 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.

10. (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 exercise, you should use your solution to exercise 7 on page 5. (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 exercise.) Due to the use of functors in that code, your file `AllIds.oz` would have the following shape.

    ```
    declare
    local [SetOps] = {Module.link ['SetOps.ozf']}
       AsSet = SetOps.asSet           EmptySet = SetOps.emptySet
       Add = SetOps.add               Union = SetOps.union
       UnionList = SetOps.unionList
    in
       fun {AllIds Stmt}

       % ...
    end
    ```

    Figure 8 on page 11 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.

11. (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 9 on page 11.

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

```
\insert 'Assert.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

{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(notequals(p q)) StdEnv} == true}
{Assert {BEval andExp(comp(notequals(p q))
                      comp(equals(x x))) StdEnv} == true}
{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}
```

Figure 7: Testing for `BEval`, exercise 9.

```
% $Id: AllIdsTest.oz,v 1.1 2007/09/24 22:14:44 leavens Exp leavens $
\insert 'AllIds.oz'
local [Testing SetOps] = {Module.link ['Testing.ozf' 'SetOps.ozf']}
   StartTesting = Testing.start   Assert = Testing.assert
   AsSet = SetOps.asSet           Equal = SetOps.equal
in
   {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]}}}
end
```

Figure 8: Testing for `AllIds`, exercise 10.

```
\insert 'Assert.oz'
{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))))}
```

Figure 9: Tests for the function `SubstIdentifier`, which is exercise 11.

Using Libraries and Higher-Order Functions

12. (15 points) [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) by writing out the recursion yourself, (b) by using the **`for`** loop in Oz (see the Oz documentation or section 3.6.3 of the text [RH04]), and (c) using Oz's built in list functions `Map` and `Filter` (see Section 6.3 of "The Oz Base Environment" [DKS06]).

    You can test by passing each of your functions as an argument to the procedure in Figure 10 on the following page, which is written using the `Test` procedure from Figure 2 on page 3. Figure 10 on the following page also shows how to use the procedure `AssociatedTest` in a way that will work if you name each of your solutions as indicated.

13. [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 11 on the next page gives examples of these written using the procedures from Figure 2 on page 3.

14. (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 12 on page 14 gives some examples.

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

```
\insert 'Test.oz'
\insert 'Associated.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}
```

Figure 10: Test procedure for Exercise 12 and its use.

```
\insert 'Test.oz'
\insert 'Borrowed.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}
```

Figure 11: Examples for exercise 13.

```
\insert 'Test.oz'
\insert 'Compose.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]}
end
{Test {{Compose [fun {$ X} X + 1 end fun {$ X} X + 2 end]} 4}
      '==' 7}
{Test {{Compose [fun {$ X} 3|X end fun {$ Y} 4|Y end]} nil}
      '==' 3|(4|nil)}
```

Figure 12: Examples for exercise 14.

15. [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 are examples.

```
\insert 'Test.oz'
\insert 'IsFunction.oz'
{StartTesting 'IsFunction'}
{Test {IsFunction nil} '==' true}
{Test {IsFunction [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 [b#2 c#3 d#2 e#2 f#2 g#3 a#1]} '==' true}
{Test {IsFunction [bush#shrub]} '==' true}
```

(b) (10 points) Using the **`for`** loop in Oz (see the Oz documentation or section 3.6.3 of the text [RH04]), 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. For example,

```
\insert 'Test.oz'
\insert 'BRelCompose.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]}
```

16. (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 `", "`. For example,

    ```
    \insert 'Test.oz'
    {StartTesting 'CommaSeparate'}
    {Test {CommaSeparate nil} '==' ""}
    {Test {CommaSeparate ["a" "b"]} '==' "a, b"}
    {Test {CommaSeparate ["Monday" "Tuesday" "Wednesday" "Thursday"]}
          '==' "Monday, Tuesday, Wednesday, Thursday"}
    ```

17. (5 points) 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.

    For example,

    ```
    \insert 'Test.oz'
    {StartTesting 'OnSeparateLines'}
    {Test {OnSeparateLines nil} '==' ""}
    {Test {OnSeparateLines ["a" "b"]} '==' "a\nb"}
    {Test {OnSeparateLines ["Monday" "Tuesday" "Wednesday" "Thursday"]}
          '==' "Monday\nTuesday\nWednesday\nThursday"}
    ```

18. (10 points) Define a curried function

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

    that is a generalization of `onSeparateLines` and `commaSeparated`. Test it by using it to define these two other functions.

19. (5 points) [UseModels]

    Define the function `MyAppend` to be just like the standard `Append` function. You definition is to be done by 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
    ```

20. (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. The following are examples.

    ```
    \insert 'Test.oz'
    \insert 'DoubleAll.oz'
    {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]}
    ```

21. (15 points) [UseModels]

Define the function `MyMap` to be just like the standard `Map` function. Your definition is to be done by using `FoldR`. As part of your testing, use `MyMap` to (a) declare `DoubleAll`, and (b) to add 1 to all the elements of a list of Ints.

22. [UseModels]

Consider the following type

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

for nary-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. For example, the procedure shown in Figure 13 on page 19 tests an implementation of `SumTree` passed to it as an argument.

(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.

For example, the procedure shown in Figure 14 on page 20 tests an implementation of `MapTree` passed to it as an argument.

(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. You should, for example, be able to define `SumTree`, and `MapTree` on Trees as follows.

```
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 'SumTreeTest.oz'
{SumTreeTest SumTree}
\insert 'MapTreeTest.oz'
{MapTreeTest MapTree}
```

23. (30 points) [UseModels] [Concepts]

A set 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 set containing all numbers strictly greater than 7. Allowing the user to construct a set from a characteristic function gives one the power to construct 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. (Hint: think about using a function type as the representation of sets.)

(a) The function `SetSuchThat` takes a characteristic function, F and returns a set such that each value X is in the set just when $\{F\ X\}$ is **`true`**.

(b) The function `Union` takes two sets, with characteristic functions F and G, and returns a set such that each value X is in the set just when either $\{F\ X\}$ or $\{G\ X\}$ is **`true`**.

(c) The function `Intersect` takes two sets, with characteristic functions F and G, and returns a set such that each value X is in the set just when both $\{F\ X\}$ and $\{G\ X\}$ are **`true`**.

(d) The function `Member` returns a Boolean that tells whether the second argument is a member of its first argument. (Note that this is *not* the same as Oz's built-in `Member` function, you are to write your own.)

(e) The function `Complement` returns a set that contains everything that is not in its argument set.

As examples, consider the tests in Figure 15 on page 21.

Note (hint, hint) that the equations in Figure 16 on page 21 must hold, for all `F`, `G`, and `X` of appropriate types.

24. (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`. Figure 17 on page 21 gives some examples.

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 17 on page 21. (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.)

References

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

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

```
\insert 'Test.oz'
declare
proc {SumTreeTest SumTree}
   {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}
end
```

Figure 13: Procedure to test exercise 22a.

```
\insert 'Test.oz'
declare
proc {MapTreeTest MapTree}
   fun {Add1 X} X+1 end
   fun {Add3 X} X+3 end
in
   {StartTesting 'MapTree'}
   {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
```

Figure 14: A procedure to test solutions to exercise 22b.

```
\insert 'Test.oz'
\insert 'Set.oz'
{StartTesting 'Set'}
declare
fun {IsCoke X} X == coke end
fun {IsPepsi X} X == pepsi end

{Test {Member {SetSuchThat IsCoke} coke} '==' true}
{Test {Member {SetSuchThat IsCoke} pepsi} '==' false}
{Test {Member {Complement {SetSuchThat IsCoke}} coke} '==' false}
{Test {Member {Union {SetSuchThat IsCoke} {SetSuchThat IsPepsi}}
       pepsi} '==' true}
{Test {Member {Union {SetSuchThat IsCoke} {SetSuchThat IsPepsi}}
       coke} '==' true}
{Test {Member {Union {SetSuchThat IsCoke} {SetSuchThat IsPepsi}}
       sprite} '==' false}
{Test {Member {Intersect {SetSuchThat IsCoke} {SetSuchThat IsPepsi}}
       coke} '==' false}
```

Figure 15: Example tests for exercise 23.

```
{Member {Union (SetSuchThat F} {SetSuchThat G}} X}
    == {F X} orelse {G X}
{Member {Intersect (SetSuchThat F} {SetSuchThat G}} X}
    == {F X} andthen {G X}
{Member {SetSuchThat F} X} == {F X}
{Member {Complement {SetSuchThat F}} X} == {Not {F X}}
```

Figure 16: Equations that give hints for exercise 23.

```
\insert 'Test.oz'
\insert 'TypeOf.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 equalExp(subExp(charLit(&a) 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}
```

Figure 17: Examples for exercise 24.