Homework 4: Declarative Programming

Due: Tuesday, October 3, 2006.

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. Many of the problems below exhibit polymorphism. The problems as a whole illustrate how functional languages work without hidden side-effects. Don't use side effects (assignment and cells) in your solutions.

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). For testing, you may want to use tests based on my code in the file `Assert.oz`, shown in Figure 2 on page 4. Hand in a printout of 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]. 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 "Introduction to the Literature" handout.

Functional Programming

1. (10 points) Write a function

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

   that takes an item of some type `T` and a list of items 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 4.

   ```
   {Test {DeleteAll 3 nil} '=' nil}
   {Test {DeleteAll 1 [1 2 3 2 1 2 3 2 1]} '=' [2 3 2 2 3 2]}
   {Test {DeleteAll 4 [1 2 3 2 1 2 3 2 1]} '=' [1 2 3 2 1 2 3 2 1]}
   {Test {DeleteAll 3 [1 2 3]} '=' [1 2]}
   ```

2. (10 points) Write a function

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

   that takes an item of some type `T` and a list of items 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 4.

   ```
   {Test {DeleteSecond 3 nil} '=' nil}
   {Test {DeleteSecond 1 [1 2 3 2 1 2 3 2 1]} '=' [1 2 3 2 2 3 2 1]}
   {Test {DeleteSecond 4 [1 2 3 2 1 2 3 2 1]} '=' [1 2 3 2 1 2 3 2 1]}
   {Test {DeleteSecond 3 [1 2 3]} '=' [1 2 3]}
   ```

   Hint: you may need a helping function.

3. (15 points) In Oz, write a function

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

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

   Do this (a) by writing out the recursion yourself, (b) by using the **`for`** loop in Oz (see the Oz documentation or section 3.8.3 of the text [RH04]), and (c) using Oz's built in list functions `Map` and `Filter` (see Section 4.3 of "The Oz Base Environment" [DKS06]).

```
% $Id: Testing.oz,v 1.4 2006/09/26 21:36:23 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}
      {Assert B}
   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`.

You can test by passing each of your functions as an argument to the procedure in Figure 3 on the next page, which is written using the Test procedure from Figure 2 on the following page.

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

```
<Database> ::= <List <Pair <Person> <Book>>>
<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 4 on the next page gives examples of these written using the procedures from Figure 2 on the following page.

5. (15 points) 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 5 on page 7 gives some examples.

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

6. Consider the following type as a representation of binary relations.

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

(a) (10 points, extra credit) 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.

```
{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, extra credit) 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,

```
% $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 page 2. 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.

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

Figure 3: Test procedure for Exercise 3.

```
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 4: Examples for exercise 4.

```
{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]}
```

7. (5 points) 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,

   ```
   {Test {CommaSeparate nil} '=' ""}
   {Test {CommaSeparate ["a" "b"]} '=' "a, b"}
   {Test {CommaSeparate ["Monday" "Tuesday" "Wednesday" "Thursday"]}
         '=' "Monday, Tuesday, Wednesday, Thursday"}
   ```

8. (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,

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

9. (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 other functions.

10. (5 points) 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 4.3 of "The Oz Base Environment" [DKS06].)

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

11. (5 points) 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.

    ```
    {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]}
    ```

12. (15 points) Define the function `MyMap` to be just like the standard `Map` function. You 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.

13. 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 6 on the following page 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 7 on page 8 tests an implementation of `MapTree` passed to it as an argument.

(c) (30 points) By generalizing your answers to the above problems, define a 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.

```
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
```

14. (30 points) 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 ϕ such that

$$\phi(\texttt{coke}) = \phi(\texttt{pepsi}) = \texttt{\textbf{true}}$$

and for all other arguments x, $\phi(x) = \texttt{\textbf{false}}$, is the characteristic function for a set containing the strings `coke`, `pepsi` and nothing else. Allowing the user to construct a set from a characteristic function gives one the power to construct sets that may "contain" an infinite number of elements (such as the set of all prime numbers).

Your problem is to implement the following operations. (Hint: think about using a function type.)

(a) The function `SetSuchThat` takes a characteristic function, f and returns a set such that each value x (of appropriate type) 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 (of appropriate type) 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 (of appropriate type) is in the set just when both $\{f\ x\}$ and $\{g\ x\}$ are **`true`**.

```
{Test {{Compose nil} [1 2 3]} '==>' [1 2 3]}
{Test {{Compose [fun {$ X} X + 1 end fun {$ X} X + 2 end]} 4}
      '==>' 7}
local fun {Tail Ls} _|Rest = Ls in Rest end in
   {Test {{Compose [Tail Tail Tail]} [1 2 3 4 5]}
         '==>' [4 5]}
end
{Test {{Compose [fun {$ X} 3|X end fun {$ Y} 4|Y end]} nil}
      '==>' 3|(4|nil)}
```

Figure 5: Examples for exercise 5.

```
declare
proc {SumTreeTest 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 6: Procedure to test exercise 13a.

```
declare
proc {MapTreeTest MapTree}
   fun {Add1 X} X+1 end
   fun {Add3 X} X+3 end
in
   {Test {MapTree node(item:4 subtrees:nil) Add1}
         '=' node(item:5 subtrees:nil)}
   {Test {MapTree node(item:3
                   subtrees:[node(item:4 subtrees:nil)
                             node(item:7 subtrees:nil)])
                 Add3}
         '=' node(item:6
                   subtrees:[node(item:7 subtrees:nil)
                             node(item:10 subtrees:nil)])}
   {Test {MapTree
          node(item:10
               subtrees:[node(item:3
                              subtrees:[node(item:4 subtrees:nil)
                                        node(item:7 subtrees:nil)])
                         node(item:10
                              subtrees:[node(item:20 subtrees: nil)
                                        node(item:30 subtrees: nil)
                                        node(item:40 subtrees: nil)]
                             )])
          Add3}
    '=' node(item:13
             subtrees:[node(item:6
                            subtrees:[node(item:7 subtrees:nil)
                                      node(item:10 subtrees:nil)])
                       node(item:13
                            subtrees:[node(item:23 subtrees: nil)
                                      node(item:33 subtrees: nil)
                                      node(item:43 subtrees: nil)]
                           )])}
end
```

Figure 7: A procedure to test solutions to exercise 13b.

```
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 8: Example tests for exercise 14.

(d) The function `Member` tells whether the second argument is a member of its first argument.

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

As examples, consider the tests in Figure 8 on the preceding page.

Note (hint, hint) that the equations in Figure 9 on the next page must hold, for all `F`, `G`, and `X` of appropriate types.

15. (25 points) 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
```

Write a function

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

that takes an `<Exp>` and returns its `OType`. Figure 10 on the following page gives some examples.

Your program should incorporate a reasonable notion of what the exact type rules are. (Exactly what "reasonable" is left up to you; explain any decisions you feel the need to make.)

Other Problems

16. (50 points total; extra credit) Do the paper review problem at the end of homework 2.

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.

```
{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 9: Equations that give hints for exercise 14.

```
{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 10: Examples for exercise 15.