Fall, 2002 Name: _____

Com S 541 — Programming Languages 1

Test on Functional Programming Languages

Special Directions for this Test

This test has 6 questions and pages numbered 1 through 7.
This test is open book and notes.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
When you write Haskell functions on this test, you may use anything in the Haskell Prelude without writing it in your test. You are encouraged to define functions not specifically asked for if they are useful to your programming; if they are not in the Prelude, write them into your test.
Just for your information the Haskell type `Maybe` is defined in the Prelude as follows:

```
data Maybe a = Nothing | Just a
       deriving (Eq, Ord, Read, Show)
```

1. (10 points) Categorize each of the following expressions in Haskell as either: monomorphic, ad hoc polymorphic, or parameterically polymorphic. Only write one of these phrases for each exprssion! (Note that the type of `toUpper` is `Char -> Char`.)

   (a) `(True, \ x -> not x)`

   (b) `(\ x -> x * x - 4)`

   (c) `(\ f -> \ g -> g . f)`

   (d) `(True, not True)`

   (e) `(map toUpper "a string")`

2. (10 points) Suppose we wanted to include something like AspectJ's "before" advice in Haskell. We might try to do so with a function, `before`, which took a piece of advice as a function, a function to be advised, and then returned an advised function. In short, we want to write a definition of the following type:

   ```
   > before :: (a -> ()) -> (a -> b) -> (a -> b)
   ```

   It is possible to write a function with this type in Haskell. For example, besides functions that return errors, the following has that type:

   ```
   > before advice advisedFun arg = advisedFun arg
   ```

   The question is, is it possible to write a more useful version of `before`? Besides not ignoring the argument `advice`, a more useful version would allow one to do things like tracing and logging (e.g., to an external file). If one can write something that is useful for such purposes, write the code and briefly explain it. If it's not possible to write something that is useful for such purposes, explain why it's not possible. (Hint: think about what a function of type `a -> ()` can do in Haskell.)

3. (10 points) In Haskell, define the function

```
> map2 :: (a -> b -> c) -> [a] -> [b] -> [c]
```

that take a two argument function, `f`, and two lists, `xs` and `ys`, and produces a list of the results of applying `f` to the corresponding elements of `xs` and `ys`. That is, the i^{th} element of the result list is the application of the `f` to the i^{th} element of `xs` and the i^{th} element of `ys` (in that order). The length of the result list should be the length of the shorter of the two argument lists. For example,

```
map2 (-) [] []                          = []
map2 (-) [10, 20, 30] [1, 2, 3]         = [9, 18, 27]
map2 (-) [10, 20, 30] [1, 2, 3, 4, 5] = [9, 18, 27]
map2 (>) [10, 20, 30] [3, 1]            = [True, True]
map2 (\x y -> x*y + 2*y) [4, 7, 9, 18, 54] [3, 1, 10, -2, 0]
     = [18, 9, 110, -40, 0]
```

4. (10 points) In Haskell, we can represent a matrix as a list of row vectors, where each row vector is a list of numbers:

```
> type Vector a = [a]
> type Matrix a = [Vector a]
```

For example, the matrix $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ would be represented as the list `[[1, 2], [3, 4]]`.

Your task is to define the function

```
> matrixAdd :: Num a => Matrix a -> Matrix a -> Matrix a
```

that takes two matricies of the same shape, `m1` and `m2`, and return a matrix in which the (i, j)-th element of the result is the sum of the (i, j)-th element of `m1` and the (i, j)-th element of `m2`. For example:

```
matrixAdd [[1, 2]] [[10, 20]]
        = [[11, 22]]
matrixAdd [[1, 2], [3, 4], [5, 6]] [[10, 20], [30, 40], [100, 200]]
        = [[11, 22], [33, 44], [105, 206]]
matrixAdd [[1, 2, 3, 4], [4, 1, 2, 1]] [[10, 20, 10, -10], [20, 50, 80, 95]]
        = [[11, 22, 13, -6], [24, 51, 82, 96]]
```

You can assume the two matrixes have the same shape, that the lists are all non-empty, and that the number of rows and columns is finite.

5. (15 points) Consider the following data type. (Note that this is different than the trees in the homework.)

```
> data Tree a = Atom a | List [Tree a]
>                 deriving Show
```

Define the function

```
> filterTree :: (a -> Bool) -> Tree a -> Maybe (Tree a)
```

that takes a predicate `pred`, and a tree `t` and returns a `Maybe` that contains the filtered version of the tree `t`. The following are examples.

```
filterTree odd (Atom 3) = Just (Atom 3)
filterTree even (Atom 3) = Nothing
filterTree even (List [])
         = Just (List [])
filterTree even (List [Atom 1, Atom 2, Atom 3])
         = Just (List [Atom 2])
filterTree even (List [Atom 1, Atom 5, Atom 3])
         = Just (List [])
filterTree odd  (List [Atom 1, Atom 2, Atom 3])
         = Just (List [Atom 1, Atom 3])
filterTree odd  (List [List [Atom 1], List [Atom 2, Atom 3], Atom 3])
         = Just (List [List [Atom 1], List [Atom 3], Atom 3])
filterTree even (List [List [Atom 1], List [Atom 2, Atom 3], Atom 3])
         = Just (List [List [], List [Atom 2]])
filterTree isUpper (List [List (map Atom "in June"),
                          List (map Atom "it is Math"),
                          List [Atom 'L']])
         = Just (List [List [Atom 'J'], List [Atom 'M'], List [Atom 'L']])
```

Hint: be careful to make sure your solution type checks!

6. (25 points) Consider the following data and function definitions.

```
> data Value = IntVal Integer | BoolVal Bool | ErrVal

> data Exp =   Lit Value | Variable String
>            | Exp `Subtract` Exp | Exp `Add` Exp | Exp `Multiply` Exp
>            | If Exp Exp Exp

> type Environment = String -> Value
> emptyEnv :: Environment
> emptyEnv = (\ x -> ErrVal)
> bind :: [String] -> [Value] -> Environment -> Environment
> bind [] _ env = env
> bind _ [] env = env
> bind (n:ns) (v:vs) env = (\ x -> if n == x
>                                  then v
>                                  else (bind ns vs env) x)
```

Using these, define a Haskell function

```
> eval :: Exp -> Environment -> Value
```

that takes an expression, `exp`, an environment, `env`, and returns the value of `exp` in `env`. Dynamic type errors should result in `ErrVal` being returned, as shown in some of the examples below.

```
eval (Lit (IntVal 3)) emptyEnv = IntVal 3
eval (Lit (BoolVal False)) emptyEnv = BoolVal False
eval (Lit ErrVal) (\ v -> (IntVal 0)) = ErrVal
eval (Variable "x") emptyEnv = ErrVal
eval (Variable "x") (bind ["x","y"] [IntVal 3, IntVal 4] emptyEnv) = IntVal 3
eval (Variable "y") (bind ["x","y"] [IntVal 3, IntVal 4] emptyEnv) = IntVal 4
eval (Variable "z") (bind ["x","y"] [IntVal 3, IntVal 4] emptyEnv) = ErrVal

eval ((Variable "x") `Subtract` (Variable "y"))
     (bind ["x","y"] [IntVal 3, IntVal 4] emptyEnv)
    = IntVal (-1)

eval (((Variable "x") `Add` (Variable "x")) `Subtract` (Variable "y"))
     (bind ["x","y"] [IntVal 3, IntVal 4] emptyEnv)
    = IntVal 2

eval (((Variable "x") `Multiply` (Variable "y")) `Subtract` (Variable "y"))
     (bind ["x","y"] [IntVal 3, IntVal 4] emptyEnv)
    = IntVal 8

eval (If (Lit ErrVal) (Lit (IntVal 3)) (Lit (IntVal 4))) emptyEnv
    = ErrVal

eval (If (Lit (IntVal 5)) (Lit (IntVal 4)) (Lit (IntVal 1))) emptyEnv
    = ErrVal

eval ((Variable "b") `Subtract` (Lit (IntVal 3)))
     (bind ["b"] [BoolVal True] emptyEnv)
    = ErrVal
```

(There are more examples on the next page...)

```
eval (If (Lit (BoolVal True))
      (((Variable "x") `Multiply` (Variable "y")) `Subtract` (Variable "y"))
      (((Variable "x") `Add` (Variable "y")) `Subtract` (Lit (IntVal 3))))
     (bind ["x","y"] [IntVal 3, IntVal 4]
      (bind ["b"] [BoolVal False] emptyEnv))
    = IntVal 8

eval (If (Lit (BoolVal False))
      (((Variable "x") `Multiply` (Variable "y")) `Subtract` (Variable "y"))
      (((Variable "x") `Add` (Variable "y")) `Subtract` (Lit (IntVal 3))))
     (bind ["x","y"] [IntVal 3, IntVal 4]
      (bind ["b"] [BoolVal False] emptyEnv))
    = IntVal 4

eval ((If (Lit (BoolVal True))
       (If (Variable "b")
        (Variable "y")
        (((Variable "x") `Multiply` (Variable "y")) `Subtract` (Variable "y")))
       (((Variable "x") `Add` (Variable "y")) `Subtract` (Lit (IntVal 3))))
      `Add` (If (Variable "b")
             (Variable "x")
             ((Variable "y") `Subtract` (Variable "x"))))
     (bind ["x","y"] [IntVal 3, IntVal 4]
      (bind ["b"] [BoolVal False] emptyEnv))
    = IntVal 9
```