Spring, 2013 Name: ______________________

(Please *don't* write your id number!)

COP 4020 — Programming Languages I

Test on Higher-Order Functional Programming

Special Directions for this Test

This test has 6 questions and pages numbered 1 through 8.
This test is open book and notes, but no electronics.
If you need more space, use the back of a page. Note when you do that on the front.
Before you begin, please take a moment to look over the entire test so that you can budget your time.
Clarity is important; if your programs are sloppy and hard to read, you may lose some points. Correct syntax also makes a difference for programming questions. Take special care with indentation and capitalization in Haskell.
When you write Haskell code on this test, you may use anything we have mentioned in class that is built-in to Haskell. But unless specifically directed, you should not use imperative features (such as the IO type).
You are encouraged to define helping functions whenever you wish. Note that if you use functions that are not in the standard Haskell Prelude, then you must write them into your test. (That is, your code may not import modules other than the Prelude.)

Hints

If you use functions like `filter`, `map`, and `foldr` whenever possible, then you will have to write less code on the test, which will mean fewer chances for making mistakes and will leave you more time to be careful.

For Grading

Question:	1	2	3	4	5	6	Total
Points:	10	10	10	10	40	20	100
Score:							

1. [Concepts] This is a question about free and bound variable identifiers. Consider the following Haskell expression.

   ```
   (\x -> (\y -> map (f x) lst))
   ```

   (a) (6 points) Write, in set brackets ({ and }), the entire set of variable identifiers that occur free in the above Haskell expression.

   (b) (4 points) Write, in set brackets ({ and }), the entire set of variable identifiers that occur bound in the above Haskell expression.

2. (10 points) [UseModels] Using `foldr`, write the function

   ```
   union :: Eq a => [a] -> [a] -> [a]
   ```

   that takes two lists that have no duplicates, `xs` and `ys`, and returns a list that has all the elements of both `xs` and `ys`, also without any duplicates. The result should contain all the elements of `xs` that are not elements of `ys`, followed by all the elements of `ys`, with the original order of `ys` preserved. (You are, of course, allowed to use Haskell's built-in `elem` and `notElem` predicates.) The following are examples, written using the `Testing` module from the homework.

   ```
   testsInteger :: [TestCase [Integer]]
   testsInteger = [eqTest (union [] []) "==" []
           ,eqTest (union [5,4,3,2] []) "==" [5,4,3,2]
           ,eqTest (union [5,4,3,2,6] [2,3,4,5]) "==" [6,2,3,4,5]
           ,eqTest (union [1 .. 10] [20 .. 9999]) "==" ([1 .. 10] ++ [20 .. 9999])
           ,eqTest (union [1,2,3,4,5,6] [7,6,5,4,3,2,1]) "==" [7,6,5,4,3,2,1]  ]
   testsChar :: [TestCase [Char]]
   testsChar = [eqTest (union ['x','y','z'] ['w','x']) "==" ['y','z','w','x']
               ,eqTest (union [] "string aelofch") "==" "string aelofch"
               ,eqTest (union "gr8 day" "to die") "==" "gr8ayto die"
               ,eqTest (union "flapjack" "flabergstd") "==" "pjckflabergstd"  ]
   ```

   Your solution must be written using `foldr`, so fill in the remainder of the following.

   ```
   union xs ys = foldr
   ```

3. (10 points) [UseModels] In Haskell, write the function:

   ```
   filterInside :: (a -> Bool) -> ([[a]] -> [[a]])
   ```

   that for any type `a`, takes a predicate, `p`, of type `(a -> Bool)` and a list of lists of elements of type `a`, `ll`, and returns a list of lists of elements of type `a` that is just like `ll`, except that the inner lists only contain the elements that satisfy `p`.

   The following are examples, written using the `Testing` module from the homework.

```
testsInteger :: [TestCase [[Integer]]]
testsInteger = [eqTest (filterInside (>2) []) "==" []
        ,eqTest (filterInside (>5) [[]]) "==" [[]]
        ,eqTest (filterInside (>5) [[1,2,3,4,5,6,7],[0,9,11]])
         "==" [[6,7],[9,11]]
        ,eqTest (filterInside (\x -> x < 4 || x > 7)
              [[3,9,86,99,6,7,8,7,1],[0,4,9,11],[],[52,3,8,-4]])
         "==" [[3,9,86,99,8,1],[0,9,11],[],[52,3,8,-4]]
        ]
testsChar :: [TestCase [[Char]]]
testsChar = [eqTest (filterInside (>'c') []) "==" []
        ,eqTest (filterInside (>'a') [[]]) "==" [[]]
        ,eqTest (filterInside (>'a') [['a','l','p','h','a'],"beta"])
         "==" ["lph","bet"]
        ,eqTest (filterInside (\c -> c /= 'a' && c /= 'e' && c /= 'i')
              ["blanched cherry","pie","plain pannini","pure","pita"])
         "==" ["blnchd chrry","p","pln pnnn","pur","pt"]
        ]
```

4. (10 points) [UseModels] Using `map`, write the polymorphic function

   ```
   applyList :: [(a -> b)] -> a -> [b]
   ```

   which takes a list of functions `fs`, of type `(a -> b)`, and a (single) value `x` of type `a`, and returns a list of values, of type `b` that result from applying each element of `fs`, in order, to `x`. The order of the result corresponds to the order of the functions in `fs`. The following are examples, written using the `Testing` module from the homework.

   ```
   testsInteger :: [TestCase [Integer]]
   testsInteger = [eqTest (applyList [] 3) "==" []
           ,eqTest (applyList [(+2), (*10), (*0)] 4) "==" [6,40,0]
           ,eqTest (applyList [(\x -> 3*x*x + 2*x +1)] 10) "==" [321]
           ,eqTest (applyList [(*23),(+4),(\x -> 3*x*x + 2*x +1)] 10)
                   "==" [230,14,321]
           ]
   testsBool :: [TestCase [Bool]]
   testsBool = [eqTest (applyList [] 5) "==" []
           ,eqTest (applyList [(==2), (/=10), (==7)] 7) "==" [False,True,True]
           ,eqTest (applyList [(== 'c'),(== 'd'),(=='x')] 'd')
                   "==" [False,True,False]
           ,eqTest (applyList [(\x -> x > 5 || x < 7),(\x -> 3*x < 50)] 10)
            "==" [True,True]
           ]
   ```

   Write your solution, using `map`, by completing the definition below (writing the rest of your code after what is there).

   ```
   applyList fs x = map
   ```

5. (40 points) [UseModels] In this problem you are going to complete the definition of an abstract datatype for Shapes. This will be done in a module that starts as follows.

```
module Shape (Shape, Point, circle, rectangle, inside, join) where
type Point = (Double,Double)
data Shape = Blob (Point -> Bool)
circle :: Point -> Double -> Shape
rectangle :: Point -> Point -> Shape
join :: Shape -> Shape -> Shape
inside :: Shape -> Point -> Bool
-- distance is a helping function we're giving you to use
distance :: (Double,Double) -> (Double,Double) -> Double
distance (x1,y1) (x2,y2) = sqrt ((x2-x1)^2 + (y2-y1)^2)
```

In the above code, `Point` is just a pair of `Double`s. Values of type `Shape` are represented by `(Blob p)`, where `p` is a function that takes a `Point` and tells whether that Point is in the shape. Your task is to write, in Haskell, implementations for the functions `circle`, `rectangle`, `inside`, and `join`, whose types are given above.

- The `circle` function takes a Point `(cx,cy)` (2 Doubles representing the center of the circle), and a Double `radius`, and returns a Shape such that points are inside it just when their distance from `(cx,cy)` is less than or equal to `radius`.
- The `rectangle` function takes two points `(tlx,tly)` and `(brx,bry)`, which specify the top left and bottom right coordinates of the rectangle, and returns a Shape such that `(x,y)` points is inside it just when `tlx` $\leq$ `x` $\leq$ `brx` and `bry` $\leq$ `y` $\leq$ `tly`.
- The `join` function takes two Shapes and produces a Shape that is their union; that is, a Point is inside the join of two Shapes just when it is inside either of the shapes individually.
- The `inside` function tells when a point is inside the shape.

The following are examples, written using the `Testing` module from the homework.

```
tests :: [TestCase Bool]
tests =
    let cr = (circle (10.0,10.0) 4.0)
        rt = (rectangle (100.0,104.0) (105.0,100.0))
        crt = cr `join` rt
    in [assertTrue (inside (circle (0.0,0.0) 2.0) (0.0,0.0))
       ,assertTrue (inside (circle (1.0,0.0) 3.0) (2.0,2.0))
       ,assertTrue (inside cr (14.0,10.0))
       ,assertTrue (inside cr (10.0,14.0))
       ,assertTrue (inside cr (6.0,10.0))
       ,assertTrue (inside cr (10.0,6.0))
       ,assertFalse (inside cr (14.0,14.0))
       ,assertTrue (inside (rectangle (0.0,4.0) (5.0,0.0)) (2.5,2.0))
       ,assertTrue (inside rt (100.0,100.0))
       ,assertTrue (inside rt (100.0,104.0))
       ,assertTrue (inside rt (105.0,104.0))
       ,assertTrue (inside rt (105.0,100.0))
       ,assertFalse (inside rt (105.1,104.1))
       ,assertFalse (inside rt (105.1,104.1))
       ,assertTrue (inside ((rectangle (0.0,10.0) (12.0,0.0))
                          `join` (rectangle (0.0,20.0) (24.0,0.0)))
                   (13.0,12.5))
       ,assertTrue (inside crt (10.0,10.0))
       ,assertTrue (inside crt (10.0,14.0))
```

Please write your answer on the next page.

Your answer for the Shape problem should be written below the start of the module `Shape`, which is repeated here for your convenience.

```
module Shape (Shape, Point, circle, rectangle, inside, join) where
type Point = (Double,Double)
data Shape = Blob (Point -> Bool)
circle :: Point -> Double -> Shape
rectangle :: Point -> Point -> Shape
join :: Shape -> Shape -> Shape
inside :: Shape -> Point -> Bool
-- distance is a helping function we're giving you to use
distance :: (Double,Double) -> (Double,Double) -> Double
distance (x1,y1) (x2,y2) = sqrt ((x2-x1)^2 + (y2-y1)^2)
```

6. (20 points) [Concepts] [UseModels] Consider the following data definition.

```
data BTree a = Value a
             | Node (BTree a) a (BTree a)
               deriving (Eq, Show)
```

Your task in this problem is to write a functional abstraction of code that works over this grammar. As examples to generalize from, consider the following. (Note: you are not supposed to write these functions, but generalize them.)

```
total :: Num a => BTree a -> a
total (Value x) = x
total (Node left x right) = (total left) + x + (total right)

inorder :: BTree a -> [a]
inorder (Value x) = [x]
inorder (Node left x right) = (inorder left) ++ (x : inorder right)

preorder :: BTree a -> [a]
preorder (Value x) = [x]
preorder (Node left x right) = x : (preorder left ++ preorder right)

tmap :: (a -> b) -> BTree a -> BTree b
tmap f (Value x) = Value (f x)
tmap f (Node left x right) = Node (tmap f left) (f x) (tmap f right)
```

Your task in this problem is to write a functional abstraction of the above examples:

```
foldBTree :: (a -> r) -> (r -> a -> r -> r) -> (BTree a) -> r
```

The polymorphic function `foldBTree` takes 2 functions, call them `vf` and `nf`, and a BTree `bt`, and returns a result which is of the type that is the result type of the functions `vf` and `nf` (written `r` in the declaration).

The `vf` function handles BTrees of the form `(Value x)`.

The `nf` function handles BTrees of the form `(Node left x right)`.

Your task is to write `foldBTree` so that it can be used as in the examples below, which continue onto the next page.

```
-- The following definitions are just for use in testing, not for you to write!
total :: Num a => BTree a -> a
total = foldBTree id (\lr x rr -> lr + x + rr)
inorder :: BTree a -> [a]
inorder bt = foldBTree (\x -> [x]) (\lr x rr -> lr ++ (x : rr)) bt
preorder :: BTree a -> [a]
preorder = foldBTree (\x -> [x]) (\lr x rr -> x : (lr ++ rr))
tmap :: (a -> b) -> BTree a -> BTree b
tmap f = foldBTree (\x -> Value (f x)) (\lr x rr -> (Node lr (f x) rr))
-- End of example definitions for help in testing, start of test cases
testsInteger :: [TestCase Integer]
testsInteger =
    [eqTest (total (Value 7)) "==" 7
    ,eqTest (total (Node (Value 3) 4 (Value 5))) "==" 12
    ,eqTest (total (Node (Node (Value 1) 2 (Value 3)) 4 (Value 5))) "==" 15
    ,eqTest (total (Node (Node (Value 1) 2 (Value 3)) 4
                  (Node (Value 5) 100 (Value 200))))
            "==" 315 ]
```

There are more tests on the next page.

```
testsBool :: [TestCase Bool]
testsBool =
    [assertTrue ((inorder (Value 7)) == [7])
    ,assertTrue ((inorder (Node (Value 3) 4 (Value 5))) == [3,4,5])
    ,assertTrue ((inorder (Node (Node (Value 1) 2 (Value 3)) 4 (Value 5)))
                 == [1,2,3,4,5])
    ,assertTrue ((inorder (Node (Node (Value 'h') 'e' (Value 'l')) 'l'
                   (Node (Value 'o') '!' (Value '!'))))
                 == "hello!!")
    ,assertTrue ((preorder (Value 7)) == [7])
    ,assertTrue ((preorder (Node (Value 3) 4 (Value 5))) == [4,3,5])
    ,assertTrue ((preorder (Node (Node (Value 1) 2 (Value 3)) 4 (Value 5)))
                 == [4,2,1,3,5])
    ,assertTrue ((preorder (Node (Node (Value 'h') 'e' (Value 'l')) 'l'
                   (Node (Value 'o') '!' (Value '!'))))
                 == "lehl!o!")
    ,assertTrue ((tmap (+1) (Value 7)) == (Value 8))
    ,assertTrue ((tmap (*2) (Node (Value 3) 4 (Value 5)))
                 == (Node (Value 6) 8 (Value 10)))
    ,assertTrue ((tmap (==5) (Node (Node (Value 1) 2 (Value 3)) 4 (Value 5)))
                 == (Node (Node (Value False) False (Value False)) False (Value True)))
    ,assertTrue ((tmap (\c -> c:[])
                  (Node (Node (Value 'h') 'e' (Value 'l')) 'l'
                   (Node (Value 'o') '!' (Value '!'))))
                 == (Node (Node (Value "h") "e" (Value "l")) "l"
                            (Node (Value "o") "!" (Value "!"))))    ]
```