Fall, 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 8 questions and pages numbered 1 through 9.
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`, `concatMap`, 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 | 7 | 8 | Total |
|---|---|---|---|---|---|---|---|---|---|
| Points: | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 30 | 100 |
| Score: | | | | | | | | | |

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

```
(g (\g -> (\a -> (\b -> (length (g b))))))
```

(a) (5 points) Write, in set brackets ({ and }), the entire set of variable identifiers that occur free in the above Haskell expression.

(b) (5 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

```
sumSquares :: (Num t) => [t] -> t
```

that takes a list of numbers `ns` and returns the sum of the squares of the elements in `ns`. The following are examples, written using the `Testing` module from the homework.

```
tests :: [TestCase Integer]
tests =
    [eqTest (sumSquares []) "==" 0
    ,eqTest (sumSquares [10]) "==" 100
    ,eqTest (sumSquares [5,10]) "==" 125
    ,eqTest (sumSquares [1 .. 10]) "==" 385
    ,eqTest (sumSquares [1 .. 1000]) "==" 333833500
    ,eqTest (sumSquares [-55 .. 55]) "==" 113960
    ,eqTest (sumSquares [99 .. 999]) "==" 332514951
    ,eqTest (sumSquares ([-55 .. 55] ++ [99 .. 999])) "==" 332628911  ]
```

Your solution must use `foldr` in an essential way, so write it by filling in the remainder of the following. You must not use explicit recursion or a list comprehension in your solution. Your solution is also *not* allowed to use `map` or `sum`.

```
sumSquares ns = foldr
```

3. (10 points) [UseModels] Using `foldr`, write the function:

```
frfilter :: (t -> Bool) -> [t] -> [t]
```

that for any type `t`, takes a predicate, `p`, of type `(t -> Bool)` and a list of elements of type `t`, `lst`, and returns a list of elements of type `t` that is just like `lst`, except that it only contains the elements that satisfy `p`. (Thus elements that do not satisfy `p` are not in the result.)

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

```
tests_int :: [TestCase [Integer]]
tests_int =
    [eqTest (frfilter (> 0) []) "==" []
    ,eqTest (frfilter (> 0) [-1]) "==" []
    ,eqTest (frfilter (> 0) [5,-1]) "==" [5]
    ,eqTest (frfilter (> 2) [99,5,-1,86]) "==" [99,5,86]
    ,eqTest (frfilter (>= 6) [99,5,-1,86]) "==" [99,86]
    ,eqTest (frfilter (<= 6) [1 .. 10000000]) "==" [1 .. 6]
    ,eqTest (frfilter (\n -> (n-5) < -7 || (n+5) > 7) [-55 .. 55])
     "==" ([-55 .. -3] ++ [3 .. 55])  ]
tests_char :: [TestCase [Char]]
tests_char =
    [eqTest (frfilter (== 'a') []) "==" []
    ,eqTest (frfilter (> 'm') "xylophone") "==" "xyopon"
    ,eqTest (frfilter (< 'm') "florida") "==" "flida"
    ,eqTest (frfilter (\c -> c == 'f' || c == 'l') "florida") "==" "fl"  ]
```

Your solution must use `foldr` in an essential way, so write it by filling in the remainder of the following. You must not use explicit recursion or a list comprehension in your solution. Your solution is also not allowed to use `filter`.

```
frfilter p lst = foldr
```

4. (10 points) [UseModels] Consider the following Haskell type definitions

```
type Address = [String]   -- 6 elements always
type NewAddress = [String] -- 5 elements always
```

Your task is to write a function

```
joinNames :: [Address] -> [NewAddress]
```

that takes a list of type `Address`, `adrdb`, and returns a list of type `NewAddress` that is the same as `adrdb`, except that the first two elements of each `Address` are to be concatenated, with a blank in between them, in the result. The following are examples, written using the `Testing` module from the homework.

```
tests :: [TestCase [NewAddress]]
tests =
    [eqTest (joinNames []) "==" []
    ,eqTest (joinNames [["Frank", "Wright", "6000 Left", "Chicago", "IL", "USA"]])
     "==" [["Frank Wright", "6000 Left", "Chicago", "IL", "USA"]]
    ,eqTest (joinNames [["Christopher", "Wren", "2000 Hyde", "London", "England", "UK"]
                       ,["Frank", "Wright", "6000 Left", "Chicago", "IL", "USA"]])
     "==" [["Christopher Wren", "2000 Hyde", "London", "England", "UK"]
          ,["Frank Wright", "6000 Left", "Chicago", "IL", "USA"]]
    ,eqTest (joinNames [["Caroline", "Herschel", "700000", "Starry Way", "Hannover", "Germany"]
                       ,["Marie", "Curie", "1 rue Pierre et Marie Curie", "6me", "Paris", "France"]
                       ,["Agusta Ada", "King", "Marylebone", "London", "England", "UK"]])
     "==" [["Caroline Herschel", "700000", "Starry Way", "Hannover", "Germany"]
          ,["Marie Curie", "1 rue Pierre et Marie Curie", "6me", "Paris", "France"]
          ,["Agusta Ada King", "Marylebone", "London", "England", "UK"]]   ]
```

5. (10 points) [UseModels] Consider the following data type definition.

```
data NWayTree t = Node t [NWayTree t] deriving (Eq, Show)
```

Write the function

```
preOrderNWay :: (NWayTree t) -> [t]
```

which takes an NWayTree, nwt, and returns a list of all the elements in nwt ordered so that the value (of type t) in a node precedes all the values in the subtrees contained in the list in that node, and so that values in a subtree that appears in an earlier list in a node precede all those that appear later in that list. The following are examples.

```
tests :: [TestCase [Int]]
tests =
    [eqTest (preOrderNWay (Node 3 [])) "==" [3]
    ,eqTest (preOrderNWay (Node 3 [(Node 7 []), (Node 9 []), (Node 6 [])])) "==" [3,7,9,6]
    ,eqTest (preOrderNWay (Node 3 [(Node 7 [(Node 8 [])]), (Node 9 [])
                              ,(Node 6 [(Node 5 []), (Node 6 [])])]))
     "==" [3,7,8,9,6,5,6]
    ,eqTest (preOrderNWay (Node 10 [(Node 1 [(Node 2 [(Node 3 [(Node 4 [])])])])
                               ,(Node 5 [(Node 6 [])])
                               ,(Node 7 [(Node 8 [(Node 9 [(Node 11 [])])])])
                               ,(Node 12 [(Node 13 [])])]))
     "==" [10,1,2,3,4,5,6,7,8,9,11,12,13]  ]
```

6. (10 points) This problem also uses type `NWayTree`, defined as:

```
data NWayTree t = Node t [NWayTree t] deriving (Eq, Show)
```

Write the function

```
scaleNWay :: (Num t) => t -> (NWayTree t) -> (NWayTree t)
```

that, for any numeric type `t`, takes a value `x` of type `t` and an `(NWayTree t)`, `nwt`, and returns an `(NWayTree t)` that is just like `nwt` except that each number in the result is `x` times the corresponding number in `nwt`. The following are examples.

```
tests :: [TestCase (NWayTree Int)]
tests =
    [eqTest (scaleNWay 5 (Node 3 [])) "==" (Node 15 [])
    ,eqTest (scaleNWay 5 (Node 3 [])) "==" (Node 15 [])
    ,eqTest (scaleNWay 10 (Node 3 [(Node 7 []), (Node 9 []), (Node 6 [])]))
     "==" (Node 30 [(Node 70 []), (Node 90 []), (Node 60 [])])
    ,eqTest (scaleNWay 0 (Node 3 [(Node 7 []), (Node 9 []), (Node 6 [])]))
     "==" (Node 0 [(Node 0 []), (Node 0 []), (Node 0 [])])
    ,eqTest (scaleNWay 4 (Node 3 [(Node 7 [(Node 8 [])]), (Node 9 [])
                                 ,(Node 6 [(Node 5 []), (Node 6 [])])]))
     "==" (Node 12 [(Node 28 [(Node 32 [])]), (Node 36 [])
                   ,(Node 24 [(Node 20 []), (Node 24 [])])])
    ,eqTest (scaleNWay 10 (Node 10 [(Node 1 [(Node 2 [(Node 3 [(Node 4 [])])])])
                                   ,(Node 5 [(Node 6 [])])
                                   ,(Node 7 [(Node 8 [(Node 9 [(Node 11 [])])])])
                                   ,(Node 12 [(Node 13 [])])]))
     "==" (Node 100 [(Node 10 [(Node 20 [(Node 30 [(Node 40 [])])])])
                    ,(Node 50 [(Node 60 [])])
                    ,(Node 70 [(Node 80 [(Node 90 [(Node 110 [])])])])
                    ,(Node 120 [(Node 130 [])])])
```

7. (10 points) [Concepts] [UseModels] Suppose we want to generalize the previous two problems involving the type `NWayTree`, which is defined as follows.

```
data NWayTree t = Node t [NWayTree t] deriving (Eq, Show)
```

That is, suppose we want to have a function:

```
foldNWayTree :: (t -> [r] -> r) -> (NWayTree t) -> r
```

This function should be such that, for any type `t` and desired result type `r`, this function should take a function `f` (of type `(t -> [r] -> r)`) and an `(NWayTree t)`, and return a value of type `r`. The function `f` should be applied to each node's value and to a list of the answers returned by recursing on all subtrees of that node. The following are test cases.

```
tests :: [TestCase Bool]
tests =
    [assertTrue ((foldNWayTree (\n res -> Node (n+1) res) (Node 3 [Node 7 []]))
                 == (Node 4 [Node 8 []]))
    ,assertTrue ((foldNWayTree (\v res -> v:(concat res)) sampleNT)
                 == [10,3,1,2,5,4,6])
    ,assertTrue ((foldNWayTree (\v res -> (reverse (v:(concat res)))) sampleNT)
                 == [5,6,4,3,1,2,10])
    ,assertTrue ((foldNWayTree (\n res -> n + (sum res)) sampleNT) == 31)
    ]
    where sampleNT = (Node 10 [(Node 3 [(Node 1 []), (Node 2 [])])
                              ,(Node 5 [(Node 4 [(Node 6 [])])])])
```

Your task in this problem is to choose which one of the following is a declaration that correctly implements `foldNWayTree`. The correct implementation should have the type and behavior described above and satisfy the test cases given above. (So don't ask us why some choice has a type error or is incorrect during the test — it's because it is the wrong answer!) Circle the letter of the correct choice.

A. `foldNWayTree f (Node v trees) = (foldr f v (map (foldNWayTree f) trees))`

B. `foldNWayTree f (Node v trees) = (map f trees)`

C. `foldNWayTree f (Node v trees) = foldNWayTree f v (f v trees)`

D. `foldNWayTree f (Node v trees) = f v (map (foldNWayTree f) trees)`

E. `foldNWayTree f (Node v trees) = (f v trees)`

F. `foldNWayTree f (Node v trees) = (Node v (map f trees))`

G. `foldNWayTree f (Node v trees) = v ++ (concatMap f trees)`

H. `foldNWayTree f (Node v trees) = v:(concatMap f trees)`

I. None of the above purported solutions are correct.

8. (30 points) [UseModels] [Concepts] In this problem you will implement an abstract data type `Drawing`. Abstractly, a `Drawing` is a mapping from values on the unit interval (real numbers between 0 and 1, inclusive) to points. Think of the unit interval as representing time, with 0 the beginning of the drawing, and 1 the end; thus as time progresses from 0 to 1, the drawing is traced on a canvas, which is modeled by the unit square (points whose x and y coordinates are between 0 and 1, inclusive). The resolution (detail) of the drawing is unlimited, since the unit interval can be sampled as finely as the resolution desired demands. The types `UnitInterval` and `Point` are used in Haskell for the concepts of the unit interval and the points in the drawing.

```
type UnitInterval = Float -- 0.0 to 1.0 inclusive
type Point = (Float, Float)
```

In this problem you will do the following:

1. Decide on a representation for the type `Drawing` and, use it in an implementation of the following 3 functions.
2. The function

   ```
   makeDrawing :: (UnitInterval -> Point) -> Drawing
   ```

   takes a function `f` that maps the unit interval to Points, and returns a `Drawing` such that `(get (makeDrawing f)` $v$`)` has the value of `f` applied to $v$, for all $v$ in the unit interval.
3. The function

   ```
   get :: Drawing -> UnitInterval -> Point
   ```

   takes a `Drawing` and a value in the unit interval and returns the point that the `Drawing` maps that value to.
4. The function

   ```
   flipH :: Drawing -> Drawing
   ```

   takes a `Drawing`, `d`, and returns its reflection through the vertical line $y = 0.5$. That is if for a value $v$ in the unit interval, if `(get d` $v$`)` is $(x, y)$, then `(get (flipH d)` $v$`)` is $(1 - x, y)$.

There are test cases in the following.

```
-- withinPt tests if two points are approximately equal
withinPt :: Point -> String -> Point -> TestCase Point
withinPt = gTest (\(ax,ay) (ex,ey) -> (ax ~=~ ex && ay ~=~ ey))
tests :: [TestCase Point]
tests =
    let -- functions for use in testing, not for you to implement!
        linearup x = (x,x)   -- picture looks like: /
        half x = (x,0.5)     -- picture looks like: --
        lineardown x = (x,1.0-x) -- picture looks like: \
        h x = if x <= 0.3 then (0.3,1-(10/3)*x) -- pic like: H
              else if 0.3 < x && x < 0.6 then (x, 0.5)
                   else (0.6, (10/4)*(x-0.6))
    in [withinPt (get (makeDrawing linearup) 0.0) "~=~" (0.0,0.0)
       ,withinPt (get (makeDrawing linearup) 0.5) "~=~" (0.5,0.5)
       ,withinPt (get (makeDrawing linearup) 1.0) "~=~" (1.0,1.0)
       ,withinPt (get (flipH (makeDrawing linearup)) 0.1) "~=~" (0.9,0.1)
       ,withinPt (get (flipH (makeDrawing linearup)) 0.4) "~=~" (0.6,0.4)
       ,withinPt (get (flipH (makeDrawing linearup)) 1.0) "~=~" (0.0,1.0)
       ,withinPt (get (makeDrawing lineardown) 0.0) "~=~" (0.0,1.0)
       ,withinPt (get (makeDrawing lineardown) 0.5) "~=~" (0.5,0.5)
       ,withinPt (get (makeDrawing lineardown) 1.0) "~=~" (1.0,0.0)
       ,withinPt (get (flipH (makeDrawing lineardown)) 0.0) "~=~" (1.0,1.0)
       ,withinPt (get (flipH (makeDrawing lineardown)) 0.4) "~=~" (0.6,0.6)
```

```
      ,withinPt (get (flipH (makeDrawing lineardown)) 1.0) "~=~" (0.0,0.0)
      ,withinPt (get (makeDrawing half) 0.0) "~=~" (0.0,0.5)
      ,withinPt (get (makeDrawing half) 0.5) "~=~" (0.5,0.5)
      ,withinPt (get (makeDrawing half) 1.0) "~=~" (1.0,0.5)
      ,withinPt (get (makeDrawing h) 0.0) "~=~" (0.3,1.0)
      ,withinPt (get (makeDrawing h) 0.15) "~=~" (0.3,0.5)
      ,withinPt (get (makeDrawing h) 0.3) "~=~" (0.3,0.0)
      ,withinPt (get (makeDrawing h) 0.35) "~=~" (0.35,0.5)
      ,withinPt (get (makeDrawing h) 0.55) "~=~" (0.55,0.5)
      ,withinPt (get (makeDrawing h) 0.6) "~=~" (0.6,0.0)
      ,withinPt (get (makeDrawing h) 0.8) "~=~" (0.6,0.5)
      ,withinPt (get (makeDrawing h) 1.0) "~=~" (0.6,1.0)      ]
```

Complete the implementation of the module `Drawing` below.

```
module Drawing (UnitInterval, Point, Drawing, makeDrawing, get, flipH) where
type UnitInterval = Float -- 0.0 to 1.0 inclusive
type Point = (Float, Float)

-- complete the following data definition:
data Drawing =

-- implement these 3 functions below
makeDrawing :: (UnitInterval -> Point) -> Drawing
get :: Drawing -> UnitInterval -> Point
flipH :: Drawing -> Drawing
```