Spring, 2013 Name: _____

(Please *do not* write your id number!)

COP 4020 — Programming Languages I

# Test on Haskell and Functional Programming

## Special Directions for this Test

This test has 7 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 functions not specifically asked for if they are useful to your programming; however, if they 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. The problem will note explicitly if you are prohibited from using such functions, but by default you can.
In the "follow the grammar" problems the examples may be very extensive and take a long time to read if you read every detail. But the basic idea of the recursion is usually clear from the description, simple examples, and the "follow the grammar" idea of recursing everywhere possible. So look to the examples to confirm your understanding and don't spend too much time reading examples after you understand the problem.

### For Grading

| Question: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | Total |
|---|---|---|---|---|---|---|---|---|
| Points: | 10 | 5 | 10 | 15 | 15 | 20 | 25 | 100 |
| Score: | | | | | | | | |

1. (10 points) [UseModels] In Haskell, write the function:

   ```
   squareEvens :: [Integer] -> [Integer]
   ```

   that takes a list of Integers, `lst`, and returns a list of Integers that is just like `lst`, except that each even element of `lst` is replaced by the square of that element. In your solution, you might find it helpful to use the built-in predicate even.

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

   ```
   tests :: [TestCase [Integer]]
   tests = [eqTest (squareEvens []) "==" []
           ,eqTest (squareEvens [3]) "==" [3]
           ,eqTest (squareEvens [4]) "==" [16]
           ,eqTest (squareEvens [4,3]) "==" [16,3]
           ,eqTest (squareEvens [1,2,3,4,5,6]) "==" [1,4,3,16,5,36]
           ,eqTest (squareEvens [3,22,3,95,600,0,-2]) "==" [3,484,3,95,360000,0,4]
   ```

2. (5 points) [Concepts] [UseModels] Consider the data type `Amount` defined below.

```
data Amount = Zero | One | Two
```

In Haskell, write the polymorphic function

```
rotate :: Amount -> (a,a,a) -> (a,a,a)
```

which takes an Amount, `amt`, and a triple of elements of some type, `(x,y,z)`, and returns a triple that is circularly rotated to the right by the number of steps indicated by the English word that corresponds to `amt`. That is, when `amt` is `Zero`, then `(x,y,z)` is returned unchanged; when `amt` is `One`, then `(z,x,y)` is returned; finally, when `amt` is `Two`, then `(y,z,x)` is returned. The following are examples, written using the `Testing` module from the homework.

```
tests :: [TestCase Bool]
tests =
    [assertTrue ((rotate Zero (1,2,3)) == (1,2,3))
    ,assertTrue ((rotate One (1,2,3)) == (3,1,2))
    ,assertTrue ((rotate Two (1,2,3)) == (2,3,1))
    ,assertTrue ((rotate Two ("jan","feb","mar")) == ("feb","mar","jan"))
    ,assertTrue ((rotate One ("jan","feb","mar")) == ("mar","jan","feb"))
    ,assertTrue ((rotate Zero (True,False,True)) == (True,False,True)) ]
```

3. (10 points) [UseModels] Consider the type of address book entries below.

```
data Entry = Record {name :: String, phone :: Integer, email :: String}
```

Write, in Haskell, the function

```
nophones :: [Entry] -> [(String,String)]
```

that takes a list of records of type `Entry`, `es`, and returns a list of pairs of the name and email address of each entry, in the same order as in `es`. The following are examples, written using the `Testing` module from the homework.

```
tests :: [TestCase [(String,String)]]
tests =
    [eqTest (nophones []) "==" []
    ,eqTest (nophones [Record {name = "Eastman", phone = 3214442211,
                               email = "kodak@k.com"}])
                "==" [("Eastman","kodak@k.com")]
    ,eqTest (nophones [Record {name = "M", phone = 44153543221, email = "m@mi6.uk"}
                      ,Record {name = "Bond", phone = 44007007007, email ="jb@mi6.uk"}])
     "==" [("M","m@mi6.uk"),("Bond","jb@mi6.uk")]
    ,eqTest (nophones sample)
     "==" [("Adams","adams@mail.com"),("Bullfinch","bf@bf.com")
          ,("Cassidy","cass@mail.com"),("Durham","bull@dingers.com"),("Eastman","kodak@k.com")]
    ,eqTest (nophones ((Record {name="Durham",phone=3059123344,
                                email="crash@yahoo.com"}):sample))
     "==" [("Durham","crash@yahoo.com"),("Adams","adams@mail.com"),("Bullfinch","bf@bf.com")
          ,("Cassidy","cass@mail.com"),("Durham","bull@dingers.com"),("Eastman","kodak@k.com")] ]
    where sample =
              [Record {name = "Adams", phone = 4075551212, email = "adams@mail.com"}
              ,Record {name = "Bullfinch", phone = 5155551212, email = "bf@bf.com"}
              ,Record {name = "Cassidy", phone = 8005551122, email = "cass@mail.com"}
              ,Record {name = "Durham", phone = 3059123344, email = "bull@dingers.com"}
              ,Record {name = "Eastman", phone = 3214442211, email = "kodak@k.com"}]
```

4. (15 points) [Concepts] [UseModels] Without using any functions from the Haskell prelude, write the polymorphic function

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

   which takes a predicate, `p`, a function `f`, and a list `lst`, and returns a list that is just like `lst`, but in which every element `x` that does *not* satisfy `p` is replaced by `f` applied to `x`. (An element `x` satisfies `p` if `(p x) == True`.) The following are examples, written using the `Testing` module from the homework.

   ```
   tests :: [TestCase Bool]
   tests =
       [assertTrue ((unpickedMap odd (\x -> x*x) []) == [])
       ,assertTrue ((unpickedMap odd (\x -> 66) [2,4,6,8,10]) == [66,66,66,66,66])
       ,assertTrue ((unpickedMap odd (\x -> x*x) [1,2,3,4,5]) == [1,4,3,16,5])
       ,assertTrue ((unpickedMap even (\x -> 6000) [3,1,-5]) == [6000,6000,6000])
       ,assertTrue ((unpickedMap (\c -> c == 'q') (\x -> 'u') "quip") == "quuu")
       ,assertTrue ((unpickedMap (== False) not [True,False,True]) == [False,False,False]) ]
   ```

5. (15 points) [Concepts] [UseModels] Without using any functions from the Haskell prelude, and without using a list comprehension, write the polymorphic function

   ```
   partition :: (a -> Bool) -> [a] -> ([a],[a])
   ```

   which takes a predicate, `p`, and a list `lst`, and returns a pair of lists `(no,yes)` such that `no` contains the elements of `lst` that do *not* satisfy `p`, and `yes` contains the elements of `lst` that satisfy `p`. In both `no` and `yes` the order of elements is the same as that in `lst`. The following are examples, written using the `Testing` module from the homework.

   ```
   tests :: [TestCase Bool]
   tests = [assertTrue ((partition odd []) == ([],[]))
           ,assertTrue ((partition odd [1..10]) == ([2,4,6,8,10],[1,3,5,7,9]))
           ,assertTrue ((partition even [1..10]) == ([1,3,5,7,9],[2,4,6,8,10]))
           ,assertTrue ((partition (== 3) [1..5]) == ([1,2,4,5],[3]))
           ,assertTrue ((partition (== 3) [5,7,2]) == ([5,7,2],[]))
           ,assertTrue ((partition (== 3) [3,3,3]) == ([],[3,3,3]))
           ,assertTrue ((partition (== 3) [3,3,4,3]) == ([4],[3,3,3]))
   ```

6. (20 points) [UseModels] This problem is about the type `WindowLayout`, which is defined as follows.

```
data WindowLayout = Window {wname :: String, width :: Int, height :: Int}
                  | Horizontal [WindowLayout]
                  | Vertical [WindowLayout]
```

In Haskell, write a function

```
iconify :: WindowLayout -> WindowLayout
```

that takes a ⟨WindowLayout⟩, `wl`, and returns a ⟨WindowLayout⟩ that is just like `wl`, except that in each ⟨Window⟩ record, the value of each `width` and `height` field is replaced by 2. The following are examples using the `Testing` module from the homework.

```
tests :: [TestCase WindowLayout]
tests =
    [eqTest (iconify Window {wname="castle", width=1280, height=740})
     "==" (Window {wname="castle", width=2, height=2})
    ,eqTest (iconify (Horizontal [Window {wname="castle", width=1280, height=740},
                                  Window {wname="bball", width=900, height=900}]))
     "==" (Horizontal [Window {wname="castle", width=2, height=2},
                       Window {wname="bball", width=2, height=2}])
    ,eqTest (iconify (Vertical [])) "==" (Vertical [])
    ,eqTest (iconify (Horizontal [])) "==" (Horizontal [])
    ,eqTest (iconify (Vertical [Horizontal [Window {wname="castle", width=1280, height=740},
                                            Window {wname="bball", width=900, height=900}],
                                Vertical [Window {wname="csi", width=1000, height=500}]]))
     "==" (Vertical [Horizontal [Window {wname="castle", width=2, height=2},
                                 Window {wname="bball", width=2, height=2}],
                     Vertical [Window {wname="csi", width=2, height=2}]])
    ,eqTest (iconify (Horizontal [Vertical [Window {wname="csi", width=1280, height=740},
                                            Window {wname="daily", width=900, height=900}],
                                  Horizontal [Window {wname="news", width=1000, height=500},
                                              Horizontal [Window {wname="pbs", width=800,height=400}]]]))
     "==" (Horizontal [Vertical [Window {wname="csi", width=2, height=2},
                                            Window {wname="daily", width=2, height=2}],
                                  Horizontal [Window {wname="news", width=2, height=2},
                                              Horizontal [Window {wname="pbs", width=2,height=2}]]]) ]
```

Be sure to follow the grammar!

7. (25 points) [UseModels] Consider the data type of quantified Boolean expressions defined as follows.

```
data QBExp = Varref String | QBExp `Or` QBExp | Exists String QBExp
```

Your task is to write a function

```
freeQBExp :: QBExp -> [String]
```

that takes a `QBExp`, `qbe`, and returns a list containing just the strings that occur as a free variable reference in `qbe`. The following defines what "occurs as a free variable reference" means. A string `s` *occurs as a variable reference* in a `QBExp` if `s` appears in a subexpression of the form `(Varref s)`. Such a string *occurs as a free variable reference* if it occurs as a variable reference in a subexpression that is outside of any expression of the form `(Exists s e)`, which declares `s`. The following are examples that use the course's Testing module. Note that the lists returned by `freeQBExp` should have no duplicates. In the tests, the `setEq` function constructs a test case that considers lists of strings to be equal if they have the same elements (so that the order is not important).

```
tests :: [TestCase [String]]
tests = [setEq (freeQBExp (Varref "x")) "==" ["x"]
        ,setEq (freeQBExp ((Varref "x") `Or` (Varref "y"))) "==" ["x","y"]
        ,setEq (freeQBExp ((Varref "y") `Or` (Varref "x"))) "==" ["y","x"]
        ,setEq (freeQBExp (((Varref "y") `Or` (Varref "x"))
                           `Or` ((Varref "x") `Or` (Varref "y"))))
                "==" ["y","x"]
        ,setEq (freeQBExp (Exists "y" (Varref "y"))) "==" []
        ,setEq (freeQBExp (Exists "y" ((Varref "y") `Or` (Varref "z"))))
                "==" ["z"]
        ,setEq (freeQBExp (Exists "z" (Exists "y" ((Varref "y") `Or` (Varref "z")))))
                "==" []
        ,setEq (freeQBExp ((Varref "z")
                           `Or` (Exists "z" (Exists "y" ((Varref "y") `Or` (Varref "z"))))))
                "==" ["z"]
        ,setEq (freeQBExp (((Varref "z") `Or` (Varref "q"))
                           `Or` (Exists "z" (Exists "y" ((Varref "y") `Or` (Varref "z"))))))
                "==" ["z","q"]  ]
    where setEq = gTest setEqual
          setEqual los1 los2 = (length los1) == (length los2)
                               && subseteq los1 los2
          subseteq los1 los2 = all (\e -> e `elem` los2) los1
```