

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

```
squareOdds :: [Integer] -> [Integer]
```

that takes a list of Integers, `lst`, and returns a list of Integers that is just like `lst`, except that each odd 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 `odd`.

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

```
tests :: [TestCase [Integer]]
tests = [eqTest (squareOdds []) "==" []
        ,eqTest (squareOdds [3]) "==" [9]
        ,eqTest (squareOdds [4,3]) "==" [4,9]
        ,eqTest (squareOdds [1,2,3,4,5,6]) "==" [1,2,9,4,25,6]
        ,eqTest (squareOdds [3,22,3,95,600,0,-1]) "==" [9,22,9,9025,600,0,1]
        ]
```

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 left 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 (y, z, x) is returned; finally, when `amt` is `Two`, then (z, x, y) 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)) == (2,3,1))
  ,assertTrue ((rotate Two (1,2,3)) == (3,1,2))
  ,assertTrue ((rotate Two ("jan","feb","mar")) == ("mar","jan","feb"))
  ,assertTrue ((rotate One ("jan","feb","mar")) == ("feb","mar","jan"))
  ,assertTrue ((rotate Zero (True,False,True)) == (True,False,True)) ]
```

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

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

Write, in Haskell, the function

```
noemails :: [Entry] -> [(String,Integer)]
```

that takes a list of records of type Entry, es, and returns a list of pairs of the name and phone number 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,Integer)]]
tests =
  [eqTest (noemails []) "==" []
  ,eqTest (noemails [Record {name = "Eastman", phone = 3214442211,
                             email = "polaroid@p.com"}])
          "==" [("Eastman",3214442211)]
  ,eqTest (noemails [Record {name = "M", phone = 44153543221, email = "m@mi6.uk"}
                    ,Record {name = "Bond", phone = 44007007007, email = "jb@mi6.uk"}])
          "==" [("M",44153543221),("Bond",44007007007)]
  ,eqTest (noemails sample)
          "==" [("Adams",4075551212),("Bullfinch",5155551212)
              ,("Cassidy",8005551122),("Durham",3059123344),("Eastman",3214442211)]
  ,eqTest (noemails ((Record {name="Durham",phone=3059123344,
                             email="crash@yahoo.com"}):sample))
          "==" [("Durham",3059123344),("Adams",4075551212),("Bullfinch",5155551212)
              ,("Cassidy",8005551122),("Durham",3059123344),("Eastman",3214442211)] ]
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 = "polaroid@p.com"}]
```

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

```
selectiveMap :: (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 satisfies `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 ((selectiveMap odd (\x -> x*x) []) == [])
  ,assertTrue ((selectiveMap odd (\x -> 6000) [2,4,6,8,10] == [2,4,6,8,10])
  ,assertTrue ((selectiveMap odd (\x -> 6000) [3,1,-5] == [6000,6000,6000])
  ,assertTrue ((selectiveMap even (\x -> x*x) [1,2,3,4,5] == [1,4,3,16,5])
  ,assertTrue ((selectiveMap (\c -> c == 'q') (\x -> 'u') "quip") == "uqip")
  ,assertTrue ((selectiveMap (== True) 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 (`yes`, `no`) such that `yes` contains the elements of `lst` that satisfy `p` and `no` contains the elements of `lst` that do not satisfy `p`. In both `yes` and `no` 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]) == ([1,3,5,7,9],[2,4,6,8,10]))
        ,assertTrue ((partition even [1..10]) == ([2,4,6,8,10],[1,3,5,7,9]))
        ,assertTrue ((partition (== 3) [1..5]) == ([3],[1,2,4,5]))
        ,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]) == ([3,3,3],[4]))
```

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 1. 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=1, height=1})
  ,eqTest (iconify (Horizontal [Window {wname="castle", width=1280, height=740},
                                Window {wname="bball", width=900, height=900}]))
    "==" (Horizontal [Window {wname="castle", width=1, height=1},
                      Window {wname="bball", width=1, height=1}])
  ,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=1, height=1},
                                Window {wname="bball", width=1, height=1}],
                    Vertical [Window {wname="csi", width=1, height=1}]]])
  ,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=1, height=1},
                                Window {wname="daily", width=1, height=1}],
                      Horizontal [Window {wname="news", width=1, height=1},
                                  Horizontal [Window {wname="pbs", width=1, height=1}]]]) ]
```

Be sure to follow the grammar!

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

```
data QBEp = Varref String | QBEp `And` QBEp | Forall String QBEp
```

Your task is to write a function

```
freeQBEp :: QBEp -> [String]
```

that takes a QBEp, 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 QBEp 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 (Forall *s* e), which declares *s*. The following are examples that use the course’s Testing module. Note that the lists returned by freeQBEp 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 (freeQBEp (Varref "x")) "==" ["x"]
        ,setEq (freeQBEp ((Varref "x") `And` (Varref "y"))) "==" ["x","y"]
        ,setEq (freeQBEp ((Varref "y") `And` (Varref "x"))) "==" ["y","x"]
        ,setEq (freeQBEp (((Varref "y") `And` (Varref "x"))
                          `And` ((Varref "x") `And` (Varref "y"))))
              "==" ["y","x"]
        ,setEq (freeQBEp (Forall "y" (Varref "y"))) "==" []
        ,setEq (freeQBEp (Forall "y" ((Varref "y") `And` (Varref "z"))))
              "==" ["z"]
        ,setEq (freeQBEp (Forall "z" (Forall "y" ((Varref "y") `And` (Varref "z")))))
              "==" []
        ,setEq (freeQBEp ((Varref "z")
                          `And` (Forall "z" (Forall "y" ((Varref "y") `And` (Varref "z"))))))
              "==" ["z"]
        ,setEq (freeQBEp (((Varref "z") `And` (Varref "q"))
                          `And` (Forall "z" (Forall "y" ((Varref "y") `And` (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
```