Fall, 2004 Name: _____

Com S 541 — Programming Languages 1

Test on Functional Languages, Haskell, and λ-Calculus

Special Directions for this Test

This test has 7 questions and pages numbered 1 through 8.
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 code 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; however, if they are not in the Prelude, then you must write them into your test.

1. (5 points) Name a feature of Haskell that can be regarded as a syntactic sugar, and give an example of that feature and its desugaring into the subset of Haskell that does not use that feature.

2. (10 points) Using the reduction rules for the lambda calculus, show how to derive the normal form (if any) for the following.

$$(\lambda c\,.\,(\lambda a\,.\,\lambda b\,.\,a)\ c)\ b\ ((\lambda f\,.\,\lambda x\,.\,f\ x)\ (\lambda f\,.\,f\ f)(\lambda g\,.\,g\ g))$$

   If this term does not have a normal form, show using the reduction rules why it does not.

3. (15 points) Using the type checking rules for the simply-typed lambda calculus, formally derive and prove the type of the following term, starting from the empty type environment.

$$\lambda c : \mathbf{o} \,.\, (\lambda a : \mathbf{o} \,.\, \lambda b : \mathbf{o} \,.\, a)\ c$$

If this term does not have type in the empty type environment, then carry out the proof to the point where the proof fails, and say why it fails.

4. (5 points) In Haskell, consider the following datatype that describes concatenations of lists directly in a datatype.

```
> data CatList a = Nil | NonEmptyList [a] | Concat (CatList a) (CatList a)
>                  deriving Show
```

The relationship between the types `CatList a` and `[a]`, can be described by two translation functions:

```
> toNormalList :: CatList a -> [a]
> toCatList :: [a] -> CatList a
> toCatList [] = Nil
> toCatList lst@(x:xs) = NonEmptyList lst
```

Your task is to implement the `toNormalList` function, that returns a list consisting of the elements of a `CatList`, in order, such that in `toNormalList (Concat` c d`)`, the elements are ordered with all those in c appearing in the answer before those in d. Furthermore `toNormalList` is the inverse of `toCatList`. The following are examples.

```
toNormalList Nil = []
toNormalList (NonEmptyList [5,4,1,5]) = [5,4,1,5]
toNormalList (Concat (NonEmptyList [5,4,1,5]) (NonEmptyList [6,7]))
          = [5,4,1,5,6,7]
toNormalList (Concat (Concat Nil (Concat Nil (NonEmptyList [3,1])))
                     (Concat (NonEmptyList [5,2]) (NonEmptyList [6,7])))
          = [3,1,5,2,6,7]
```

5. (10 points) Still using the definition of the datatype `CatList` given in the previous problem, implement a function

```
> catlistMap :: (a -> b) -> CatList a -> CatList b
```

such that `catlistMap` f c returns a `CatList` with the same structure as c, but with its elements replaced by the result of applying f to each of them. For example:

```
catlistMap (+10) Nil = Nil
catlistMap (+10) (NonEmptyList [5,4,1,5]) = (NonEmptyList [15,14,11,15])
catlistMap (2*) (Concat (NonEmptyList [5,4,1,5]) (NonEmptyList [6,7]))
          = (Concat (NonEmptyList [10,8,2,10]) (NonEmptyList [12,14]))
catlistMap (\b -> if b then 1 else 0)
           (Concat (Concat Nil (Concat Nil (NonEmptyList [True,False])))
                     (Concat (NonEmptyList [False,False])
                                          (NonEmptyList [True,True])))
         = (Concat (Concat Nil (Concat Nil (NonEmptyList [1,0])))
                     (Concat (NonEmptyList [0,0])
                                          (NonEmptyList [1,1])))
```

6. (15 points) Still using the definition of the datatype `CatList` given in problem 4, write a function,

```
> foldCat :: b -> ([a] -> b) -> (b -> b -> b) -> CatList a -> b
```

such that `foldCat` is a generalization of the previous two problems. For example, if we define

```
> catlistMap2 f = foldCat Nil (NonEmptyList . (map f)) Concat
> toNormalList2 = foldCat [] id (++)
> sumCatList = foldCat 0 sum (+)
```

then we have the following

```
sumCatList Nil = 0
sumCatList (NonEmptyList [5,4,1,5]) = 15
sumCatList (Concat (NonEmptyList [5,4,1,5]) (NonEmptyList [6,7]))
           = 28

toNormalList2 (Concat (NonEmptyList [5,4,1,5]) (NonEmptyList [6,7]))
           = [5,4,1,5,6,7]
toNormalList2 (Concat (Concat Nil (Concat Nil (NonEmptyList [3,1])))
                      (Concat (NonEmptyList [5,2]) (NonEmptyList [6,7])))
           = [3,1,5,2,6,7]

catlistMap2 (2*) (Concat (NonEmptyList [5,4,1,5]) (NonEmptyList [6,7]))
              = (Concat (NonEmptyList [10,8,2,10]) (NonEmptyList [12,14]))
catlistMap2 (\b -> if b then 1 else 0)
            (Concat (Concat Nil (Concat Nil (NonEmptyList [True,False])))
                    (Concat (NonEmptyList [False,False])
                                          (NonEmptyList [True,True])))
          = (Concat (Concat Nil (Concat Nil (NonEmptyList [1,0])))
                    (Concat (NonEmptyList [0,0])
                                          (NonEmptyList [1,1])))
```

7. (40 points) Consider the following simplified abstract syntax of a variant of Abadi and Cardelli's ς-calculus, a normal-order, applicative object calculus.

```
> data Term =
>             Varref Name                              -- variables
>           | Object [FieldBinding] [MethodBinding] -- new objects
>           | Select Access                            -- field selection
>           | Call Access [Term]                       -- method calls
>           | Access :<= Term                          -- update field value
>              deriving Show
>
> type Name = String
> data FieldBinding = Name := Term
>                       deriving Show
> data MethodBinding = Name `Is` Method
>                        deriving Show
> data Method = Sigma Name [Name] Term  -- first name is for the 'this' obj
>                deriving Show
> data Access = Term `Dot` Name
>                deriving Show
```

Your task is to define an instance of the type class `FreeVars` given below, for the datatype `Term`.

```
> class FreeVars a where
>   freeVars :: a -> [Name]
```

The meaning of `freeVars` t, for a `Term` t, is a list (potentially with duplicates) containing just the names used inside a `Varref` subterm of t that do not have a surrounding declaration. The only declaration form is the method constructor, `Sigma`. Thus, for any term b, the free variables of the `Method`,

`Sigma "self" ["x", "y", "z"]` b,

are the free variables of b, minus the declared names `self`, `x`, `y`, and `z`. There are examples on the next page.

The following are examples for the problem on the previous page.

```
freeVars (Varref "f") = ["f"]
freeVars (Select ((Varref "o") `Dot` "f")) = ["o"]
freeVars (Call ((Varref "o") `Dot` "m") [Varref "q", Varref "o"])
         = ["o","q","o"]
freeVars (((Varref "o") `Dot` "m") :<= (Varref "z"))
         = ["o","z"]
freeVars (Object [] [])
         = []
freeVars (Object ["f" := (Varref "x")]
                 ["m" `Is` Sigma "s" ["v", "w"] (Varref "q")])
         = ["x","q"]
freeVars (Object ["f" := (Varref "x"), "g" := (Varref "gval")]
                 ["m" `Is` Sigma "s" ["v", "w"] (Varref "q")])
         = ["x","gval","q"]
freeVars (Object ["f" := (Varref "x"), "g" := Select ((Varref "t") `Dot` "f")]
                 ["m" `Is` Sigma "s" ["v", "w"] (Varref "q")])
         = ["x","t","q"]
freeVars (Object ["f" := (Varref "x")]
                 ["n" `Is` Sigma "self" [] (Varref "y"),
                  "m" `Is` Sigma "s" ["v", "w"] (Varref "z")])
       = ["x","y","z"]
freeVars (Object ["f" := (Varref "x")]
                 ["n" `Is` Sigma "self" [] (Varref "y"),
                  "m" `Is` Sigma "s" ["v", "w"] (Varref "v")])
      = ["x","y"]
 freeVars (Object []
                  ["m" `Is` Sigma "s" ["v", "w"]
                            (Select ((Varref "x") `Dot` "f"))])
        = ["x"]
freeVars (Object []
                 ["m" `Is` Sigma "s" ["v", "w"]
                           (Select ((Varref "v") `Dot` "f"))])
        = []
freeVars (Object ["f" := (Varref "x")]
                 ["m" `Is` Sigma "s" ["v", "w"]
                           (Object ["f" := (Varref "v"),
                                    "g" := (Select ((Varref "s") `Dot` "f")),
                                    "h" := (Varref "w")]
                                   [])])
       = ["x"]
freeVars (Object ["f" := (Varref "x")]
                 ["m" `Is` Sigma "self" []
                           (Object ["f" := Object []
                                                  ["n" `Is`
                                                    Sigma "s" []
                                                     (Select ((Varref
                                                      "qq")
                                                      `Dot` "f"))]]
                                   [])])
      = ["x","qq"]
```