Spring, 1997 Name: ____________________

Com S 541 — Programming Languages 1

Test on Haskell and Functional Programming

Special Directions for this Test

This test has 7 questions and pages numbered 1 through 6.
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.
You can use anything that is built-in to Haskell. **You may use any other function in a book, or from your notes, but to do so you must write it into your answer.**

1. (10 points) In Haskell, write a function

```
> disjoint :: (Eq a) => [(a,b)] -> [(a,b)] -> Bool
```

such that `disjoint r1 r2` is true if no first element of a pair in `r1` is also a first element of a pair in `r2` and vice versa.

For example:

```
disjoint [] [(3,7), (9,2)] = True
disjoint [(3,4), (5,7)] [(7,5), (4,3)] = True
disjoint [(1,'a'), (3,'c'), (2,'b'), (4,'b')] [(5,'e'), (2,'f')] = False
disjoint  [(5,'e'), (2,'f')] [(1,'a'), (3,'c'), (2,'b'), (4,'b')] = False
```

2. (15 points) The function

```
> bind :: Eq a => [a] -> [b] -> (a -> b) -> (a -> b)
```

can be used in an interpreter, for example to binding formals to actuals. For example the following equations should hold for all `g`.

```
(bind ["f1","f2"] [3,4] g)("f1") = 3
(bind ["f1","f2"] [3,4] g)("f2") = 4
(bind ["f1","f2"] [3,4] g)("x") = g("x")
(bind ["f2"] [4] g)("f2") = 4
(bind [] [] g)("f2") = g("f2")
```

You may assume that there are no duplicates in the first argument to `bind`, and that the second argument has the same length as the first.

3. (20 points) Consider the following Haskell definitions.

```
> data LambdaTerm = Var String
>                   | Apply LambdaTerm LambdaTerm | Lambda String LambdaTerm

> freeVars :: LambdaTerm -> [String]
> freeVars (Var s)            = [s]
> freeVars (Apply rator rand) = (freeVars rator) `union` (freeVars rand)
> freeVars (Lambda x term)    = delete x (freeVars term)

> boundVars :: LambdaTerm -> [String]
> boundVars (Var s)            = []
> boundVars (Apply rator rand) = (boundVars rator) `union` (boundVars rand)
> boundVars (Lambda x term)    = [x] `union` (boundVars term)

> rename :: String -> String -> LambdaTerm -> LambdaTerm
> rename new old (Var s)            = if s == old then (Var new) else (Var s)
> rename new old (Apply rator rand) = Apply (rename new old rator)
>                                           (rename new old rand)
> rename new old (Lambda x term)    = (Lambda (if x == old then new else x)
>                                             (rename new old term))
```

Write a Haskell function,

```
> lambda_recur :: (String -> a) -> (a -> a -> a) -> (String -> a -> a)
>                   -> LambdaTerm -> a
```

such that one can define the above functions as follows.

```
freeVars = lambda_recur (\s -> [s]) union delete
boundVars = lambda_recur (\s -> []) (\ls1 -> \ls2 -> ls1 `union` ls2) f
            where f x a = [x] `union` a
rename new old = lambda_recur rename_var Apply rename_abs
                 where rename_var s = if s == old then (Var new) else (Var s)
                       rename_abs s a = Lambda (if x == old then new else x) a
```

Please answer the following briefly, in a few sentences eachr and an example. **Be sure to include an example in each answer.**

4. (15 points) Why is it important that a lazy language be applicative?

5. (15 points) What is the importance of syntactic sugars in language design?

6. (10 points) Describe how a closure is like an object in an object-oriented programming language.

7. Consider Haskell code below. This is an abstract syntax for a very small subset of Prolog.

```
> type Program = [HornClause]
> data HornClause = Assert Term | Term `If` [Term]
> data Term = Sym String | Num Int | App Term Term
```

(a) (20 points) Write a function

```
> symbols :: Program -> [String]
```

that takes a `Program` and returns a list of all the strings that appear in that program. For example:

```
symbols [Assert (App (Sym "good") (Sym "prolog")),
         (Sym "imokay") `If` [(Sym "youreokay")]]
 = ["good", "prolog", "imokay", "youreokay"]

symbols [(App (Sym "num") (Sym "x")) `If` [(App (Sym "n2") (Num 3))]]
 = ["num", "x", "n2"]
```

(b) (25 points) Using the same grammar on the previous page, write a function

```
> sane :: Program -> Bool
```

such that `sane prog` returns `True` just when numbers do not occur in inappropriate positions. That is, it returns `True` just when all of the following conditions are true:

- In each `HornClause` of the form `(Assert` t`)`, t is *not* of the form `(Num i)`.
- In each `HornClause` of the form `(`t_0 `` `If` `` `[`t_1`,...,`t_n`])`, t_0, and each of $t_1, \ldots, t_n$ is *not* of the form `(Num i)`.
- In each `Term` of the form `App` t_1 t_2, t_1 is *not* of the form `(Num i)`.

Otherwise, `sane prog` returns `False`

For example:

```
sane [Assert (App (Sym "good") (Sym "prolog")),
         (Sym "imokay") `If` [(Sym "youreokay")]]
 = True

sane [(App (Sym "num") (Sym "x")) `If` [(App (Sym "n2") (Num 3))]]
 = True

sane [(App (Num 3) (Sym "x")) `If` [(App (Sym "n2") (Num 3))]]
 = False

sane [(App (Sym "num") (Sym "x")) `If` [(App (Sym "n2") (Num 3))],
      (Assert (Num 2))]
 = False

sane [] = True
```