Fall, 1998 Name: ____________________

Com S 541 — Programming Languages 1

Test on Functional and Logic Programming Languages

Special Directions for this Test

This test has 7 questions and pages numbered 1 through 9.
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 functions 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; if they are not in the Prelude, write them into your test.
When you write λProlog specifications on this test, you may use any constructors and relations built-in to λProlog; this includes `not` and cut (`!`). You are encouraged to define relations not specifically asked for if they are useful to your programming; if they are not built-in, please write them into your test.

1. (5 points) In logic programming, why are most general unifiers important?

2. (10 points) How does laziness contribute to the utility of programming in Haskell? Give examples.

3. (15 points) Both logic and functional programming are claimed to help reduce the cost of programming. Compare the ease of programming in each paradigm, using Haskell and λProlog as examples of such languages.

4. (15 points) What are some good language design ideas from Haskell that could be used in Java or Smalltalk? Explain why each would be helpful.

5. (10 points) Specify, in λProlog, a relation

```
type append_all  (list (list T)) -> (list T) -> o.
```

that takes a list of lists and appends them all, in order. For example

```
?- append_all ((3::nil)::(2::4::nil)::nil::nil) L.

L = 3 :: 2 :: 4 :: nil.
;
no more solutions
?- append_all LS (1::2::3::2::nil).

LS = (1 :: 2 :: 3 :: 2 :: nil) :: nil.
;

LS = (1 :: 2 :: 3 :: 2 :: nil) :: nil :: nil.
;

LS = (1 :: 2 :: 3 :: nil) :: (2 :: nil) :: nil.
;

LS = (1 :: 2 :: nil) :: (3 :: 2 :: nil) :: nil.
;

LS = (1 :: nil) :: (2 :: 3 :: 2 :: nil) :: nil.
;

LS = nil :: (1 :: 2 :: 3 :: 2 :: nil) :: nil.

yes
?- append_all ((1::nil)::(2::nil)::(3::nil)::(2::nil)::nil) L.

L = 1 :: 2 :: 3 :: 2 :: nil.

yes
```

6. (15 points) Using Haskell, define a function

```
> partitions :: [a] -> [[[a]]]
```

that takes a nonempty list, `xs`, and returns a list of all partitions of `xs` that do not involve the empty list. A partition of a list, `xs`, of elements of some type `a` is a list of lists of elements of type `a`, `xss`, such that appending all the elements of `xss` gives `xs` (as in the previous problem). For example,

```
partitions [1] = [[[1]]]
partitions [2,1] = [[[2], [1]], [[2, 1]]]
partitions [3,2,1] = [[[3], [2], [1]], [[3], [2, 1]],
                      [[3, 2], [1]], [[3, 2, 1]]]
partitions [4,3,2,1] =
   [[[4], [3], [2], [1]], [[4], [3], [2, 1]], [[4], [3, 2], [1]],
   [[4], [3, 2, 1]], [[4, 3], [2], [1]], [[4, 3], [2, 1]],
   [[4, 3, 2], [1]], [[4, 3, 2, 1]]]
partitions "lambda" =
  [["l", "a", "m", "b", "d", "a"], ["l", "a", "m", "b", "da"],
   ["l", "a", "m", "bd", "a"], ["l", "a", "m", "bda"],
   ["l", "a", "mb", "d", "a"], ["l", "a", "mb", "da"],
   ["l", "a", "mbd", "a"], ["l", "a", "mbda"], ["l", "am", "b", "d", "a"],
   ["l", "am", "b", "da"], ["l", "am", "bd", "a"], ["l", "am", "bda"],
   ["l", "amb", "d", "a"], ["l", "amb", "da"], ["l", "ambd", "a"],
   ["l", "ambda"], ["la", "m", "b", "d", "a"], ["la", "m", "b", "da"],
   ["la", "m", "bd", "a"], ["la", "m", "bda"], ["la", "mb", "d", "a"],
   ["la", "mb", "da"], ["la", "mbd", "a"], ["la", "mbda"],
   ["lam", "b", "d", "a"], ["lam", "b", "da"], ["lam", "bd", "a"],
   ["lam", "bda"], ["lamb", "d", "a"], ["lamb", "da"], ["lambd", "a"],
   ["lambda"]]
```

7. (40 points) John Backhus's Turing award lecture described a functional language called "FP." In this problem you will write a semantic interpreter for part of FP.

   Consider the following data definitions.

```
> data Object = IntAtom Integer | BoolAtom Bool | Sequence [Object]
>               deriving Eq
> data Exp = Nth Int | Tail | IsNull
>          | IsAtom | IsEqual | Plus | Sub | And | Not
>          | Compose Exp Exp | Construct [Exp]
>          | Condition Exp Exp Exp | Constant Object
```

   Write a function

```
> meaning :: Exp -> Object -> Object
```

   that takes an expression (in `Exp`) and evaluates it, returning a Haskell function from `Object` to `Object`. This can be regarded as a semantics (or compiler) for FP. The following is an explanation of the expressions, using examples.

   The expressions `Tail`, `IsNull`, `IsAtom`, `IsEqual`, `Plus`, `Sub`, `And`, and `Not` denote primitive functions that take the tail of an object that is a sequence, check if an object is a null sequence, check to see if an object is an atom, check if a sequence of two objects consists of two objects with the same value, adds a sequence of two integer atoms, etc. For example,

```
meaning Tail (Sequence [IntAtom 3, IntAtom 4, IntAtom 5])
          = Sequence [IntAtom 4, IntAtom 5]
meaning IsNull (Sequence [IntAtom 3, IntAtom 4, IntAtom 5]) = BoolAtom False
meaning IsNull (Sequence []) = BoolAtom True
meaning IsNull (IntAtom 3) = BoolAtom False
meaning IsAtom (IntAtom 4) = BoolAtom True
meaning IsAtom (BoolAtom False) = BoolAtom True
meaning IsAtom (Sequence []) = BoolAtom False
meaning IsEqual (Sequence [IntAtom 3, IntAtom 4]) = BoolAtom False
meaning IsEqual (Sequence [BoolAtom False, BoolAtom False]) = BoolAtom True
meaning IsEqual (Sequence [Sequence [], Sequence []]) = BoolAtom True
meaning Plus (Sequence [IntAtom 4, IntAtom 3]) = IntAtom 7
meaning Sub (Sequence [IntAtom 4, IntAtom 3]) = IntAtom 1
meaning And (Sequence [BoolAtom True, BoolAtom True]) = BoolAtom True
meaning And (Sequence [BoolAtom True, BoolAtom False]) = BoolAtom False
meaning Not (BoolAtom True) = BoolAtom False
```

   To make things easier, you don't have to worry about the error cases.

   The meaning of the expression `Nth n` is a function that returns the `n`th element of its argument sequence, for example.

```
meaning (Nth 2) (Sequence [IntAtom 1, IntAtom 2, IntAtom 3]) = IntAtom 2
meaning (Nth 3) (Sequence [IntAtom 1, IntAtom 2, IntAtom 3]) = IntAtom 3
```

   The meaning of the expression `Constant x` is a function that returns `x`. For example,

```
meaning (Constant (IntAtom 3)) (BoolAtom True) = IntAtom 3
meaning (Constant (IntAtom 3)) (Sequence []) = IntAtom 3
meaning (Constant (BoolAtom True)) (Sequence []) = BoolAtom True
```

The meaning of the expression `Construct es`, where `es` is a list of expressions is a function that, when applied, gives a sequence of the result of applying each element of `es` to the argument. (This is, in some sense, the opposite of map.) For example,

```
meaning (Construct []) (IntAtom 4) = Sequence []
meaning (Construct [Constant (IntAtom 3), Constant (IntAtom 4)])
        (BoolAtom True)
     = Sequence [IntAtom 3, IntAtom 4]
meaning (Construct [Sub, Plus, Plus]) (Sequence [IntAtom 7, IntAtom 5])
     = Sequence [IntAtom 2, IntAtom 12, IntAtom 12]
```

The expression `Compose e1 e2` gives the composition of the functions that are the meaning of `e1` and `e2`.

The expression `Condition e1 e2 e3` builds a function that, tests its argument using the meaning of `e1`, and if that application returns `BoolAtom True`, then it applys the meaning of `e2` to the argument, otherwise it applies the meaning of `e3` to the argument. For example:

```
meaning (Condition (Compose Not IsEqual) Sub Plus)
        (Sequence [IntAtom 4, IntAtom 3])
   = IntAtom 1

meaning (Condition (Compose Not IsEqual) Sub Plus)
        (Sequence [IntAtom 4, IntAtom 4])
   = IntAtom 8
```

Hint: be particularly careful to make sure the meaning of `Construct` type checks.

There is more space on the next page for your answer.

-- Additional space for your answer.