Spring, 1992 Name: ______________________________

CS 541 — Programming Languages 1

Functional Programming Test

(Problem Solving, Version 1)

Before you begin, please take a moment to look over the entire test so that you can budget your time.
If you get an incomplete on this test, you can get at most a 'B' on the makeup test.
For programming problems, there is usually more than one right answer, hence it is important to present with your program evidence that it is correct. Answers to programming questions will be scored on the following points:

- elegance and clarity (conciseness, using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution, and how clear and convincing your argument is that the code is correct)
- correctness (whether your program solves the given problem correctly in the given programming language)

You should try to eliminate all syntax errors from your programs, as they will affect both the correctness and clarity of your solution; you may receive no points for programs with major syntax errors. Run-time space and time efficiency of programs is not important, unless the problem says so. Also, checking of external inputs, prompting, labeling output, and other conventions for dealing with humans are not important (for this class), unless the problem says so. The breakdown of points between these categories is roughly as follows.

- 60% for elegance and clarity (using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution),
- 40% for correctness (whether your program solves the given problem correctly in the given subset of the given language)

You are encouraged to use auxiliary subroutines whenever they are helpful.
For this test, you may all the predefined identifiers in Standard ML without further mention. If you wish, you may also use anything that is coded in *ML for the Working Programmer*, provided you tell use the page number where to find the code in the form: "Using facti from page 56." (If you find the solution to one of these problems, copy it out, but don't ask the us if that's a mistake on our part. It certainly is, but we won't want to hear about it during the test!)
If you wish to use functions from your solutions in other parts of the test, you must clearly state where (which problem) we can find those function definitions.

1. Consider the following definition of binary trees (this is different than the one in the book and the one discussed in class).

```
datatype 'a Btree = EmptyTree
                  | Node of 'a
                  | Tree of ('a Btree) * 'a * ('a Btree);
```

For example,

```
val myBtree = Tree(Node("left"), "root", Node("right"));
```

is a `string Btree` whose root is the string `"root"`, whose left subtree is the node containing the string `"left"`, and whose right subtree is the node containing the string `"right"`.

(a) (5 points) Write a function

```
Btree_preorder: 'a Btree -> 'a list
```

that returns a list of the items in the Btree found in the preorder traversal. For example, using the `myBtree` defined above, `Btree_preorder myBtree` = `["root", "left", "right"]`.

(b) (5 points) Write a function

```
Btree_count: 'a Btree -> int
```

that returns the number of items in a tree. For example, `Btree_count myBtree` = 3.

2. (5 points) In Backus's language FP, one of the built-in functions is called `distl` (for distribute left). Write the function

```
distl: ('a* ('a list)) -> ('a list list)
```

so that `distl(`x`,`l`)` makes a list of 2-element lists, of the same length as l, such that the nth 2-element list consists of x followed by the nth element of l. For example, `distl(5,[1,2,3])` = `[[5,1], [5,2], [5,3]]`, and `distl(6,[])` = `[]`. Note that a list of 2-element lists is different from a list of pairs.

3. (10 points) Another primitive in Backus's language FP is `constr`, which turns a list of functions into a function. You are to write a curried version of `constr`

```
constr: (('a*'b -> 'c) list) -> ('a*'b) -> 'c list
```

so that `((constr` f`::`l`) (`x`,`y`))` = f`(`x`,`y`) :: ((constr` l`) (`x`,`y`)` and `((constr []) (`x`,`y`))` = `[]`. For example, `(constr [(op +), (op *)]) (3,4)` = `[7, 12]`.