Spring, 1997 Name: ______________________

Com S 541 — Programming Languages 1

Test on Logic Programming and Operational Semantics

(with a bit of functional programming)

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.
For programming questions, you can use anything that is built-in to the appropriate language (Haskell or λProlog, including the Haskell prelude). **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. Consider the following Haskell functions, from the standard prelude.

```
(.)               :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x          = f (g x)
map               :: (a -> b) -> ([a] -> [b])
map f []           = []
map f (x:xs)       = f x : map f xs
foldr             :: (a -> b -> b) -> b -> [a] -> b
foldr f z []       = z
foldr f z (x:xs) = f x (foldr f z xs)
and               :: [Bool] -> Bool
and                = foldr (&&) True
all               :: (a -> Bool) -> [a] -> Bool
all p              = and . map p
```

Using the above functions, complete the coding of the following problems, by filling in the blanks in the partial solutions given below.

(a) (5 points) The function `lookupff` takes a finite function and a key (an element of the finite function's domain), and returns the corresponding value from the function's range. Finite functions are represented as lists of key-value pairs, with no duplicate keys.

```
> type FiniteFun key value = [(key,value)]
> lookupff :: (Eq key) => FiniteFun key value -> key -> value
```

For example:

```
lookupff [(3,30), (1,10), (2,20)] 2 = 20
lookupff [(3,30), (1,10), (2,20)] 1 = 10
lookupff [(7,77), (9,99)] 7 = 77
```

Write the code for this by filling in the blank in the following; you may also add a `where` clause if you wish.

```
lookupff ff key =
      foldr ______________________________________________________
            (error "no corresponding value") -- second argument
            ff                               -- third argument
```

(b) (5 points) The function `lookupEach` takes a finite function and a list of keys and returns a list of the corresponding values.

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

For example:

```
lookupEach [(3,30), (1,10), (2,20)] [] = []
lookupEach [(3,30), (1,10), (2,20)] [1,2,3,1] = [10,20,30,10]
```

Write the code for this by filling in the blanks in the following; you may use the function `lookupff` from the previous problem and you may also add a `where` clause if you wish.

```
lookupEach ff keys =
      map ____________________________________________
          ____________________________________________
```

(c) (5 points) The function `nonDirected` returns True just when its argument is a list of pairs in which the second element of each pair is the empty list. It has the type

```
> nonDirected :: [(a,[b])] -> Bool
```

For example:

```
nonDirected [] = True
nonDirected [(3,[]), (5,[])] = True
nonDirected [(3,[]), (5,[]), (1,[22])] = False
```

Write the code for this by filling in the blank in the following; you may also add a `where` clause if you wish.

```
nonDirected args =
      all ________________________________________________
          args
```

2. (25 points) Consider the following two λProlog modules.

```
module maps.
type mappred  (A -> B -> o)  -> (list A) -> (list B) -> o.
mappred P nil nil.
mappred P (X::L) (Y::K) :- P X Y, mappred P L K.
type forevery (A -> o) -> (list A) -> o.
forevery P nil.
forevery P (X::L) :- P X, forevery P L.

module csp.
import maps.
kind process            type.
type seq                (list process) -> process.
type par                (list process) -> process.
type skip               process.
type is_skip            process -> o.
type reducesto          process -> process -> o.
type p                  process.
type q                  process.
type r                  process.
is_skip skip.
reducesto (seq nil) skip.
reducesto (seq (P::PS)) (seq (P1::PS)) :- reducesto P P1.
reducesto (seq (skip::PS)) (seq PS).
reducesto (par LP1) (par LP2) :- mappred reducesto LP1 LP2.
reducesto (par LP) skip :- forevery is_skip LP.
reducesto p skip.
reducesto q (par (p::q::nil)).
reducesto r (seq (p::q::r::nil)).
```

Fill in the blanks in the output below to show what λProlog would print for the following query. (Hint, the query does succeed.)

```
?- use "maps".
?- use "csp".
?- reducesto r A, reducesto A B, reducesto B C,
   reducesto C D, reducesto D E.

A = ____________________________________________________________,
B = ____________________________________________________________,
C = ____________________________________________________________,
D = ____________________________________________________________,
E = ____________________________________________________________.
```

3. (10 points) Write, in λProlog, a relation `merge`

```
type merge (list int) -> (list int) -> (list int) -> o.
```

that relates two lists that are sorted in nondecreasing order, to a list with all their elements, also sorted in nondecreasing order.

For example.

```
?- merge nil nil Ans.
Ans = nil.
?- merge nil (1::2::3::nil) Ans.
Ans = 1 :: 2 :: 3 :: nil.
?- merge (1::3::5::nil) (2::4::5::6::nil) Ans.
Ans = 1 :: 2 :: 3 :: 4 :: 5 :: 5 :: 6 :: nil.
```

Hint: you can use terms of the form t_1 `<` t_2, t_1 `>=` t_2, and so on to compare integers. The infix operators `<`, `<=` and so on have the type `int -> int -> o`.

4. (15 points) This is a problem about the λ-calculus. Give the normal form for each of the following terms; however, if the term has no normal form, write "no normal form". To receive partial credit, show the steps of your work.

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

   (b) $((\lambda x\,.\,((\lambda k\,.\,(\lambda z\,.\,((k\ z)\ z)))\ (\lambda x\,.\,(\lambda y\,.\,x))))\ ((\lambda x\,.\,(x\ x))\ (\lambda x\,.\,(x\ x))))$

5. (30 points) Complete the following λProlog module so that it specifies binary trees and relations that given a binary tree, determine its:

   - its size (returning an integer),
   - its depth (returning an integer),
   - and do a preorder traversal (returning a list of elements).

   A binary tree holds *elements*, which are values of some arbitrary type. For constructors, think about ways to make a bigger tree out of two smaller trees and an element. You should make whatever decisions you feel appropriate about the trees; there is no one right answer for this problem.

```
module binary_trees.
import lists.    % includes append, etc.
kind btree  type -> type.
type preorder (btree T) -> (list T) -> o.
%% fill in your answer below...
```

6. (30 points) Write an big-step (evaluation) operational semantics for boolean expressions. Use the following grammar as the concrete syntax.

$$
\begin{array}{rl}
\langle \mathtt{B} \rangle ::= & \mathtt{err} \mid \mathtt{truth} \mid \mathtt{falsity} \\
\mid & \mathtt{not}\ \langle \mathtt{B} \rangle \mid \mathtt{(}\ \langle \mathtt{B} \rangle\ \mathtt{)} \\
\mid & \langle \mathtt{B} \rangle\ \mathtt{and}\ \langle \mathtt{B} \rangle \mid \langle \mathtt{B} \rangle\ \mathtt{or}\ \langle \mathtt{B} \rangle
\end{array}
$$

Your semantics should evaluate boolean expressions left-to-right. It should also use "short circuit" evaluation; that is, no term should be evaluated if its value is not needed to determine the value of the overall expression. Note that the constant `err` generates an error. You can use either λProlog or the standard mathematical notations.

7. (30 points) Describe the similarities and differences between operational semantics and programs in λProlog.