Exam I
Sample Solutions

Problem 1.

(a) X is the correct answer -- the number of choices of 0 **{**110**}** 1 corresponds to the number of times around the outer-loop of diagram A; for each 0 **{**110**}** 1 choice, the number of choices of 110 corresponds to the number of times around the inner-loop on that pass of the outer loop. For instance, W is wrong because it does not include 0101 while A does.

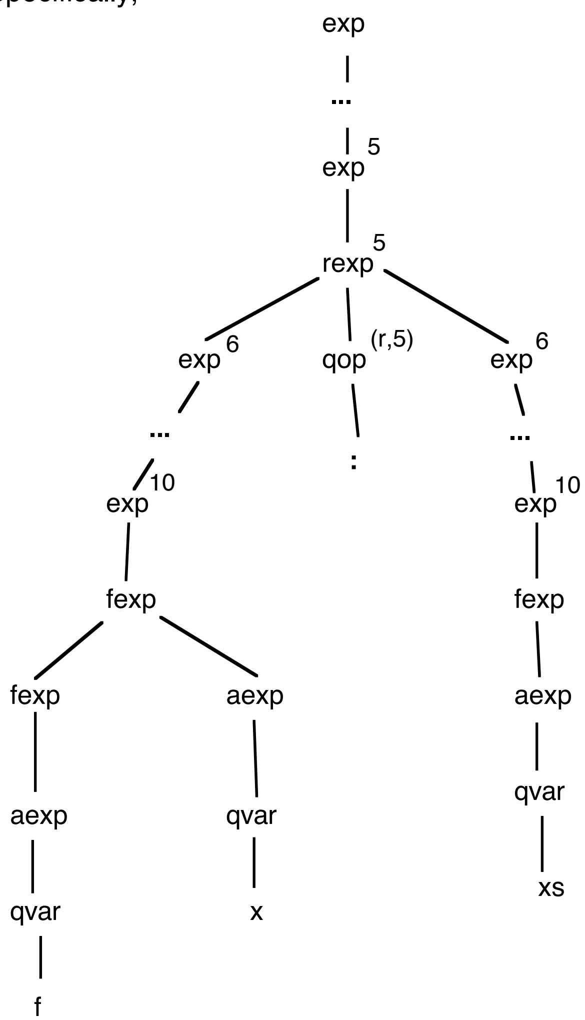
(b) Z is the correct answer -- □ agrees with B by the "straight through" path, 0**{**1**}**0Z agrees for the 1-subloop possibly followed by another outer-loop iteration, and 0Z0Z agrees for the B-recursion for the first Z and another possible outer-loop iteration for the second. For instance, Y is wrong because it allows 0100 while B does not.

Problem 2.

Function application has higher precedence than list construction (:) so both the typing and derivation structure should indicate that f is applied only to x, and the result of the expression is a list with first element f x and tail xs. Note that the "types" referred to are either concrete (Int, Bool, etc.) or polymorphic (a, b, etc.) types.

(a) f x is function application and since there are no pre-defined types involved, f is polymorphic with f :: a -> b, x :: a, and f x :: b. From its use with ':', xs is a list whose element type is undetermined, so xs :: [c]. For f x : xs to be type correct, it must be that c = b. Hence the types are f :: a -> b, x :: a, and xs :: [b].

(b) The lower precedence operation ':' will appear earlier (i.e., above) function application in the derivation tree. To introduce the ':' operation, we must derive $qop^{(r,5)}$, and this in turn requires deriving $rexp^5$, so this guides the first steps. Specifically,



Problem 3

This problem requires nested repetitions -- at the outer level, to cycle through each list item, and then for each one to search for a repetition. For this problem, we present two solutions. First, a recursive approach. Recursion is used for the outer repetition, and the nested cycle is accomplished iteratively using pre-defined functions filter and elem. Note that neither the amount nor depth of recursion exceed the length of the argument list.

```
> extractRepeats [] = []
> extractRepeats (x:xs)
>   | elem x xs     = x:extractRepeats rest   -- add x, repeat with later x removed
>   | otherwise     = extractRepeats xs       -- x not duplicated, drop x and repeat
>     where rest = filter (/=x) xs
```

A solution without recursion can also be constructed. The outer repetition is accomplished using list comprehension with a generator for the list indices, followed by two filters for the nested repetitions using elem, take, and drop pre-defined functions; the first filter checks that an item xs!!k is duplicated later in the list, and the second makes sure items repeated multiple times appear in the result only once.

```
> extractRepeats2 xs
>      = [xs!!k | k<-[0..length xs -1],            -- generate each index
>                 elem (xs!!k) (drop (k+1) xs),     -- test item reappearing later in xs
>                 not (elem (xs!!k) (take k xs))]   -- but not earlier
```