Sample Exam 3 Questions

Problem 1. We can make binary trees with strings for leaves, or binary trees with numbers for leaves, or binary trees with anything we'd like for the leaves, using the following data definition.

```
(define-struct node (left right))
;; A [BT X] is one of:
;; - an X
;; - (make-node [BT X] [BT X])
```

For example, we can make a binary tree of strings (*i.e.*, a `[BT String]`) with

```
(define bt1 (make-node (make-node "Olin" "Shivers")
(make-node "David"
(make-node "Van" "Horn"))))
```

Recall that the `foldr` operation allows us to process the elements of a list: add them up, multiply them together, assemble them into a set, *etc.* Similarly, we can define an analogous "loop function" to fold up a binary tree, called `fold-tree`.
Applying `fold-tree` with these arguments
`(fold-tree + string-length bt1)`
will replace every occurrence of make-node in bt1 with +, and every leaf s with `(string-length s)`, computing

```
(+ (+ (string-length "Olin") (string-length "Shivers"))
(+ (string-length "David")
(+ (string-length "Van") (string-length "Horn"))))
```

In other words, the first argument to fold-tree says what to do to the node structures of the tree, while the second argument says what to do to its leaves; in the example above, we produced the total length of all the strings in the tree.

a. Design `fold-tree`.
(Hint: You might want to check your signature against the `bt1` example above.)

b. Use `fold-tree` to define the `height` function, which produces the height of a tree.
(Assume the height of a leaf is zero.)

Problem 2. You have just been hired as a programmer at Budget Tech, a startup that produces accounting software for small businesses. In many of the computations, the following data definition is used to represent the monthly budget summary:

```
;; a Monthly-Report is a
;;     (make-monthly-report string number number number)
;; where month is the current month
;; year is the current year
;; expenditures is the total expenditures for the month
;; revenues is the total revenues

(define-struct monthly-report
           (month year expenditures revenues))
```

All of these monthly reports are stored sequentially in a database (i.e., a list):

```
;; A [Listof Monthly-Report] is one of
;; --empty
;; --(cons Monthly-Report [Listof Monthly-Report])
```

Design a function `total-profits` that computes the total profits for a given year,
where Profit = Revenue – Expenditures
Your function must use an accumulator.

```
;; total-profits: Number [Listof Monthly-Report] -> Number
;; consumes a list of monthly budget reports and a year and
;; computes the total profits for that year
```

Problem 3. A variable definition associates a number with the name `target`. The number is unknown to you, but you are given the left endpoint and the right endpoint of the search space that contains `target`.

Design a function, `guess-number`, that consumes two integers `lo` and `hi`, and produces the number stored in target. Assume that *`lo < hi`*, and that `target` is an integer in the range *`[lo, hi]`*. Your function must be efficient to get credit; it should only make at most about 20 guesses in order to find a number in the range [0, 1000000].

Problem 4. We order strings using *lexicographic order*, where, for example,

"cap" < "capillary" < "cats"

This is how we order words in a dictionary.

We can apply the idea of lexicographic order to lists of numbers. A list of numbers *a* is *lexicographically less than* another list *b* if either:
(1) *a* is a proper prefix of *b*, or
(2) some element of *a* is less than the corresponding element of *b*, and the two lists match identically on the preceding elements. So, for example,

`'(3 -8)` is lexicographically less than `'(3 -8 9 -100 5 7)`
`'(3 2)` is lexicographically less than `'(3 5)`
`'(3 -8 9 -100 5)` is lexicographically less than `'(3 -8 10 -500))`

Design `lex<`, the comparison function that takes two lists of numbers as inputs, and determines if its first argument is lexicographically less than its second argument.

Problem 5. You can define binary trees that contain values at every node of the tree, not just at the leaves. For example, this data definition lets us represent binary trees that have a numeric grade in the range [0, 100] at every node in the tree:

```
(define-struct node (left val right))
;;; A GradeTree is one of:
;;; - a Grade
;;; - (make-node GradeTree Grade GradeTree)
;;;
;;; A Grade is a number in the range [0,100].
```

Here are three example `GradeTrees`:

```
(define tree1 (make-node (make-node 3 0 1)
                         10
                         (make-node 8 100 (make-node 1 2 3))))
(define tree2 (make-node (make-node 1 0 10)
                         3
                         105))
(define tree3 (make-node (make-node 1 5 8)
                         10
                         (make-node (make-node 11 17 18)
                                    23
                                    87)))
```

A `GradeTree` node is *ordered* if
_ all the numbers in its left child are smaller than the node's value,
_ all the numbers in its right child are greater than the node's value, and
_ the left and right subtrees are both ordered.
(Note that this is a recursive definition.) A tree *leaf* (being just a number) is considered to be trivially ordered.

So, for example, `tree1` is *not* ordered, because the left subtree `(make-node 3 0 1)` is not ordered. Likewise, `tree2` is also not ordered, because the left child contains the numbers {0, 1, 10}, but 10 isn't less than root node's value, 3.

On the other hand, `tree3` *is* ordered.

Ordered trees are useful. We can, for instance, check a large ordered tree to see if it contains a given number much more quickly than we can check an unordered tree, or the equivalent list of numbers. Please design a function `tree-ordered?` that takes a `GradeTree` and determines if the tree is ordered or not. Note: for credit, your program must be simple, elegant and efficient: the amount of work it does should be proportional to the total number of numbers in the tree. Hint: you may find it useful to declare a helper function. . .