The exam is a **one-hour** exam. To accommodate everyone’s needs for time and space, the instructors will stay for three hours.

Write down the answers in the space provided. You may use the back of each piece of paper, too, but please keep your work *legible* and *organized*.

You may use all the definitions, expressions, and functions found in ISL, especially those suggested in hints. Define everything else.

The phrase “design a function” means that you should apply the design recipe. *Show all steps*, though you may skip the template unless the problem explicitly calls for it. You may use a shorthand notation to write any examples or test cases: for example, \((+\ 2\ 2) \rightarrow 4\) to indicate \(\text{check-expect} \ ((+\ 2\ 2))\ 4\).

Some basic test taking advice: Before you start answering any problems, read *every* problem, so your brain can be thinking about the harder problems in the background while you knock off the easy ones.
Problem 1  Design the function take-while, that takes a list and a predicate, and returns all the elements from the front of the list for which the predicate returns true: that is, all the elements up until the first element that does not pass the predicate. Do not use any list abstractions to implement this function, including length. And, be sure to give the best signature you can for this function.
Problem 2 Design a function \texttt{counts-of-multiples}, that takes a list of Naturals and a Natural, and produces a Count of how many numbers in the given list are a multiple of the given number and how many numbers in the list are not. For instance, if the given number is 2, then \texttt{counts-of-multiples} returns the number of even numbers and number of odd numbers in the list. Reminder: \texttt{(modulo m n)} returns the remainder when \texttt{m} is divided by \texttt{n}.

; A Count is a (make-count Natural Natural)
; \textbf{INTERPRETATION:} the number of exact multiples of a number
; and the number of non-multiples in some collection of numbers
(define-struct count [multiples leftovers])

Use \texttt{local} and list abstractions (figures 95 and 96 from the book, which are reproduced at the end of the exam) to design this function.
(space for problem 2)
**Problem 3** Consider the following data definition:

```plaintext
; A Shrub is one of
; - Number
; - [List-of Shrub]
; INTERPRETATION: Describes a branching garden plant, either
; the size of a leaf (in inches), or a fork with an arbitrary
; number of branches coming off it
```

Design the function `max-branches`, that computes the largest number of branches coming out of a fork in a Shrub.

You may (but do not have to) use list abstractions for this problem.
Problem 4 Consider the following definition:

; A StringExpr is one of
; - String
; - (list StringExpr '+' StringExpr)

Design the function combine, that takes a StringExpr and concatenates all the Strings inside it.
(space for problem 4)
; [X] Natural [Natural -> X] -> [List-of X]
; constructs a list by applying f to 0, 1, ..., (sub1 n)
; (build-list n f) == (list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

; [X] [X -> Boolean] [List-of X] -> [List-of X]
; produces a list from those items on lx for which p holds
(define (filter p lx) ...)

; [X] [List-of X] [X X -> Boolean] -> [List-of X]
; produces a version of lx that is sorted according to cmp
(define (sort lx cmp) ...)

; [X Y] [X -> Y] [List-of X] -> [List-of Y]
; constructs a list by applying f to each item on lx
; (map f (list x-1 ... x-n)) == (list (f x-1) ... (f x-n))
(define (map f lx) ...)

; [X] [X -> Boolean] [List-of X] -> Boolean
; determines whether p holds for every item on lx
; (andmap p (list x-1 ... x-n)) == (and (p x-1) ... (p x-n))
(define (andmap p lx) ...)

; [X] [X -> Boolean] [List-of X] -> Boolean
; determines whether p holds for at least one item on lx
; (ormap p (list x-1 ... x-n)) == (or (p x-1) ... (p x-n))
(define (ormap p lx) ...)

; [X Y] [X Y -> Y] Y [List-of X] -> Y
; applies f from right to left to each item in lx and b
; (foldr f b (list x-1 ... x-n)) == (f x-1 ... (f x-n b))
(define (foldr f b lx) ...)

; [X Y] [X Y -> Y] Y [List-of X] -> Y
; applies f from left to right to each item in lx and b
; (foldl f b (list x-1 ... x-n)) == (f x-n ... (f x-1 b))
(define (foldl f b lx) ...)

Figure 1: ISL's abstract functions for list-processing