About Exam 2

The exam is closed-book, closed-notes except for one (8.5×11-inch) sheet of notes.

The exam is technically cumulative, but it will emphasize the earliest material covered after Exam 1. Thus, the topics and questions are likely to include:

1 Lists

- You can assume you're using Intermediate Student Language (isl) on this exam, so it's at least as important to know how to make a list with the list shorthand as it is to make one an item at a time using cons.
- You should know how to use the basic selectors and predicates for lists. For each one, I expect you to know what type of data the function expects as input and generates as output:
  - first
  - rest
  - empty?
  - cons?
- You may be asked what the result is after calling one of the above functions.
- You need to be able to design or write a function that does something to a list without using list abstractions (i.e., without higher-order functions like map)
- You should be able to work with a list-like recursive data definition like the Russian Dolls example in § 9.4. These can look different, but they’re just thinly veiled lists; the idea is the same!
- You may be asked to work with a list of structures (e.g., a list of Posns).

2 Functional abstraction: Abstracting over two functions. Stages:

- Design a more general, parameterized version of the two given similar functions
- Use the new parameterized function to rewrite the two original functions
- Generalize the new function even more by adding a predicate function as a parameter, making it a higher-order function
- Write a new function, using the higher-order function within a local expression that defines a predicate function
3 Using higher-order functions: Simplify your design of a function. Stages:

- Define a top-level function
- Define a new function that passes the previous top-level function to map, foldr, or filter
- Revise the last function to use a local expression
- Replace the local expression with an anonymous (lambda) function

4 List abstractions

- You'll need to understand and be able to use the following built-in abstractions over functions that consume lists:
  - map
  - filter
  - foldr
  - andmap
  - ormap
- Given a parameterized function signature, e.g.,

  ```
  ;; foo : [List-of X], [X -> Y] -> [List-of Y]
  ```

  a question might ask you to demonstrate calling foo and/or to identify the data types of the parameters.

5 Scope and local

- You should know that, by default, functions and constants are global but parameters are local.
- You should know the basic structure of local in ISL:

  ```
  (local [defn
          defn
          ...
          ]
    expression)
  ```

- You might be asked to rewrite a global helper function as a local function.
- The syntax for local can be tricky, so this would probably be a good thing to have an example of on your sheet of notes. You'll definitely be asked to write a function using local.
Problem 1

According to Wikipedia, Russian (Matryoshka) dolls are “dolls of decreasing size placed one inside another”. Section 9.4 of How to Design Programs gives the following data definition for a Russian doll:

```
(define-struct layer [color doll])
```

;; An RD (short for Russian doll) is one of:
;; - String
;; - (make-layer String RD)

For the following problems, follow the design recipe, but you do not need to show a template (which is provided in the textbook).

a. Design the function inner that consumes an RD and produces the (color of the) innermost doll.

b. Design the function contains-color?, which consumes an RD and a color (e.g., “red”) and determines whether the given RD contains a doll of the given color.
Problem 2

Here are two examples of a list of numbers, expressed using list abbreviations:

```
(define ODDS (list 1 3 5 7 9))
(define EVENS (list 2 4 6 8 10))
```

a. Rewrite the definitions for ODDS and EVENS using cons expressions – no list abbreviations.

b. Evaluate the following expressions. For expressions that evaluate to a list, you’re free to use list abbreviations or cons expressions.

```
(first EVENS)

(rest ODDS)

(first (rest (rest EVENS)))

(empty? (rest (rest (rest (rest (rest EVENS))))))

(cons? (rest (rest (rest (rest (rest EVENS))))))
```
c. Write expressions using first, rest, cons, '(), and/or list abbreviations, and the two lists defined above – but no numbers – to produce the following:

- a list containing only the last number from the ODDS list

- the fourth element – not a list – from the EVENS list

- a list containing the second element of each list (i.e., 3, 4)
Problem 3

Do not use built-in higher-order functions for this problem.

a. Design a function to add up all the numbers in a list of numbers.

b. Design a function to multiply together all the numbers in a list of numbers.
c. Given the similarities in the functions for parts (a) and (b), define an abstract function, combine-nums that parameterizes their difference. Be sure to include a general signature for the combine-nums function.

d. Rewrite your functions from parts (a) and (b) so that they both use your combine-nums function. 

*Hint*: Each function's body should be a one-liner.
Problem 4

Recall the abstract filter function, shown with its signature here:

\[
\text{filter : } [X \rightarrow \text{Boolean}] \ [\text{List-of } X]
\]

a. First, develop the function odd-single-digit? which checks whether the number it consumes is only one digit long and is odd.

b. Next, use filter and odd-single-digit? (i.e., no cond expression or recursion) to develop the function filter-small-odds, which consumes a list of numbers and returns a list of only those elements that were small and odd. Be sure to include a signature and purpose statement.
c. Now, consolidate the functions you wrote on the previous page so that the odd-single-digit? function is defined within filter-small-odds using a local expression.

d. Finally, simplify filter-small-odds by removing the local expression altogether and replacing odd-single-digit? with its equivalent lambda expression.
Problem 5

Recall the abstract, higher-order functions given below:

\begin{align*}
\text{filter} & : [X \to \text{Boolean}], \ [\text{List-of} \ X] \to \ [\text{List-of} \ X] \\
\text{map} & : [X \to Y], \ [\text{List-of} \ X] \to \ [\text{List-of} \ Y] \\
\text{foldr} & : [X, Y \to Y], \ Y, \ [\text{List-of} \ X] \to \ Y \\
\text{ormap} & : [X \to \text{Boolean}], \ [\text{List-of} \ X] \to \text{Boolean} \\
\text{andmap} & : [X \to \text{Boolean}], \ [\text{List-of} \ X] \to \text{Boolean}
\end{align*}

The following examples of lists are used below:

\begin{verbatim}
(define NUMS (list 1 2 3 4 5 6 7 8 9 10))
\end{verbatim}

Using only the higher-order functions given above, built-in functions (like odd? and even?) or lambda expressions you write yourself, write expressions for the following:

- Double the elements of NUMS

- Removes all the odd values from NUMS

- Removes all the even values from NUMS

- Sum the numbers in NUMS

- Determines whether all values in NUMS are odd

- Determines whether any values in NUMS are even