C211 Exam 2 – Fall 2015

Name: _____

Student Id (last 4 digits): _____

- You may use the usual primitives and expression forms from Intermediate Student Language with lambda; for everything else, define it. You may use any function defined in ISL with lambda, but you are *not* required to—the entire exam can be completed without any extra functions.
- You may write `c` → `e` for `(check-expect c e)`.
- We expect data definitions to appear in your answers, unless they're given to you. We expect you to follow the design recipe on all problems. In a number of cases, defining helper functions will be useful.
- If they are given to you, you do not need to repeat the signature and purpose statements in your implementations. Likewise, you do not need to repeat any test cases given to you, but you should add tests wherever appropriate. You do not need to write separate templates unless the problem explicitly asks for them. We expect that the functions you write follow the appropriate template, however.
- Some basic test taking advice: Before you start answering any problems, read *every* problem, so your brain can think about the harder problems in background while you knock off the easy ones.

Problem	Points	/out of
1		/ 14
2		/ 14
3		/ 12
4		/ 16
Total		/ **56**

Good luck!

Problem 1 Consider the data definition *Person* where a person has a name, age, and height in inches. 14 POINTS

```
; A Person is a structure: (make-person String Number Number)
(define-struct person (name age height))
```

1. Write two examples of *Person*s and one example of a *[Listof Person]*.

2. Design a function, `average-age`, that takes a list of *Person* and returns the average age of the people in list. Remember that the average of a sequence of numbers is the sum of all the numbers, divided by how many numbers there are. You may want to use the `length` function.

3. Design a function, `remove-young`, that removes all people from a list of people who are not at least the average age of the people in the list of people. Use `average-age`, but make sure to not call it with every recursive step. You may want to use the function `filter`.

[Here is some more space for the previous problem.]

Problem 2 14 POINTS

Consider the following data and structure definition for a music playlist

```
; A Playlist is one of:
; - empty
; - (make-playlist String String Playlist)
(define-struct playlist (song artist next-song))
```

1. List the signatures for the functions automatically created by this structure definition.

2. Define three different examples of playlists.

3. Design a template for a procedure to process a playlist.

4. Design a function `beginning` which accepts a playlist and a number n, and creates a new playlist such that the new playlist contains the first n songs from the given playlist. You can assume that the given playlist has at least n songs—you do not have to worry about what your function does if this is not the case.

Problem 3 Consider a family tree where each person has a name, a spouse, and some amount of children. 12 POINTS

1. Create a data definition *Person* where each *Person* has a name, information about whether or not the *Person* is married, and a collection of children. Note that a *Person* will not have information about who they are married to, if they are married.

2. Write an example family tree where someone in the family tree has a grandchild.

3. Design a function that counts the number of people in a tree who are married.

Problem 4 Consider the following skeletons for computing the sum and product of numbers in a list, each of which uses an accumulator-based helper function. 16 POINTS

```
;; sum : [Listof Number] -> Number
;; sum the numbers in the list
(define (sum lon)
  (sum/acc lon ...))
(check-expect (sum empty) 0)
(check-expect (sum (list 1 2 3)) 6)

;; prod : [Listof Number] -> Number
;; compute the product of the numbers in a list
(define (prod lon)
  (prod/acc lon ...))
(check-expect (prod empty) 1)
(check-expect (prod (list 1 2 3)) 6)
```

1. Design the `sum/acc` function. Be sure to state the accumulator invariant. You do not need to write new `check-expect` tests for this, but they may be helpful.

2. Design the `prod/acc` function. Be sure to state the accumulator invariant. You do not need to write new `check-expect` tests for this, but they may be helpful.

3. What should the argument to `sum/acc` be in the body of `sum`? What should the argument to `prod/acc` be in the body of `prod`?

4. Abstract `sum/acc` and `prod/acc` into a single function called `abs/acc`, following the design recipe for abstraction. As test cases, show how to replace the bodies of `sum` and `prod`.