# C211 Exam 1 – Fall 2014

Name: _____

Student Id (last 4 digits): _____

- You may use the usual primitives and expression forms from Beginning Student; for everything else, define it.
- 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.
- 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 background while you knock off the easy ones.

| **Problem** | Points | /out of |
|---|---|---|
| 1 | | / 20 |
| 2 | | / 15 |
| 3 | | / 12 |
| 4 | | / 9 |
| 5 | | / 19 |
| **Total** | | / **75** |

*Good luck!*

**Problem 1** The staff of C211 has been assigned to work as part-time zoo-keepers at the Bloomington Zoo. Their first task is feeding the monkeys. 20 POINTS

A *Fruit* can be a `"banana"`, an `"apple"`, or an `"orange"`.

1. Write the data definition for *Fruit*. Then write the data definition for a *ListOfFruit*. Write three examples of *ListOfFruit*.

2. Design a function `is-banana?` that determines if a given *Fruit* is a banana.

3. Design a function `enough-fruit?` that determines if the number of *Fruit* in a *ListOfFruit* is more than 5.

4. Design a function `feed-monkey` which takes a *ListOfFruit* and produces a new list which has all of the bananas removed.

**Problem 2** You're running a gift-wrapping service that can wrap box-shaped *Gift*s (which have a width, length, and height) and tube-shaped *Gift*s (which have a length and a diameter). For the purposes of wrapping, you don't need to know any other information about the *Gift*s.

15 POINTS

1. Create a data definition and structure definition appropriate for storing information about the *Gift*s. (All dimensions are measured in inches.)

2. Your research department has come up with the following formulas that determine how much wrapping paper is needed. Design the `total-paper` function that takes a *Gift* and computes the amount of wrapping paper needed to wrap it (measured in square inches).

   Here's the domain knowledge you need to implement this function. For a box-shaped *Gift*, you'll need $(L + H + 1)(2W + 2H + 1)$ square inches of paper (where $L$ is the length, $W$ is the width, and $H$ is the height). For a tube-shaped *Gift*, you'll need $(L + D + 1)(3.14D + 1)$ square inches of paper.

3. Design a new variant of a *Gift* for square gifts that have the same width, height, and length, and adapt the data definiton of *Gift* and your `total-paper` function to handle squares.

   A square *Gift* with side $S$ requires $(2S+1)(4S+1)$ square inches of paper.

   You do not have to repeat things you wrote earlier in this problem, but if you don't, you must make clear exactly what changes you want to make to data definitions, signatures, purpose statements, function definitions, and tests.

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

**Problem 3** The School of Informatics and Computing has declared tomorrow to be backwards day. To help them with this, you've been tasked to develop two functions that help things go backwards. 12 POINTS

To start with, here's the data definition for a list of numbers:

```
;; A ListOfNumber is one of:
;; - empty
;; - (cons Number ListOfNumber)
```

1. Design a function `snoc` that takes a *Number* and a *ListOfNumber* and adds that number to the list, at the **end**. Here are two examples:

```
(check-expect (snoc 6 empty) (list 6))
(check-expect (snoc 4 (list 3 2 6)) (list 3 2 6 4))
```

2. Using `snoc`, design the function `reverse` which puts the elements of a list backwards.

```
(check-expect (reverse empty) empty)
(check-expect (reverse (list 4 3 6 8)) (list 8 6 3 4))
```

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

**Problem 4** Consider the following data definition and structure definition. 9 POINTS

```
;; An Address is a structure:
;;   (make-address Number String)
(define-struct address (num street))
```

1. List the functions that are automatically created by this definition. (Hint: there are 4.)

2. Give two examples of *Address*es.

3. Design the function *neighbors?*, which consumes two *Address*es and determines if they are neighbors. Two addresses are neighbors if they have the same street name and the house number is different by exactly two. So 150 Woodlawn Ave and 148 Woodlawn Ave are neighbors, but 149 Woodlawn Ave is not the neighbor of either one.

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

**Problem 5** Here is a simple design and implementation for dictionaries that associate numbers to strings. 19 POINTS

```
;; A Dict is one of:
;;  - (make-empty-dict)
;;  - (make-item-dict Number String Dict)
(define-struct empty-dict ())
(define-struct item-dict (key val rest))

;; has-key? : Dict Number -> Boolean
;; Does this dict. have the given key?

(define (has-key? d k)
  (cond [(empty-dict? d) false]
        [else (or (= k (item-dict-key d))
                  (has-key? (item-dict-rest d) k))]))

;; lookup : Dict Number -> String
;; Produce the value associated with the given key in this dict.
;; Assume: the key exists in this dict. This means we don't have to
;; worry about empty dicts.

(define (lookup d k)
  (cond [(= k (item-dict-key d)) (item-dict-val d)]
        [else (lookup (item-dict-rest d) k)]))
```

1. Write two examples of *Dict*s, neither of which are empty.

2. Design the function `set`, which takes a *Dict* and a *Number* and a *String* and produces a new *Dict* that associates the given number with the given string.

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

The dictionary data structure has some important properties, one of which is that if we associate a value with a key using `set` and then look up that same key using `lookup`, we expect to get back the value we associated. For example, for any given dictionary, if we associate `1` with `"Fred"`, then look up `1`, we should get `"Fred"`. More generally, we expect this to be true for *any* key-value pair, not just `1` and `"Fred"`.

3. Write a `check-expect` that shows that this property is true for an example *Dict*, *Number*, and *String*.

Now consider a generalization of this property which says that if we set *two* keys to *two* values then look each of them up, we should get back the original values. For example, for any given dictionary, if we associate `1` with `"Fred"` and `2` with `"Wilma"`; then looking up `1` will produce `"Fred"` and looking up `2` will produce `"Wilma¨`. More generally, we can state this as a property for any two key-value pairs.

4. Does this new property hold for all dictionaries and key-value pairs? If so, why? If not, write a counter-example. If you think you have a counter-example, write it in the form of a `check-expect` that you think will fail.

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