COMP 280 : Assignment 1, sample solutions

1. (a) Assume $(A \cap B) \cup C = A \cap (B \cup C)$.
If x is an element of C,
then x is in $(A \cap B) \cup C$ (by definition of the union operation),
so it is in $A \cap (B \cup C)$ (since we assumed that $(A \cap B) \cup C = A \cap (B \cup C)$),
so it has to be in A (by definition of the intersection operation).
So $C \subset A$.

Assume $C \subset A$.
x is an element of $A \cap (B \cup C)$,
if and only if x is an element of A and an element of $B \cup C$ (by definition of the intersection operation),
if and only if x is an element of A and an element of B or C (by definition of the union operation),
if and only if x is an element of A and B or an element of A and C (by distributivity of the "and" logical operation over the "or" logical operation),
if and only if x is an element of $A \cap B$ or an element of $A \cap C$ (by definition of the intersection operation),
if and only if x is an element of $(A \cap B) \cup (A \cap C)$ (by definition of the union operation).
But $C \subset A$, so if y is an element of C, then y is an element of A, so y is an element of $A \cap C$. And if y is an element of $A \cap C$, then y is an element of C. So $A \cap C = C$.
So x is an element of $A \cap (B \cup C)$ if and only if x is an element of $(A \cap B) \cup C$.
So $(A \cap B) \cup C = A \cap (B \cup C)$.

(b) Counterexample: if B and C are empty, then $A \cap (B \cup C)$ is empty, but $(A \cup B) \cap (A \cup C)$ is A.

2. (a) Using the first representation:

```
#| A multiset is either:
      – empty
      – (cons E MS) where E is an element and MS is a multiset
|#
;; union : multiset multiset → multiset
;; computes the union of two multisets
(define (union ms1 ms2)
  (cond
    [(empty? ms1) ms2]
    [else (cons (first ms1) (union (rest ms1) ms2))]))
```

Using the second representation:

```
#| An elementRec is a structure (make-elt elt count) where elt
      is an element and count is a number.
   A multiset is either:
      – empty
      – (cons ER MS) where ER is an elementRec and MS is a multiset
        that does not ahve an elementRec for the element in ER.
|#
(define-struct elt (elt count))
```

```
;; element? : elementRec multiset → boolean
;; checks whether elementRec appears in multiset
(define (element? er ms)
  (cond
    [(empty? ms) false]
    [else (or (equal? (elt-elt er) (elt-elt (first ms)))
              (element? er (rest ms)))]))
;; update : elementRec multiset → multiset
;; looks up for an element in a multiset, and updates
;; its count. This function assumes that the element
;; is present in the multiset.
(define (update er ms)
  (if (equal? (elt-elt er) (elt-elt (first ms)))
     (cons (make-elt (elt-elt (first ms))
                     (+ (elt-count er) (elt-count (first ms))))
           (rest ms))
     (cons (first ms) (update er (rest ms)))))
;; union : multiset multiset → multiset
;; computes the union of two multisets
(define (union ms1 ms2)
  (cond
    [(empty? ms1) ms2]
    [else (if (element? (first ms1) ms2)
              (union (rest ms1) (update (first ms1) ms2))
              (cons (first ms1) (union (rest ms1) ms2)))]))
```

(b) First representation:

Let's prove that: (*union ms1 ms2*) is the union of multiset *ms1* and multiset *ms2*. We will prove this by induction on the structure of *ms1*, assuming any value for *ms2*.

If *ms1* is *empty*, then:

```
(union ms1 ms2)
=
(union empty ms2)
=
(cond
  [(empty? empty) ms2]
  [else (cons (first empty) (union (rest empty) ms2))])
=
(cond
  [true ms2]
  [else (cons (first empty) (union (rest empty) ms2))])
=
ms2
```

else, assume that *ms1* is (*cons elt rest-ms1*) for some *elt* and *rest-ms1*, and that (*union rest-ms1 ms2*) is the union of *rest-ms1* and *ms2*. Then:

(*union ms1 ms2*)
=
(*union* (*cons elt rest-ms1*) *ms2*)
=
(**cond**
  [(*empty?* (*cons elt restms1*)) *ms2*]
  [**else** (*cons* (*first* (*cons elt rest-ms1*)) (*union* (*rest* (*cons elt rest-ms1*)) *ms2*))])
=
(**cond**
  [*false ms2*]
  [**else** (*cons* (*first* (*cons elt rest-ms1*)) (*union* (*rest* (*cons elt rest-ms1*)) *ms2*))])
=
(*cons* (*first* (*cons elt rest-ms1*)) (*union* (*rest* (*cons elt rest-ms1*)) *ms2*))
=
(*cons elt* (*union rest-ms1 ms2*))

but then the induction hypotheses applies and we know that (*union rest-ms1 ms2*) is the union of the multiset *rest-ms1* and of the multiset *ms2*. Then the only element missing is *elt* and (*cons elt* (*union rest-ms1 ms2*)) is the union of multiset *ms1* and multiset *ms2*.

Second representation:

Let's prove first that (*element? er ms*) returns a boolean telling whether *er* appears in *ms*.

If *ms* is *empty* then:

(*element? er ms*)
=
(*element? er empty*)
=
(**cond**
  [(*empty? empty*) *false*]
  [**else** (**or** (*equal?* (*elt-elt er*) (*elt-elt* (*first empty*)))
            (*element? er* (*rest empty*)))])
=
(**cond**
  [*true false*]
  [**else** (**or** (*equal?* (*elt-elt er*) (*elt-elt* (*first empty*)))
            (*element? er* (*rest empty*)))])
=
false

else assume that *ms* = (*cons eltr rest-ms*), and that (*element? er rest-ms*) tests whether *er* is in *ms* or not. Then:

```
(element? er ms)
=
(element? er (cons eltr rest-ms)
=
(cond
  [(empty? (cons eltr rest-ms)) false]
  [else (or (equal? (elt-elt er) (elt-elt (first (cons eltr rest-ms))))
            (element? er (rest (cons eltr rest-ms))))])
=
(cond
  [false false]
  [else (or (equal? (elt-elt er) (elt-elt (first (cons eltr rest-ms))))
            (element? er (rest (cons eltr rest-ms))))])
=
(or (equal? (elt-elt er) (elt-elt (first (cons eltr rest-ms))))
    (element? er (rest (cons eltr rest-ms))))
=
(or (equal? (elt-elt er) (elt-elt eltr))
    (element? er (rest (cons eltr rest-ms))))
```

Then, either eltr and er contain the same element, or they don't. If they do, then:

```
(or (equal? (elt-elt er) (elt-elt eltr))
    (element? er (rest (cons eltr rest-ms))))
=
(or true
    (element? er (rest (cons eltr rest-ms))))
=
true
```

If they don't, then:

```
(or (equal? (elt-elt er) (elt-elt eltr))
    (element? er (rest (cons eltr rest-ms))))
=
(or false
    (element? er (rest (cons eltr rest-ms))))
=
(element? er (rest (cons eltr rest-ms)))
=
(element? er rest-ms)
```

which, by the induction hypothesis, provides the correct answer.

Let's prove that (*update er ms*) increases the count of the corresponding *elementRec* of *ms* by the value of the count of *er* and returns the updated list. We assume that the *elementRec* we are trying to update is present in the multiset.

Since the element in *er* has to be present in *ms*, *ms* cannot be *empty*. The base case of the induction is therefore when *ms* = (*cons eltr empty*) where the element contained in *eltr* has to be the element in *er*:

```
(update er ms)
=
(update er (cons eltr empty))
=
(if (equal? (elt-elt er) (elt-elt (first (cons eltr empty))))
   (cons (make-elt (elt-elt (first (cons eltr empty)))
                   (+ (elt-count er) (elt-count (first (cons eltr empty)))))
         (rest (cons eltr empty)))
   (cons (first (cons eltr empty)) (update er (rest (cons eltr empty))))
=
(if (equal? (elt-elt er) (elt-elt eltr))
   (cons (make-elt (elt-elt (first (cons eltr empty)))
                   (+ (elt-count er) (elt-count (first (cons eltr empty)))))
         (rest (cons eltr empty)))
   (cons (first (cons eltr empty)) (update er (rest (cons eltr empty))))
=
(if true
   (cons (make-elt (elt-elt (first (cons eltr empty)))
                   (+ (elt-count er) (elt-count (first (cons eltr empty)))))
         (rest (cons eltr empty)))
   (cons (first (cons eltr empty)) (update er (rest (cons eltr empty))))
=
(cons (make-elt (elt-elt (first (cons eltr empty)))
                (+ (elt-count er) (elt-count (first (cons eltr empty)))))
      (rest (cons eltr empty)))
=
(cons (make-elt (elt-elt eltr)
                (+ (elt-count er) (elt-count eltr)))
      empty)
```

Assume that *ms* = (*cons eltr rest-ms*) with *rest-ms* not being *empty*, and that (*update er rest-ms*) works if the element in *er* is present in *rest-ms*. Then:

```
(update er ms)
=
(update er (cons eltr rest-ms))
=
(if (equal? (elt-elt er) (elt-elt (first (cons eltr rest-ms))))
   (cons (make-elt (elt-elt (first (cons eltr rest-ms)))
                   (+ (elt-count er) (elt-count (first (cons eltr rest-ms)))))
         (rest (cons eltr rest-ms)))
   (cons (first (cons eltr rest-ms)) (update er (rest (cons eltr rest-ms)))))
=
(if (equal? (elt-elt er) (elt-elt eltr))
   (cons (make-elt (elt-elt (first (cons eltr rest-ms)))
                   (+ (elt-count er) (elt-count (first (cons eltr rest-ms)))))
         (rest (cons eltr rest-ms)))
   (cons (first (cons eltr rest-ms)) (update er (rest (cons eltr rest-ms)))))
```

Since the element in *er* appears somewhere in *ms*, then either it appears in *eltr* or in *rest-ms*. If it appears in *eltr*:

```
(if (equal? (elt-elt er) (elt-elt eltr))
    (cons (make-elt (elt-elt (first (cons eltr rest-ms)))
                    (+ (elt-count er) (elt-count (first (cons eltr rest-ms)))))
          (rest (cons eltr rest-ms)))
    (cons (first (cons eltr rest-ms)) (update er (rest (cons eltr rest-ms)))))
=
(if true
    (cons (make-elt (elt-elt (first (cons eltr rest-ms)))
                    (+ (elt-count er) (elt-count (first (cons eltr rest-ms)))))
          (rest (cons eltr rest-ms)))
    (cons (first (cons eltr rest-ms)) (update er (rest (cons eltr rest-ms)))))
=
(cons (make-elt (elt-elt (first (cons eltr rest-ms)))
                (+ (elt-count er) (elt-count (first (cons eltr rest-ms)))))
      (rest (cons eltr rest-ms)))
=
(cons (make-elt (elt-elt eltr)
                (+ (elt-count er) (elt-count eltr)))
      rest-ms)
```

If the element in *er* appears in *rest-ms*, then:

```
(if (equal? (elt-elt er) (elt-elt eltr))
    (cons (make-elt (elt-elt (first (cons eltr rest-ms)))
                    (+ (elt-count er) (elt-count (first (cons eltr rest-ms)))))
          (rest (cons eltr rest-ms)))
    (cons (first (cons eltr rest-ms)) (update er (rest (cons eltr rest-ms)))))
=
(if false
    (cons (make-elt (elt-elt (first (cons eltr rest-ms)))
                    (+ (elt-count er) (elt-count (first (cons eltr rest-ms)))))
          (rest (cons eltr rest-ms)))
    (cons (first (cons eltr rest-ms)) (update er (rest (cons eltr rest-ms)))))
=
(cons (first (cons eltr rest-ms)) (update er (rest (cons eltr rest-ms))))
=
(cons eltr (update er rest-ms))
```

which, by the induction hypothesis, provides the correct answer.

We just have now to prove that (*union ms1 ms2*) returns the union of the two multisets *ms1* and *ms2*.
Again, by induction on the structure of *ms1*, assuming any value for *ms2*:
If *ms1* is *empty*:

```
(union ms1 ms2)
=
(union empty ms2)
=
(cond
  [(empty? empty) ms2]
  [else (if (element? (first empty) ms2)
            (union (rest empty) (update (first empty) ms2))
            (cons (first empty) (union (rest empty) ms2)))])
=
(cond
  [true ms2]
  [else (if (element? (first empty) ms2)
            (union (rest empty) (update (first empty) ms2))
            (cons (first empty) (union (rest empty) ms2)))])
=
ms2
```

Else *ms1* = (*cons elt rest-ms1*), and if we assume that (*union rest-ms1 ms2*) is the union of the multisets *rest-ms1* and *ms2*, then:

```
(union ms1 ms2)
=
(union (cons elt rest-ms1) ms2)
=
(cond
  [(empty? (cons elt rest-ms1)) ms2]
  [else (if (element? (first (cons elt rest-ms1)) ms2)
            (union (rest (cons elt rest-ms1)) (update (first (cons elt rest-ms1)) ms2))
            (cons (first (cons elt rest-ms1)) (union (rest (cons elt rest-ms1)) ms2)))])
=
(cond
  [false ms2]
  [else (if (element? (first (cons elt rest-ms1)) ms2)
            (union (rest (cons elt rest-ms1)) (update (first (cons elt rest-ms1)) ms2))
            (cons (first (cons elt rest-ms1)) (union (rest (cons elt rest-ms1)) ms2)))])
=
(if (element? (first (cons elt rest-ms1)) ms2)
    (union (rest (cons elt rest-ms1)) (update (first (cons elt rest-ms1)) ms2))
    (cons (first (cons elt rest-ms1)) (union (rest (cons elt rest-ms1)) ms2)))
=
(if (element? elt ms2)
    (union (rest (cons elt rest-ms1)) (update (first (cons elt rest-ms1)) ms2))
    (cons (first (cons elt rest-ms1)) (union (rest (cons elt rest-ms1)) ms2)))
```

Now, either *elt* is in *ms2*, or it is not. If it is:

```
(if (element? elt ms2)
    (union (rest (cons elt rest-ms1)) (update (first (cons elt rest-ms1)) ms2))
    (cons (first (cons elt rest-ms1)) (union (rest (cons elt rest-ms1)) ms2)))
=
(if true
    (union (rest (cons elt rest-ms1)) (update (first (cons elt rest-ms1)) ms2))
    (cons (first (cons elt rest-ms1)) (union (rest (cons elt rest-ms1)) ms2)))
=
(union (rest (cons elt rest-ms1)) (update (first (cons elt rest-ms1)) ms2))
=
(union rest-ms1 (update elt ms2))
=
(union rest-ms1 ms2')
```

which, by the induction hypothesis, gives the correct result (note that, since we never assumed anything regarding *ms2*, the induction hypothesis can be used with *ms2'*).
If instead *elt* is not in *ms2*, then:

```
(if (element? elt ms2)
    (union (rest (cons elt rest-ms1)) (update (first (cons elt rest-ms1)) ms2))
    (cons (first (cons elt rest-ms1)) (union (rest (cons elt rest-ms1)) ms2)))
=
(if false
    (union (rest (cons elt rest-ms1)) (update (first (cons elt rest-ms1)) ms2))
    (cons (first (cons elt rest-ms1)) (union (rest (cons elt rest-ms1)) ms2)))
=
(cons (first (cons elt rest-ms1)) (union (rest (cons elt rest-ms1)) ms2))
=
(cons elt (union rest-ms1 ms2))
```

Then, by the induction hypothesis, (*union rest-ms1 ms2*) is the union of *rest-ms1* and *ms2*, and it also does not contain *elt*, so (*cons elt* (*union rest-ms1 ms2*)) is the expected union, which completes the proof.

3. (a) First model: a set of tuples (student name, college name) and a set of tuples (college name, total space).
   Second model: a set of colleges, with, for each college, a tuple containing the number of students assigned to that college, a set of their names, and the number of spaces left.
   The first model is conceptually the simplest one, and will be easier to use and to reason about, but performing queries will mean at least processing the whole set of student tuples, which will be slow if the set is big. The second model is more complex, but doing queries will simply mean looking for the right tuple in the college set, in which all the necessary information will be immediately available.

   (b) First implementation:

```
#| A RiceCollege is one of 'Baker, 'Brown, 'Hanszen, 'Jones, 'Lovett,
                           'SidRich, 'Weiss, 'WillRice
   A StudentRec is a structure (make-student name college)
   where name is a symbol and college is a RiceCollege
   A HousingInfo is either
   – empty
   – (cons SR HI) where SR is a StudentRec and HI is a HousingInfo
|#
```

```
(define RiceCollege (list 'Baker 'Brown 'Hanszen 'Jones 'Lovett
                          'SidRich 'Weiss 'WillRice))
(define-struct StudentRec (name college))
;; InCollege : HousingInfo RiceCollege → (listof StudentRec)
;; returns a list of records of all students assigned to the named college
(define (InCollege hi rc)
  (cond
    [(empty? hi) empty]
    [else (if (eq? rc (StudentRec-college (first hi)))
              (cons (first hi) (InCollege (rest hi) rc))
              (InCollege (rest hi) rc))]))
```

Second implementation, using an accumulator:

```
#| A RiceCollege is one of 'Baker, 'Brown, 'Hanszen, 'Jones, 'Lovett,
                           'SidRich, 'Weiss, 'WillRice
   A StudentRec is a structure (make-student name college)
   where name is a symbol and college is a RiceCollege
   A HousingInfo is either
   – empty
   – (cons SR HI) where SR is a StudentRec and HI is a HousingInfo
|#
(define RiceCollege (list 'Baker 'Brown 'Hanszen 'Jones 'Lovett
                          'SidRich 'Weiss 'WillRice))
(define-struct StudentRec (name college))
;; InCollegeAcc : HousingInfo RiceCollege (listof StudentRec) → (listof StudentRec)
;; returns a list of records of all students assigned to the named college,
;; using an accumulator
(define (InCollegeAcc hi rc acc)
  (cond
    [(empty? hi) acc]
    [else (if (eq? rc (StudentRec-college (first hi)))
              (InCollegeAcc (rest hi) rc (cons (first hi) acc))
              (InCollegeAcc (rest hi) rc acc))]))
;; InCollege : HousingInfo RiceCollege → (listof StudentRec)
;; returns a list of records of all students assigned to the named college,
(define (InCollege hi rc)
  (InCollegeAcc hi rc empty))
```

We will prove now that:

(*append* (*reverse* (*InCollege hi rc*)) *L*) = (*InCollegeAcc hi rc L*)

for any list L, by induction on the structure of hi.

If hi is *empty*, then:

```
(append (reverse (InCollege hi rc)) L)
=
(append (reverse (InCollege empty rc))L)
=
(append (reverse (cond
                    [(empty? empty) empty]
                    [else (if (eq? rc (StudentRec-college (first empty)))
                              (cons (first empty) (InCollege (rest empty) rc))
                              (InCollege (rest empty) rc))])) L)
=
(append (reverse (cond
                    [true empty]
                    [else (if (eq? rc (StudentRec-college (first empty)))
                              (cons (first empty) (InCollege (rest empty) rc))
                              (InCollege (rest empty) rc))])) L)
=
(append (reverse empty) L)
=
(append empty L)
=
L
```

And:

```
(InCollegeAcc hi rc L)
=
(InCollegeAcc empty rc L)
=
(cond
  [(empty? empty) L]
  [else (if (eq? rc (StudentRec-college (first empty)))
            (InCollegeAcc (rest empty) rc (cons (first empty) L))
            (InCollegeAcc (rest empty) rc L))])
=
(cond
  [true L]
  [else (if (eq? rc (StudentRec-college (first empty)))
            (InCollegeAcc (rest empty) rc (cons (first empty) L))
            (InCollegeAcc (rest empty) rc L))])
=
L
```

So the property holds.

If *hi* = (*cons first-hi rest-hi*), then:

```
(append (reverse (InCollege hi rc)) L)
=
(append (reverse (InCollege (cons first-hi rest-hi) rc)) L)
=
(append (reverse (cond
                   [(empty? (cons first-hi rest-hi)) empty]
                   [else (if (eq? rc (StudentRec-college (first (cons first-hi rest-hi))))
                             (cons (first (cons first-hi rest-hi)) (InCollege (rest (cons first-hi rest-hi)) rc))
                             (InCollege (rest (cons first-hi rest-hi)) rc))])) L)
=
(append (reverse (cond
                   [false empty]
                   [else (if (eq? rc (StudentRec-college (first (cons first-hi rest-hi))))
                             (cons (first (cons first-hi rest-hi)) (InCollege (rest (cons first-hi rest-hi)) rc))
                             (InCollege (rest (cons first-hi rest-hi)) rc))])) L)
=
(append (reverse (if (eq? rc (StudentRec-college (first (cons first-hi rest-hi))))
                     (cons (first (cons first-hi rest-hi)) (InCollege (rest (cons first-hi rest-hi)) rc))
                     (InCollege (rest (cons first-hi rest-hi)) rc))) L)
=
(append (reverse (if (eq? rc (StudentRec-college first-hi))
                     (cons (first (cons first-hi rest-hi)) (InCollege (rest (cons first-hi rest-hi)) rc))
                     (InCollege (rest (cons first-hi rest-hi)) rc))) L)
```

And:

```
(InCollegeAcc hi rc L)
=
(InCollegeAcc (cons first-hi rest-hi) rc L)
=
(cond
  [(empty? (cons first-hi rest-hi)) L]
  [else (if (eq? rc (StudentRec-college (first (cons first-hi rest-hi))))
            (InCollegeAcc (rest (cons first-hi rest-hi)) rc (cons (first (cons first-hi rest-hi)) L))
            (InCollegeAcc (rest (cons first-hi rest-hi)) rc L))])
=
(cond
  [false L]
  [else (if (eq? rc (StudentRec-college (first (cons first-hi rest-hi))))
            (InCollegeAcc (rest (cons first-hi rest-hi)) rc (cons (first (cons first-hi rest-hi)) L))
            (InCollegeAcc (rest (cons first-hi rest-hi)) rc L))])
=
(if (eq? rc (StudentRec-college (first (cons first-hi rest-hi))))
    (InCollegeAcc (rest (cons first-hi rest-hi)) rc (cons (first (cons first-hi rest-hi)) L))
    (InCollegeAcc (rest (cons first-hi rest-hi)) rc L))
=
(if (eq? rc (StudentRec-college first-hi))
    (InCollegeAcc (rest (cons first-hi rest-hi)) rc (cons (first (cons first-hi rest-hi)) L))
    (InCollegeAcc (rest (cons first-hi rest-hi)) rc L))
```

So we have to prove that:

```
(append (reverse (if (eq? rc (StudentRec-college first-hi))
                     (cons (first (cons first-hi rest-hi)) (InCollege (rest (cons first-hi rest-hi)) rc))
                     (InCollege (rest (cons first-hi rest-hi)) rc))) L)
=
(if (eq? rc (StudentRec-college first-hi))
   (InCollegeAcc (rest (cons first-hi rest-hi)) rc (cons (first (cons first-hi rest-hi)) L))
   (InCollegeAcc (rest (cons first-hi rest-hi)) rc L))
```

Now, if the first StudentRec in *hi* is assigned to *rc*, then:

```
(append (reverse (if (eq? rc (StudentRec-college first-hi))
                     (cons (first (cons first-hi rest-hi)) (InCollege (rest (cons first-hi rest-hi)) rc))
                     (InCollege (rest (cons first-hi rest-hi)) rc))) L)
=
(append (reverse (if true
                     (cons (first (cons first-hi rest-hi)) (InCollege (rest (cons first-hi rest-hi)) rc))
                     (InCollege (rest (cons first-hi rest-hi)) rc))) L)
=
(append (reverse (cons (first (cons first-hi rest-hi)) (InCollege (rest (cons first-hi rest-hi)) rc))) L)
=
(append (reverse (cons first-hi (InCollege rest-hi rc))) L)
= ;; to be proved
(append (append (reverse (InCollege rest-hi rc)) (cons first-hi empty)) L)
```

and:

```
(if (eq? rc (StudentRec-college first-hi))
   (InCollegeAcc (rest (cons first-hi rest-hi)) rc (cons (first (cons first-hi rest-hi)) L))
   (InCollegeAcc (rest (cons first-hi rest-hi)) rc L))
=
(if true
   (InCollegeAcc (rest (cons first-hi rest-hi)) rc (cons (first (cons first-hi rest-hi)) L))
   (InCollegeAcc (rest (cons first-hi rest-hi)) rc L))
=
(InCollegeAcc (rest (cons first-hi rest-hi)) rc (cons (first (cons first-hi rest-hi)) L))
=
(InCollegeAcc rest-hi rc (cons first-hi L))
= ;; by induction hypothesis
(append (reverse (InCollege rest-hi rc)) (cons first-hi L))
= ;; to be proved
(append (reverse (InCollege rest-hi rc)) (append (cons first-hi empty) L))
= ;; to be proved
(append (append (reverse (InCollege rest-hi rc)) (cons first-hi empty)) L)
```

So the property holds.
On the other hand, if the first StudentRec in *hi* is not assigned to *rc*, then:

```
(append (reverse (if (eq? rc (StudentRec-college first-hi))
                     (cons (first (cons first-hi rest-hi)) (InCollege (rest (cons first-hi rest-hi)) rc))
                     (InCollege (rest (cons first-hi rest-hi)) rc))) L)
=
(append (reverse (if false
                     (cons (first (cons first-hi rest-hi)) (InCollege (rest (cons first-hi rest-hi)) rc))
                     (InCollege (rest (cons first-hi rest-hi)) rc))) L)
=
(append (reverse (InCollege (rest (cons first-hi rest-hi)) rc)) L)
=
(append (reverse (InCollege rest-hi rc)) L)
```

and:

```
(if (eq? rc (StudentRec-college first-hi))
   (InCollegeAcc (rest (cons first-hi rest-hi)) rc (cons (first (cons first-hi rest-hi)) L))
   (InCollegeAcc (rest (cons first-hi rest-hi)) rc L))
=
(if false
   (InCollegeAcc (rest (cons first-hi rest-hi)) rc (cons (first (cons first-hi rest-hi)) L))
   (InCollegeAcc (rest (cons first-hi rest-hi)) rc L))
=
(InCollegeAcc (rest (cons first-hi rest-hi)) rc L)
=
(InCollegeAcc rest-hi rc L)
```

and, by the induction hypothesis, the two results are equal, again.

To complete the proof, we will now show the following relations (for some element *elt*, and some lists *L*, *L1*, *L2*, *L3*):

```
(cons elt L) = (append (cons elt empty) L)
(reverse (append L1 L2)) = (append (reverse L2) (reverse L1))
(append L1 (append L2 L3)) = (append (append L1 L2) L3)
```

We will use the following implementations of *reverse* and *append*:

```
;; reverse-acc : (listof any) (listof any) → (listof any)
;; reverses the order of the elements of a list, using an accumulator
(define (reverse-acc l acc)
  (cond
    [(empty? l) acc]
    [else (reverse-acc (rest l) (cons (first l) acc))]))
;; reverse : (listof any) → (listof any)
;; reverses the order of the elements of a list
(define (reverse l)
  (reverse-acc l empty))
```

```
;; append : (listof any) (listof any) → (listof any)
;; appends two lists
(define (append l1 l2)
  (cond
    [(empty? l1) l2]
    [else (cons (first l1) (append (rest l1) l2))]))
```

First, we are going to prove that these functions behave as expected.

For *reverse*:

If *l* is *empty*, then:

```
(reverse-acc empty acc)
=
(cond
  [(empty? empty) acc]
  [else (reverse-acc (rest empty) (cons (first empty) acc))])
=
(cond
  [true acc]
  [else (reverse-acc (rest empty) (cons (first empty) acc))])
=
acc
```

So:

```
(reverse empty)
=
(reverse-acc empty empty)
=
empty
```

If *l* = (*cons l1* (...(*cons ln empty*)...)), and if we assume that (*reverse-acc l acc*) = (*cons ln* (...(*cons l1 acc*)...)), then, for (*cons l0 l*), we have:

```
(reverse-acc (cons l0 l) acc)
=
(cond
  [(empty? (cons l0 l)) acc]
  [else (reverse-acc (rest (cons l0 l)) (cons (first (cons l0 l)) acc))])
=
(cond
  [false acc]
  [else (reverse-acc (rest (cons l0 l)) (cons (first (cons l0 l)) acc))])
=
(reverse-acc (rest (cons l0 l)) (cons (first (cons l0 l)) acc))
=
(reverse-acc l (cons l0 acc))
```

Since the induction hypothesis is true for any value of *acc*, we have:

```
(reverse-acc l (cons l0 acc))
=
(cons ln (...(cons l1 (cons l0 acc))...))
```

So:

```
(reverse (cons l0 l))
=
(reverse-acc (cons l0 l) empty)
=
(cons ln (...(cons l1 (cons l0 empty))...))
```

Which completes the proof for *reverse*.

For *append*, by induction on *l1*:
If *l1* is *empty*:

```
(append l1 l2)
=
(append empty l2)
=
(cond
  [(empty? empty) l2]
  [else (cons (first empty) (append (rest empty) l2))])
=
(cond
  [true l2]
  [else (cons (first empty) (append (rest empty) l2))])
=
l2
```

otherwise l = (*cons first-l rest-l*), and if we assume that (*append rest-l l2*) behaves as expected, then:

```
(append l1 l2)
=
(append (cons first-l rest-l) l2)
=
(cond
  [(empty? (cons first-l rest-l)) l2]
  [else (cons (first (cons first-l rest-l)) (append (rest (cons first-l rest-l)) l2))])
=
(cond
  [false l2]
  [else (cons (first (cons first-l rest-l)) (append (rest (cons first-l rest-l)) l2))])
=
(cons (first (cons first-l rest-l)) (append (rest (cons first-l rest-l)) l2))
=
(cons first-l (append rest-l l2))
```

which, by the induction hypothesis, shows the result.

Now, with these definitions of *reverse* and *append*, we can prove:

```
(append (cons elt empty) L)
=
(cond
  [(empty? (cons elt empty)) L]
  [else (cons (first (cons elt empty)) (append (rest (cons elt empty)) L))])
=
(cond
  [false L]
  [else (cons (first (cons elt empty)) (append (rest (cons elt empty)) L))])
=
(cons (first (cons elt empty)) (append (rest (cons elt empty)) L))
=
(cons elt (append empty L))
=
(cons elt (cond
            [(empty? empty) L]
            [else (cons (first empty) (append (rest empty) L))]))
=
(cons elt (cond
            [true L]
            [else (cons (first empty) (append (rest empty) L))]))
=
(cons elt L)
```

If *L1* = (*cons l11* (...(*cons l1n empty*)...)) and *L2* = (*cons l21* (...(*cons l2m empty*)...)), then, since we have proven that *append* and *reverse* both behave as expected, we have:

```
(append (reverse L2) (reverse L1))
=
(append (reverse (cons l21 (...(cons l2m empty)...)))
        (reverse (cons l11 (...(cons l1n empty)...))))
=
(append (cons l2m (...(cons l21 empty)...))
        (cons l1n (...(cons l11 empty)...)))
=
(cons l2m (...(cons l21 (cons l1n (...(cons l11 empty)...)))...))
=
(reverse (cons l11 (...(cons l1n (cons l21 (...(cons l2m empty)...)))...)))
=
(reverse (append (cons l11 (...(cons l1n empty)...))
                 (cons l21 (...(cons l2m empty)...))))
=
(reverse (append L1 L2))
```

Note that this proof does not make any assumptions on the structure of *L1* or *L2*, and is not a proof by induction: the same proof works whether the lists are empty or not.

Finally:

$$
\begin{array}{l}
(append\ L1\ (append\ L2\ L3)) \\
= \\
(append\ (cons\ l11\ (\ldots(cons\ l1n\ empty)\ldots)) \\
\qquad\quad (append\ (cons\ l21\ (\ldots(cons\ l2m\ empty)\ldots)) \\
\qquad\qquad\qquad (cons\ l31\ (\ldots(cons\ l3k\ empty)\ldots)))) \\
= \\
(append\ (cons\ l11\ (\ldots(cons\ l1n\ empty)\ldots)) \\
\qquad\quad (cons\ l21\ (\ldots(cons\ l2m\ (cons\ l31\ (\ldots(cons\ l3k\ empty)\ldots)))\ldots))) \\
= \\
(cons\ l11\ (\ldots(cons\ l1n\ (cons\ l21\ (\ldots(cons\ l2m\ (cons\ l31\ (\ldots(cons\ l3k\ empty)\ldots)))\ldots)))\ldots)) \\
= \\
(append\ (cons\ l11\ (\ldots(cons\ l1n\ (cons\ l21\ (\ldots(cons\ l2m\ empty)\ldots)))\ldots)) \\
\qquad\quad (cons\ l31\ (\ldots(cons\ l3k\ empty)\ldots))) \\
= \\
(append\ (append\ (cons\ l11\ (\ldots(cons\ l1n\ empty)\ldots)) \\
\qquad\qquad\qquad (cons\ l21\ (\ldots(cons\ l2m\ empty)\ldots))) \\
\qquad\quad (cons\ l31\ (\ldots(cons\ l3k\ empty)\ldots))) \\
= \\
(append\ (append\ L1\ L2)\ L3)
\end{array}
$$

And again, no assumptions were made regarding the structure of *L1*, *L2*, *L3*.

As a really-final-now step, we have to show that
$(reverse\ (cons\ element\ L)) = (append\ (reverse\ L)\ (cons\ element\ empty))$
since this is what we actually used in the proof of the property for *InCollege* and *InCollegeAcc*. So (again using the fact that we have proved that *reverse* and *append* behave as expected, and using two of the three properties we have proved above):

$$
\begin{array}{l}
(append\ (reverse\ L)\ (cons\ element\ empty)) \\
= \\
(append\ (reverse\ L)\ (reverse\ (cons\ element\ empty))) \\
= \\
(reverse\ (append\ (cons\ element\ empty)\ L)) \\
= \\
(reverse\ (cons\ element\ L))
\end{array}
$$

Which completes the proof.