

## Notes on Exam 1

This page may be found online at

<http://www.cs.grinnell.edu/~rebelsky/Courses/CS151/2001S/Exams/notes.01.html>.

### Useful Files

- `prob1.ss`
- `prob2.ss`
- `prob3.ss`
- `prob4.ss`

## Some General Grading Issues

Many of the following issues appeared in many problems. Since I didn't consider it appropriate to penalize you multiple times for the same error, I tended to take off in problem 1 and then let the issue slide for the remaining problems. I will not be so generous in the next exam.

### Indentation

As I've said many times, proper indentation makes code much more readable (and improper indentation makes code much less readable). I took off if I saw bad indentation.

You can always get DrScheme to indent for you by hitting tab in each line (starting from the first line of a procedure and working your way down through the procedure).

### Line width

A number of you found that your code looked significantly different when it appeared in printed or email form (and if you didn't, I did). Most printers limit themselves to 80-or-so characters per line. Most email programs limit themselves to even fewer.

You make your code and comments more readable if you make it a habit to design it in such a way that it does not take more than 70 characters per line (including spaces). For comments, you simply have to hit a carriage return regularly. For non-comments, you need to be careful about where you break up complex expressions.

I did not take off for line width in this exam. I will in the next.

### Return values and postconditions

You need to be very careful about your postconditions. Typically, your postconditions speak to the value of the return value. Hence, in the section of your documentation headed "Produces" you should name that return value (and, probably, speak to its type). Then, in the postconditions, you can clarify what value it should have.

For `nested-tally`, many of you had but one postcondition: “returns an integer” (or something similar). If that’s the only thing you’re willing to guarantee about `nested-tally`, then it’s pretty easy to write it, since the following procedure meets that postcondition.

```
(define nested-tally
  (lambda (value structure)
    1))
```

Similarly, for `lookup`, many of you had only one postcondition for `lookup` that said something like “every element of `results` contains `key` in position `pos`”. While that certainly should be one postcondition, it should not be the only postcondition. Suppose we had a `make-list` procedure that works a lot like `make-string` in that it makes a list of a specified length containing multiple copies of the same value. If we used only your one postcondition, we could write `lookup` as:

```
(define lookup
  (lambda (pos key database)
    (make-list 2 (make-list (+ pos 1) database))))
```

We could also write

```
(define lookup
  (lambda (pos key database)
    null))
```

since every element of the null list meets the criterion that the desired key is at the desired position.

*Yes, I know I make similar mistakes some times. Let’s all agree to be much more careful on our postconditions.*

## The Six P’s

Here’s my recommended format for the six P’s

```
;;; Procedure:
;;;   Name the procedure
;;; Parameters:
;;;   Name and give a type to each parameter.
;;;   Only write one parameter per line.
;;; Purpose:
;;;   Describe the purpose of the procedure.
;;; Produces:
;;;   Name and give a type to the return value, if it has one.
;;; Preconditions:
;;;   Requirements for the form of parameters.
;;;   Requirements for other things.
;;; Postconditions:
;;;   Guarantees about the form of the result.
;;;   Guarantees about the type of the result.
;;;   Guarantees about the state of the program’s world.
```

## Document, write, and test

For almost every problem in which you were to write code, I said that you should “document, write, and test” the procedure. The order was intentional: I strongly recommend that you write the documentation first so that you know what you’re supposed to do. Next you write the procedure. Finally, you do *careful* testing (using a variety of legal and illegal inputs) to make sure that the procedure works correctly. Start with some simple tests before working up to complex tests. You don’t need to show me all of your tests, but you should show me some of them. I’ve given an example of a fairly comprehensive test suite in problem 1.

Since I told you to test your procedures, I did take off if you did not include evidence of testing.

I appreciate it if you comment out your tests (both the input and the result) so that when I execute your file, I just end up with the procedure definitions.

### **Naming your procedures**

In many problems, I tell you what to call your procedure. Please use the requested name. When you use the requested name, I can run your program through an automatic test suite to help identify problems. (No, I didn’t do so on this exam, but I might in the future.)

### **Needless complications**

I saw a number of procedures that had things like

```
... (+ a) ...  
... (+ 0 a) ...
```

Since both of these are equal to `a`, I’m not sure what the advantage of using the addition is.

### **In-code comments**

Unless you believe that your code is perfectly clear, it’s nice to include some basic comments to help someone understanding your thinking.

I took off points if you did not include such comments.

### **Don’t use quote to create lists**

I’ve said many times that you should use `cons`, `append`, or `list` to create lists. Do *not* use the single-quote mark. Why not? Because you will often get unexpected results. For example,

```
> (symbol? (car '(a)))  
#t  
> (symbol? (car '('a)))  
#f
```

### **Spelling and grammar**

You wouldn’t turn a paper in to someone in the humanities or social sciences with many misspelled words and incorrect grammar, would you? Then why do you do it in my class? I understand that you will make the occasional mistake, but many of you gave me exams that were full of misspellings and bad grammar.

I'm letting it pass this time, but I won't let it pass the next time.

By the way, the plural of ‘parenthesis’ is ‘parentheses’.

## Problem 1: Tallying in Nested Structures

In the past, we've seen that it is often useful to count the number of times a particular value appears in a list. Now that we know how to build more complex structures (e.g., lists of lists of lists), it may also be useful to count the number of times a value appears anywhere in a structure. For example, "Sam" appears 4 times in

```
(( "Sam" ("James" Jane" ))
 ("Jack" "Jill" ("Sam" ("Sam")) )
 "Sam"
 "William")
```

Document, write, and test a procedure, (*nested-tally value structure*), that counts how many times *value* appears anywhere in *structure*.

### A Solution

As is the case of all recursive procedures you write, I recommend that you start by approaching this problem with four questions, which I've duplicated and answered below. Typically, you answer the questions in order, with your answer to one possibly forcing you to rethink your answers to the other ones.

#### 1. How can I break up the problem?

Alternately: *How can I make the problem smaller?*

In this case, the only kinds of structures we know how to break up are those that involve cons cells (pairs). I suppose we also know how to break up strings, but that seems a little bit involved for this point in the semester.

Note that we tend to break a cons cell into three parts: the cons cell itself, the car, and the cdr.

#### 2a. How do I know when I've reached a base case?

Alternately: *When can't I break up the problem?*

One can't take the car and cdr of a non-pair. So we'll make the base case something involving non-pairs.

#### 2b. What value should you return when you've reached a base case?

If we don't have a pair, we have some other kind of value (symbol, string, null, procedure, character, ...). If that value is equal to the value we're looking for, we should count it as 1. Otherwise, we should count it as 0.

### 3. How do I write the recursive calls?

This is essentially a “can I figure out how to recurse?” question. Since you know how you’ve broken up the problem, you should try the most straightforward recursive call first.

In this case, I need to do two recursive calls:

- one for the car, (nested-tally value (car structure))
- one for the cdr, (nested-tally value (cdr structure))

### 4. Assume the recursive calls work correctly. How do they contribute to the final result?

If we’ve counted how many times the value appears in the car and how many times the value appears in the cdr, then the total number of times it appears is just the sum of those two things.

Okay, now we’re ready to put the solution together.

```
;;; Procedure:
;;; nested-tally
;;; Parameters:
;;; value, a basic value
;;; structure, a value (including, potentially, a nested structure)
;;; Purpose:
;;; Counts the number of times that value "appears" in structure.
;;; Produces:
;;; tally, a non-negative integer
;;; Preconditions:
;;; value is a basic value (not a pair).
;;; Postconditions:
;;; tally counts the number of times that value appears in structure.
;;; A value appears in a basic value (non-pair) if and only if the two
;;; values are equal. (That is, a substring does not appear in a string).
;;; A value appears in a structure with a pair at the root if it appears
;;; in either the car or the cdr of that structure.
(define nested-tally
  (lambda (value structure)
    ; Base case: Not a pair
    (if (not (pair? structure))
        ; Down to a single element. Does it match? If so, it counts
        ; for 1. If not, it counts for 0.
        (if (equal? value structure) 1 0)
        ; Okay, it's a pair, so count the car, count the cdr, and shove
        ; the results together.
        (+ (nested-tally value (car structure))
           (nested-tally value (cdr structure))))))
```

Here are some tests in the form that I prefer for tests.

```
; Does it work correctly for empty structures?
; (nested-tally 'a null)
; => 0
;
; Does it work correctly for matching singletons?
; (nested-tally 'a 'a)
```

```

; => 1
; (nested-tally "a" "a")
; => 1
;
; Does it work correctly for non-matching singletons?
; (nested-tally 'a 'b)
; => 0
; (nested-tally "a" 'a)
; => 0
; (nested-tally 'a "a")
; => 0
;
; Does it work for flat lists in which the value appears
; everywhere?
; (nested-tally 'a (list 'a))
; => 1
;
; Does it work for flat lists in which the value appears nowhere?
; (nested-tally 'a (list 'b))
; => 0
; (nested-tally 'a (list 'b 'c 'd 'e))
; => 0
;
; Does it work for flat lists in which the value appears in some places
; (including just at the front and just at the end)?
; (nested-tally 'a (list 'a 'b 'c 'd))
; => 1
; (nested-tally 'a (list 'b 'c 'a))
; => 1
; (nested-tally 'a (list 'b 'a 'c))
; => 1
; (nested-tally 'a (list 'a 'b 'c 'a 'b 'a 'd))
; => 3
; (nested-tally 'a (list "a" 'a 3 #\a))
; => 1
;
; Does it work using null as the value I'm searching for?
; (nested-tally null null)
; => 1
; (nested-tally null (list null))
; => 2
; This counts the "hidden" null at the end of the list.
; (nested-tally null (list 'a null 'b))
; => 2
; So does this.
;
; Can I look for procedures?
; (nested-tally nested-tally (list 'a nested-tally 'b 'c))
; => 1
;
; Can I look in non-lists?
; (nested-tally 'a (cons 'a 'a))
; => 2
; (nested-tally 'a (cons 'a 'b))
; => 1
; (nested-tally 'a (cons 'b 'a))
; => 1

```

```

; (nested-tally 'a (cons 'b 'b))
;   => 0
;
; At this point, I'm fairly confident that the procedure works fine, no
; matter which basic type I use. Hence, further testing will primarily
; use atomic symbols (although I may also do some other tests with
; null.
;
; Try one level of nesting in lists.
; (nested-tally 'a (list (list 'a 'b) 'c))
;   => 1
; (nested-tally 'a (list (list 'b 'a) 'c))
;   => 1
; (nested-tally 'a (list 'b (list 'a 'b)))
;   => 1
; (nested-tally 'a (list 'b (list 'b 'a)))
;   => 1
; (nested-tally 'a (list 'a (list 'a 'a)))
;   => 3
; (nested-tally 'a (list 'a (list 'b 'a 'c)))
;   => 2
; (nested-tally 'a (list (list 'a 'b) (list 'b 'a) (list 'a 'c)))
;   => 3
;
; Try one level of nesting in pairs
; (nested-tally 'a (cons (cons 'a 'a) 'a))
;   => 3
; (nested-tally 'a (cons 'a (cons 'a 'a)))
;   => 3
; (nested-tally 'a (cons (cons 'a 'a) (cons 'a 'a)))
;   => 4
;
; Okay, try some deeper nesting.
; (nested-tally 'a (list (list (list 'a) 'b)))
;   => 1
; (nested-tally 'a (list 'b (list (list 'a 'b 'c 'a 'd) (list 'e)) (list 'c 'a 'b)))
;   => 3
; (nested-tally 'a (cons (cons (cons 'b 'a) (cons 'a 'c))
;   (cons 'b (cons (cons 'a 'c) 'd))))
;   => 3

```

No, I don't expect you to turn in that detailed testing. I do, however, expect you to test more than a few cases.

## Grading

### Structures that are not lists

A number of you wrote code that would not work for any of the following:

```
> (nested-tally 'a 'a)
1
> (nested-tally 'a 'b)

> (nested-tally 'a (cons 'a 'a))
2
```

Why wouldn't your code work? Because you decided that the only kinds of structures you knew about were lists. However, we'd just finished discussing cons cells, so you should have known about non-list structures.

I took off 2 points if you wouldn't do the examples I just gave. I took off only one if you carefully documented (in your preconditions) that your procedure only accepted lists.

### Using `list?`

A number of you used `list?` in what I'd call a "willy nilly" fashion. That is, you didn't really use it to check preconditions. Rather, you used it to check something about some substructure. As you should know from our discussions of what makes a list, `list?` is a very expensive predicate as it must step through every cons cell until it finds (or fails to find) `null`. You are much better off using `pair?`.

I took off 2 points if you used `list?` in a somewhat ad-hoc manner.

### Checking preconditions

A few of you decided to carefully check your preconditions and report an error if the preconditions were met. If you were using `list?` to check preconditions, I looked to see that you were using husk-and-kernel programming, since that's a costly procedure.

Since precondition testing was not part of this exam, I awarded a few points of extra credit for testing your preconditions here and elsewhere on the exam.

### Incorrect code

As you might guess, I took off for incorrect code. The amount I took off tended to depend on how incorrect the code seemed to be and whether or not you'd indicated that you knew that the code was incorrect.

## Problem 2: Improving `assoc`

As we've noted in our exercises with searching in lists of data, it is sometimes inconvenient that `assoc` assumes that the key is the car of an element. It would be useful to tell `assoc` (or a similar procedure) where in each element to find the key (assuming that each element is itself a list).

For example, suppose we had a list of people in which each person is represented by the list

```
(firstname lastname birthday)
```

If we wanted to search by last name, we'd tell the procedure to look in position 1. Similarly, if we wanted to search by birthday, we'd tell the procedure to look in position 2. (Remember, Scheme counts positions starting with position 0.)

You may recall that we also decided that we often want to get *all* of the matching entries, rather than just the first one. For example, if two people have the last name Smith and I look for the last name Smith, I should get a list of those two people.

Document, write, and test a procedure, (`lookup position key database`) that returns a list of all the entries in *database* that match the key at the specified position.

## Note 2.1: Using `list-ref`

Although I generally discourage students from using `list-ref`, you can feel free to use it for this problem. [22 February 2001]

## Note 2.2: Comparing Values

For this problem, you should compare values with `equal?`. [22 February 2001]

## A Solution

### 1. How can I break up the problem?

It doesn't make sense to change the position or the key, so the only thing to break up will be the database. Since the database is a list of entries, we'll break it up into the first entry (the `car`) and the remaining entries (the `cdr`).

### 2a. How do I know when I've reached a base case?

One can't take the `car` and `cdr` of a non-pair. So we'll make the base case something involving non-pairs. Here, we know we're dealing with lists, so we'll make the base case that the database is empty.

### 2b. What value should you return when you've reached a base case?

Well, nothing is in the empty database, so we return the empty list. How do I know that I'm supposed to return a list? I'd thought about the return type.

### 3. How do I write the recursive calls?

This is essentially a "can I figure out how to recurse?" question. Since you know how you've broken up the problem, you should try the most straightforward recursive call first.

I can certainly recurse on the `cdr` of the database, since that's another database. I'll use the same position and key.

```
(lookup pos key (cdr database))
```

The entry is also a list. Should I recurse on the entry, too? No. It's a different kind of list than the database. While the database is a list of entries (in which the order doesn't really matter), each entry is a list of fields in which the position of a value gives its meaning.

What should I do with the entry? See if it matches the desired key at the desired position. (Yes, this isn't really a recursive call. You could include this note just as easily in the next step.)

```
(if (equal? key (list-ref entry pos)) ...)
```

#### 4. Assume the recursive calls work correctly. How do they contribute to the final result?

If we know all the times the desired key appears in the remainder of the list and we know whether or not the key appears in the current element, we can just join them together.

#### Putting it all together

```
;;; Procedure:
;;; lookup
;;; Parameters:
;;; pos, a non-negative integer
;;; key, a value
;;; database, a list of lists
;;; Purpose:
;;; Extract all elements of the database that match the key
;;; in the specified position.
;;; Returns:
;;; result, a list of lists
;;; Preconditions:
;;; position is a non-negative integer
;;; database is a list of entries, where each entry is a list
;;; each entry in database is of length at least pos+1
;;; Postconditions:
;;; All lists in result are also in database.
;;; For any list in result, the value at position pos is
;;; equal to key.
;;; For any list in database that is not in result, the value at
;;; position pos is not equal to key.
;;; Does not modify any of the parameters.
(define lookup
  (lambda (pos key database)
    (cond
      ; Base case, the database is empty. Return the empty list
      ((null? database) null)
      ; If the key is at the appropriate place in the current element,
      ; keep that element and then look through the rest.
      ((equal? key (list-ref (car database) pos))
       (cons (car database) (lookup pos key (cdr database))))
      ; Otherwise, just look through the rest.
      (else (lookup pos key (cdr database))))))
```

## Grading

### Structure of the database

For this problem, I looked to see that you had carefully written the preconditions to `lookup`. Since you were looking at a particular position in each entry list, it was important that you made sure that each entry had at least `position+1` elements.

If you didn't write this precondition, I took off 2 points.

### The form of the result

As I suggested in the earlier note on postconditions, I had hoped to see a careful specification of the results (each entry in the result list is a member of database, etc.). However, since I consider that specification particularly hard to write, I did not penalize you for failing to specify the postcondition.

### Other issues

There were few other general issues in this problem.

## Problem 3: Computing Worth

Sarah and Steven Schemer are modern Americans who are tied to their things. They have decided to use Scheme to build a simple database to store descriptions of their possessions. As you might expect, they've decided to use a lists of lists for this information. Each individual list has the following form

```
(name-of-thing room how-many value-each)
```

That is, they have a short name for each kind of thing, the room in which that thing resides, how many of that thing they have in that room, and how much it would cost to replace each thing. They've chosen this representation because they find it easier to do inventory by room, and they find that some things appear in multiple rooms.

For example, here's a list of a few of the things they have.

```
(("iMac"      "Office"  1 500)
 ("Pen"       "Office"  20 35/100)
 ("Pen"       "Kitchen"  4 35/100)
 ("Printer"   "Office"  2 100)
 ("Toaster"   "Kitchen"  1 10)
 ("Coffee Mug" "Kitchen" 20 1))
```

Of course, having this list alone doesn't do them much good. They now want to know the total value of their assets.

a. Document, write, and test a procedure, (`item-value item`), that computes the value of the items described by one entry. That procedure should return 500 for the iMac in the office. Similarly, it should return 7 for the 20 pens worth 35 cents each that reside in the office.

```

> (item-value (list "iMac" "Office" 1 500))
500
> (item-value (list "Pen" "Office" 20 35/100))
7

```

b. Document, write, and test a procedure, (`total-value inventory`), that computes the total value of a list of items in the form described above. For the list above, the procedure should return 738.40 or 73840/100 ( $500+7+1.40+200+10+20$ ).

### Note 3.1: Form of Values

Even if you don't like fractions, you should make sure that these two procedures return *exact* numbers. If you want to see that pretty decimal point, you can follow a call to `item-value` or `total-value` with a call to `exact->inexact`. [22 February 2001]

### A Solution

For part a, the problem is rather straightforward. We simply have to multiply the number of items in a particular entry by the cost of each item in that entry. (A few of you were convinced that I was asking for something much harder. Sorry, this was intended to be a simple step on the way to the next half of the problem.)

```

;;; Procedure:
;;;   item-value
;;; Parameters:
;;;   entry, an entry in the inventory
;;; Purpose:
;;;   Compute the value of the given entry.
;;; Produces:
;;;   value, a number
;;; Preconditions:
;;;   entry is a four element list of the form
;;;     (name-of-thing room how-many value-each)
;;;   how-many is a positive whole exact number
;;;   value-each is a positive exact number
;;; Postconditions:
;;;   value is the value of the item or items described
;;;   in the entry.
(define item-value
  (lambda (entry)
    (* (list-ref entry 2) (list-ref entry 3))))

; Testing:
; (item-value (list "iMac" "Office" 1 500))
;   => 500
; (item-value (list "Pen" "Office" 20 35/100))
;   => 7
; (item-value (car schemers-stuff))
;   => 500

```

Okay, now we're ready to move on to the somewhat harder problem of computing the total value of the inventory. Let's start by making sure we know exactly what we want the procedure to do. Here's what I came up with. Your answer may differ slightly.

```
;;; Procedure:
;;;   total-value
;;; Parameters:
;;;   inventory, a list of inventory items
;;; Purpose:
;;;   Compute the total value of an inventory
;;; Produces:
;;;   value, an exact number
;;; Preconditions:
;;;   inventory is a list of entries
;;;   each entry is of the form
;;;     (name-of-thing room how-many value-each)
;;;   name-of-thing is a string
;;;   room is a string
;;;   how-many is a positive whole number
;;;   value-each is a positive exact number
;;; Postconditions:
;;;   value is the total value of the inventory.
```

Once again, we're now ready to ask ourselves the four basic questions about recursion.

### **1. How can I break up the problem?**

Alternately: *How can I make the problem smaller?*

We have decided to represent an inventory as a list of inventory entries. The first natural inclination for any list is to break it up into the car (one inventory entry) and the cdr (all the remaining entries in the inventory).

### **2a. How do I know when I've reached a base case?**

Alternately: *When can't I break up the problem?*

One can't take the car and cdr of a null list. We don't have to worry about anything else since we made one of our preconditions that the inventory is a list.

### **2b. What value should you return when you've reached a base case?**

If you own nothing, the total value of that which you own is 0.

### **3. How do I write the recursive calls?**

This is essentially a "can I figure out how to recurse?" question. Since you know how you've broken up the problem, you should try the most straightforward recursive call first.

I shouldn't recurse on the car, since the car is not a list of entries.

I can recurse on the `cdr`, since the `cdr` is a list of entries. (See, there are reasons we write the preconditions first. Now we know what kind of value is acceptable.) I can write that recursive call as

```
(total-value (cdr entries))
```

#### 4. Assume the recursive calls work correctly. How do they contribute to the final result?

If we've just figured out the value of everything but the first entry, and we know the value of the first entry, then we can just add them together to get the total value.

#### Putting it all together.

```
(define total-value
  (lambda (inventory)
    ; If there's nothing in the inventory, the total value is 0
    (if (null? inventory) 0
        ; Otherwise, add the value of the first entry to the
        ; value of the remaining entries
        (+ (item-value (car inventory))
           (total-value (cdr inventory))))))
```

Now we can try a few simple tests.

```
;(total-value schemers-stuff)
; => 3692/4
;(exact->inexact (total-value schemers-stuff))
; => 738.4
;(total-value (list (list "iMac" "Office" 1 500)
                    (list "Keyboards" "Office" 5 50)))
; => 750
```

## Grading

For this problem, I predictably penalized people for poor preconditions, primarily for forgetting to specify the form of each entry in the inventory. (Such a failure typically cost you two points.)

I also took off points for people who gave inexact results since I said that “you should make sure that these two procedures return *exact* numbers”. (This failure cost you three points.)

Since I like to see you build on prior work, I took off a point or two if you didn't bother to use `item-value` in your definition of `total-value`.

A terrifying number of you wrote the following

```
(define item-value
  (lambda (entry)
    (if (null? entry)
        0
        (* (list-ref entry 2) (list-ref entry 3)))))
```

I'm not sure why you're testing whether an entry is null, other than we expect entries to be lists. If your precondition is that it must be a valid entry, then you should check that it's a valid entry, not that it's not null. I tended to take off a point or two for an unexplained test like this.

## Problem 4: Valuing Partial Inventories

Sarah and Steven use your code and their list to determine the total value of their possessions. (Aren't you glad that you've furthered the cause of capitalism?) They take their summaries to their insurance agents and the agents say "We need this information broken down by room and object."

a. Document and write a procedure, (*room-value room inventory*) that computes the total value of all the items in a particular room. Assume that *inventory* may contain items from multiple rooms.

b. Document and write a procedure (*thing-value kind-of-thing inventory*) that computes the total value of all the items of a particular type (e.g., all pens). Assume that *inventory* may contain items of multiple types.

### Note 4.1: Reuse and Permission for Non-working Code

Your answers to this problem can assume that the procedures you wrote for the previous problems have been implemented, even if you have not successfully implemented those procedures. That is, your implementation of *room-value* and *thing-value* can include calls to *total-value* and *such*.

Even if your code doesn't work on those problems (which is not a good thing and will certainly give you no more than partial credit on those problems), you can still receive full credit for this problem.

This is the only programming problem on the exam for which you may potentially write non-working code. [22 February 2001]

## A Solution

These questions were intended to be relatively easy. More importantly, they were intended to build upon the prior questions. So, how do you get the value of all the things in a particular room? You extract all the things in a room and then total them up. Here's the code without the introductory comments.

```
(define room-value
  (lambda (room inventory)
    ; Extract only the inventory items in the room and then
    ; compute their total value.
    (total-value (lookup 0 room inventory))))
```

You can total all the items with a particular name using a similar process.

```
(define thing-value
  (lambda (thing
    ; Extract only the inventory items in the room and then
    ; compute their total value.
    (total-value (lookup 1 thing inventory))))
```

Here's the code in all its commented glory.

```
;;; Procedure:
;;;   room-value
;;; Parameters:
;;;   desired-room, a string
;;;   inventory, a list of inventory entries
;;; Purpose:
;;;   Compute the value of all the items in a particular room.
;;; Produces:
;;;   value, an exact number
;;; Preconditions:
;;;   inventory is a list of entries
;;;   each entry is of the form
;;;     (name-of-thing room how-many value-each)
;;;   name-of-thing is a string
;;;   room is a string
;;;   how-many is a positive whole number
;;;   value-each is a positive exact number
;;; Postconditions:
;;;   Returns the sum of the values of all the items that appear
;;;   in the desired room (that have the desired room as the room).
;;; Note:
;;;   Room name comparisons are case-sensitive, so, for example,
;;;   "office" and "Office" are treated as different values.
;;; Implementation:
;;;   The implementation is rather straightforward. We get a list
;;;   that contains only the entries for a particular room (using lookup)
;;;   and then we total the values of the entries in that list.
(define room-value
  (lambda (desired-room inventory)
    (total-value (lookup 1 desired-room inventory))))

; Examples
; (room-value "Office" schemers-stuff)
;   => 707
; (room-value "office" schemers-stuff)
;   => 0

;;; Procedure:
;;;   thing-value
;;; Parameters:
;;;   desired-thing, a string
;;;   inventory, a list of inventory entries
;;; Purpose:
;;;   Compute the value of all the items in a particular room.
;;; Produces:
;;;   value, an exact number
;;; Preconditions:
;;;   inventory is a list of entries
;;;   each entry is of the form
;;;     (name-of-thing room how-many value-each)
;;;   name-of-thing is a string
;;;   room is a string
;;;   how-many is a positive whole number
;;;   value-each is a positive exact number
;;; Postconditions:
```

```

;;; Returns the sum of the values of all the particular kind
;;; of thing.
;;; Note:
;;; Thing name comparisons are case-sensitive, so, for example,
;;; "office" and "Office" are treated as different values.
;;; Implementation:
;;; The implementation is rather straightforward. We get a list
;;; that contains only the entries for a particular thing (using lookup)
;;; and then we total the values of the entries in that list.
(define thing-value
  (lambda (desired-thing inventory)
    (total-value (lookup 0 desired-thing inventory))))

; Tests

; (thing-value "iMac" schemers-stuff)
; => 500
; Just one iMac. Okay.

; (thing-value "Coffee Mug" schemers-stuff)
; => 20
; Also correct for a single entry (this time with multiple objects)
; that is in the middle of the list.

; (thing-value "Pen" schemers-stuff)
; => 42/5
; (exact->inexact (thing-value "Pen" schemers-stuff))
; => 8.4
; Also correct for things that appear in multiple entries.

; (thing-value "Petrified Pizza" schemers-stuff)
; => 0
; Also correct for things that don't appear at all.

```

## Grading

Almost all of you got this right (either through the elegant technique described above or through less elegant techniques that look a whole lot like other procedures you'd written before). One of the goals here was for you to realize that you could build upon previously written procedures without starting from scratch. If you started from scratch, I took off a few points for inelegance.

## Problem 5: Programming Proverbs

Write down, explain, and illustrate (via examples) four programming proverbs that you've developed over the first few weeks of the semester. A programming proverb is a rule that you've found it pays to follow as you write, test, or debug code. A proverb is typically a sentence or two. The explanation for a proverb should be a short paragraph.

## Notes

You've heard enough proverbs from me that I'm not going to duplicate them here. I do plan to post the proverbs you folks came up with to the Web and let you vote on the best ones. (Please don't campaign for your solutions.)

## Grading

I was fairly lenient in grading these. As long as you had something helpful to say and said it in reasonably correct English, I gave you full or nearly-full credit. Next time, I'll look for better writing.

I did take off if you said something blatantly false or not very helpful.

Oh, one final proverb of my own, *Read the instructions!* A few of you missed out on extra credit because you did not carefully read the instructions. (Yes, I hid some extra credit stuff in the instructions.)