

Exam 1

Distributed: Friday, 23 February, 2001

Due: 9 a.m., Friday, 2 March, 2001

Those with extenuating circumstances may request an extension on the exam. Such requests must be received by Wednesday, 28 February 2001.

This page may be found online at

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

Preliminaries

There are five problems on this exam. Each problem is worth twenty points. Some problems may be broken up into subproblems. The point value associated with a problem does not necessarily correspond to the complexity of the problem or the time required to solve the problem. If you write down the amount of time you spend on each question, I'll give you three points of extra credit.

This examination is open book, open notes, open mind, open computer, open Web. Feel free to use all reasonable resources available to you. As always, you are expected to turn in your own work. If you find ideas in a book or on the Web, be sure to cite them appropriately.

This is a take-home examination. You may use any time or times you deem appropriate to complete the exam, provided you return it to me by the due date. It is likely to take you about five to ten hours, depending on how well you've learned topics and how fast your work. I expect that someone who has mastered the material and works at a moderate rate should have little trouble completing the exam in a reasonable amount of time. Since I worry about the amount of time my exams take, I will give three points of extra credit to the first two people who honestly report that they've spent at least eight hours on the exam. (At that point, I may then change the exam.)

You must include each of the following statements on the cover sheet of the examination. Please sign and date each statement. Note that the statements must be true; if you are unable to sign either statement, please talk to me at your earliest convenience. Note also that "inappropriate assistance" is assistance from (or to) anyone other than myself or our teaching assistant.

1. I have neither received nor given inappropriate assistance on this examination.
2. I am aware of no other students who have given or received inappropriate assistance on this examination.

Because different students may be taking the exam at different times, you are not permitted to discuss the exam with anyone until after I have returned it. If you must say something about the exam, you are allowed to say "This is among the hardest exams I have ever taken. If you don't start it early, you will have no chance of finishing the exam." You may also summarize these policies. You may not tell other students which problems you've finished.

Answer all of your questions electronically and turn them in in hardcopy. That is, you must write all of your answers on the computer and print them out. You should also email me a copy of your exam by copying your exam and pasting it into an email message. Put your answers in the same order that the problems appear in. Make sure that your solution confirms to the format for laboratory writeups

In many problems, I ask you to write code. Unless I specify otherwise in a problem, should write working code and include examples that show that you've tested the code.

You should fully document all of the primary procedures (including parameters, purpose, value produces, preconditions, and postconditions). If you write helper procedures (and you may certainly write helper procedures) you should document those, too, although you may opt to write less documentation. [Note added 24 February 2001]

I will give partial credit for partially correct answers. You ensure the best possible grade for yourself by emphasizing your answer and including a *clear* set of work that you used to derive the answer.

I may not be available at the time you take the exam. If you feel that a question is badly worded or impossible to answer, note the problem you have observed and attempt to reword the question in such a way that it is answerable. If it's a reasonable hour (before 10 p.m. and after 8 a.m.), feel free to try to call me in the office (269-4410) or at home (236-7445).

I will also reserve time at the start of classes next week to discuss any general questions you have on the exam.

Problems

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*.

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]

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.

```
(define schemers-stuff
  (list (list "iMac"      "Office"  1 500)
        (list "Pen"      "Office"  20 35/100)
        (list "Pen"      "Kitchen"  4 35/100)
        (list "Printer"  "Office"  2 100)
        (list "Toaster"  "Kitchen"  1 10)
        (list "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
> (item-value (car schemers-stuff))
```

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).

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

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]

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]

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.