

Exam 2: Recursion and Beyond

Distributed: Friday, March 16, 2007

Due: 10:00 a.m., Friday, April 6, 2007

No extensions.

This page may be found online at

<http://www.cs.grinnell.edu/~rebelsky/Courses/CS151/2007S/Exams/exam.02.html>.

This exam is also available in PDF format.

Contents

- Preliminaries
- Problems
 - Problem 1: Documenting Procedures
 - Problem 2: Separating a List
 - Problem 3: Membership, Revisited
 - Problem 4: Printing Lists and Other Values
- Some Questions and Answers
- Errors

Preliminaries

There are four problems on the exam. Some problems have subproblems. Each problem is worth twenty-five (25) points. 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.

This examination is open book, open notes, open mind, open computer, open Web. However, it is closed person. That means you should not talk to other people about the exam. Other than as restricted by that limitation, you should 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.

Although you may use the Web for this exam, you may not post your answers to this examination on the Web (at least not until after I return exams to you). And, in case it's not clear, you may not ask others (in person, via email, via IM, by posting a "please help" message, or in any other way) to put answers on the Web.

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.

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. In particular, this exam is likely to take you about four to six hours, depending on how well you've learned topics and how fast you work. *You should not work more than eight hours on this exam. Stop at eight hours and write "There's more to life than CS" and you will earn at least 75 points on this exam.*

I would also appreciate it if you would write down the amount of time each problem takes. Each person who does so will earn two points of extra credit. Since I worry about the amount of time my exams take, I will give two points of extra credit to the first two people who honestly report that they've spent at least five hours on the exam or completed the exam. (At that point, I may then change the exam.)

You must include both 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. You need not reveal the particulars of the dishonesty, simply that it happened. Note also that "inappropriate assistance" is assistance from (or to) anyone other than Professor Rebelsky (that's me) or Professor Davis.

1. I have neither received nor given inappropriate assistance on this examination.
2. I am not aware of any 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. You may not tell other students how long you've spent on the exam.

You must present your exam to me in two forms: both physically and electronically. That is, you must write all of your answers using the computer, print them out, number the pages, put your name on the top of every page, and hand me the printed copy. You must also email me a copy of your exam. You should create the emailed version by copying the various parts of your exam and pasting them into an email message. In both cases, you should put your answers in the same order as the problems. Failure to name and number the printed pages will lead to a penalty of two points. Failure to turn in both versions may lead to a much worse penalty.

In many problems, I ask you to write code. Unless I specify otherwise in a problem, you should write working code and include examples that show that you've tested the code. If you need helper procedures, make them local (using `let`, `letrec`, or named `let`).

Just as you should be careful and precise when you write code and documentation, so should you be careful and precise when you write prose. Please check your spelling and grammar. Since I should be equally careful, the whole class will receive one point of extra credit for each error in spelling or grammar you identify on this exam. I will limit that form of extra credit to five points.

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: Documenting Procedures

Topics: Documentation, code reading, local procedures, husk and kernel

Some programmers prefer to give their procedures and variables short names, such as `f` and `x`, rather than things like `find-in-tree?` and `tree-of-symbols`.

Unfortunately, such short names can obfuscate code. For example, consider the following useful procedure. It is difficult for most programmers to figure out what the procedure does and how it does what it does.

```
(define r
  (lambda (l)
    (letrec ((c (lambda (p v)
                  (let ((x (if (odd? v) 1 0)))
                    (cons (+ (car p) x) (+ (cdr p) (- 1 x))))))
             (s (lambda (q m)
                  (if (null? m)
                      (/ (car q) (cdr q))
                      (s (c q (car m)) (cdr m))))))
      (s (cons 0 0) l))))
```

- Change the various names in the procedure to clarify the roles of the various things. You should certainly rename `r`, `l`, `c`, `p`, `v`, `s`, `q`, and `m`. [5 points]
- Add internal comments to explain the various parts. [5 points]
- Add introductory comments (the six P's) to explain the purpose (and the other P's) of the procedure. [10 points]
- Add a husk that ensures that the preconditions are met and reports a separate error for each failed precondition. [5 points]

Problem 2: Separating a List

Topics: Local procedures, list recursion, documentation, unit testing

In this exercise, you will document, write, and test a Scheme procedure, `(unriffle lst)`, that takes a list as argument and returns a list of two lists, one comprising the elements in even-numbered positions in the given list, the other comprising the elements in odd-numbered-positions.

```
> (unriffle (list 'a 'b 'c 'd 'e 'f 'g 'h 'i))
((a c e g i) (b d f h))
> (unriffle (list))
(() ())
> (unriffle (list 'a))
((a) ())
> (unriffle (list 'b))
((b) ())
> (unriffle (list 'a 'b))
((a) (b))
```

- a. Document this procedure. [10 points]
- b. Using `unit-test.ss`, write a tester for this procedure. [5 points]
- c. Implement this procedure. [10 points]

Hint: It is likely that you will need a recursive kernel. If you think carefully about the parameters to that procedure, you are more likely to be successful.

Problem 3: Membership, Revisited

Topics: Deep recursion

As you may recall, we've regularly seen a `member?` procedure that can be used to determine whether or not a value appears in a list. Here is a common definition of that procedure.

```
;;; Procedure:
;;; member?
;;; Parameters:
;;; val, a Scheme value
;;; lst, a list of values
;;; Purpose:
;;; Determine whether val appears in lst.
;;; Produces:
;;; is-a-member, a Boolean
;;; Preconditions:
;;; (none)
;;; Postconditions:
;;; If there exists a position, p, such that
;;; (equals? val (list-ref lst p))
;;; then is-a-member is #t.
;;; If there is no such position, then is-a-member is #f.
(define member?
  (lambda (val lst)
    (and (not (null? lst))
         (or (equal? val (car lst))
             (member? val (cdr lst))))))
```

Write a new procedure, (`in-tree? val tree`), that determines whether or not *val* appears in *tree*. You can assume that *val* is not a pair structure, but rather something with a simpler structure (e.g., boolean, number, string, symbol, or procedure).

You need neither document nor write a test suite for the `in-tree?` procedure.

Problem 4: Printing Lists and Other Values

Topics: Output, List recursion, Scheme implementation

You may recall that I described the process that Scheme uses for printing list values:

- If the value is null, print ().
- If the value is not a pair, print the value as is.
- Otherwise, the value must be a pair, so
 - print an open paren,
 - print the car of the pair,
 - call a helper on the cdr of the pair, and
 - print a close paren.

The helper does the following:

- If the parameter is null, print nothing.
- If the parameter is not a pair, print a space, a period, another space, and then the parameter.
- Otherwise, the value must be a pair, so
 - print a space,
 - print the car of the pair, and
 - recurse on the cdr of the pair.

Implement a variant of this strategy in which the list items are separated by commas as well as spaces. That is, write a procedure `print`, that takes a Scheme value as a parameter and checks whether or not it is a pair. If it is not a pair, `print` can call `write` on the value. If it is a pair, `print` should print the open paren, print the car, call the helper on the cdr, and print the close paren. The helper, instead of printing a space, should print a comma and a space.

For example,

```
> (print 'a)
a
> (print "hello")
"hello"
> (print (list 'a 'b 'c))
(a, b, c)
> (print (cons "a" (cons 3 #\x)))
("a", 3 . #\x)
> (print (list (list 1 2 3) (list "hello" "goodbye")))
((1, 2, 3), ("hello", "goodbye"))
> (print null)
()
```

You need neither document nor write a unit test for `print`. You should, however, show some examples of the procedure as it generates output.

Some Questions and Answers

These are some of the questions students have asked about the exam and my answers to those questions.

Problem 1

Problem 2

Problem 3

Problem 4

Errors

Here you will find errors of spelling, grammar, and design that students have noted. Remember, each error found corresponds to a point of extra credit for everyone. I usually limit such extra credit to five points. However, if I make an astoundingly large number of errors, then I will provide more extra credit.

Copyright © 2007 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.