

Exam 3: Advanced Scheming

Distributed: Wednesday, 22 November 2006

Due: 9:00 a.m., Friday, 1 December 2006

No extensions.

This page may be found online at

<http://www.cs.grinnell.edu/~rebelsky/Courses/CS151/2006F/Exams/exam.03.html>.

This exam is also available in PDF format.

Contents

- Preliminaries
- Problems
 - Problem 1: Is it Sorted?
 - Problem 2: A map! for Vectors
 - Problem 3: Folding
 - Problem 4: Not Quite Binary Search
 - Problem 5: More Higher-Order Procedures
- Some Questions and Answers
 - Questions on Problem 1
 - Questions on Problem 2
 - Questions on Problem 3
 - Questions on Problem 4
 - Questions on Problem 5
 - Other Questions
- Errors

Preliminaries

There are five problems on the exam. Some problems have subproblems. Each problem is worth twenty (20) 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.

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: Is it Sorted?

Topics: Sorting, Orderings, Higher-order procedures, Objects

As you've probably noticed, there are two key postconditions of a procedure that sorts lists: The result is a permutation of the original list and the result is sorted. We're fortunate that the unit test framework lets us test permutations (with `test-permutation!`). Hence, if we wanted to test merge sort in the unit test framework, we might write

```
(define some-list ...)
(test-permutation! (merge-sort some-list pred?) some-list)
```

However, we still need a way to make sure that the result is sorted, particularly if the result is very long.

a. [5 points] Document a procedure, `(sorted? lst may-precede?)` that checks whether or not `lst` is sorted by `may-precede?`.

b. [5 points] Write that procedure.

For example,

```
> (sorted? (list 1 3 5 7 9) <)
#t
> (sorted? (list 1 3 5 4 7 9) <)
#f
> (sorted? (list "alpha" "beta" "gamma") string<?)
#t
```

Note that we can use that procedure in a test suite for any sorting routine with

```
(test! (sorted? (sort some-list may-precede?) may-precede?) #t)
```

c. [10 points] Suppose that we are representing students in this class with objects, as in the file `student-object.scm`.

Suppose also that we have stored all of the student records in a list, `CSC151`.

i. Write an expression that checks whether `CSC151` is sorted by student id (from alphabetically first to alphabetically last).

ii. Write an expression that checks whether `CSC151` is sorted by participation grade, from largest to smallest.

iii. Write an expression that checks whether `CSC151` is sorted by name, from alphabetically first to alphabetically last. (Students may have the same last name; two students with the same last name should be ordered by first name.)

iv. Write an expression that checks whether `CSC151` is sorted by average exam grade, from largest to smallest.

v. Write an expression that checks whether `CSC151` is sorted by graduation year (class) and, within graduation year, is sorted by last name. (You need not worry about first names.)

Problem 2: A `map!` for Vectors

Topics: Vectors, Higher-order procedures, `map`, Objects

As you may recall, the `map` procedure typically takes two parameters, a unary procedure and a list, and then builds a new list by applying the procedure to each element of the list. One definition of this simple version follows. (Advanced Schemers know that the built-in `map` can take more than two parameters; but that's irrelevant to this problem.)

```
(define map
  (lambda (proc lst)
    (if (null? lst)
        null
        (cons (proc (car lst)) (map proc (cdr lst))))))
```

As you've seen many times this semester, we often find it useful to write vector procedures that mimic (or at least are similar to) list procedures. For the case of `map`, we might write a `map!` that applies a procedure to each element of a vector, modifying the value in place.

For example,

```

> (define grades (vector 80 70 85 90))
> grades
#4(80 70 85 90)
> (map! (1-s + 5) grades) ; give everyone five points
> grades
#4(85 75 90 95)
> (map! (compose exact->inexact (r-s / 25)) grades) ; Divide grades by 25 for 4-point scale
> grades
#4(3.4 3.0 3.6 3.8)

```

- a. [5 points] Document the `map!` procedure.
- b. [10 points] Implement the `map!` procedure.
- c. [5 points] In `student-object.scm`, you will find an implementation of a simple student object. Build a vector of five randomly-generated student objects. Then, use `map!` to give each of them one additional homework grade, a plus.

Problem 3: Folding

Topics: Higher-order procedures

As you may recall, we began our investigation of recursion by considering how we might step through a list of numbers, adding them (or multiplying them or ...). Here are variants of the two versions we wrote:

One uses simple recursion:

```

(define sumr
  (lambda (numbers)
    (if (null? (cdr numbers))
        (car numbers)
        (+ (car numbers) (sumr (cdr numbers))))))

```

The other uses a helper that accumulates the results.

```

(define suml
  (letrec ((kernel
            (lambda (remaining so-far)
              (if (null? remaining)
                  so-far
                  (kernel (cdr remaining) (+ so-far (car remaining)))))))
    (lambda (numbers)
      (kernel (cdr numbers) (car numbers)))))

```

If you think carefully about it, the two processes compute results slightly differently. Given the list (`n0 n1 n2 n3 n4`), the first computes

```
(+ n0 (+ n1 (+ n2 (+ n3 n4))))
```

while the second computes

```
(+ (+ (+ (+ n0 n1) n2) n3) n4)
```

For addition, this doesn't make a difference. If the operation were subtraction, it would certainly make a difference.

Now, the process of inserting a binary operation into a list of values is quite common. We've seen it for addition, subtraction, and multiplication. We might also use it for appending strings and many other operations. Computer scientists have various names for such a process. We'll call it *fold*, but others use *insert* (as in "insert this procedure") or *reduce*.

a. [5 points] Write a procedure, `(foldr proc lst)` that mimics `sumr`, but with `proc` in place of `+`. That is, `(foldr proc (list v0 v1 v2 ... vn-1 vn))` should compute

```
(proc v0 (proc v1 (proc v3 (... (proc vn-1 vn) ... a))))
```

For example,

```
> (foldr + (list 1 2 3 4))
10
> (foldr * (list 1 2 3 4))
24
> (foldr - (list 1 2 3 4))
-2
> (foldr (lambda (s1 s2) (string-append s1 ", " s2))
      (list "sentient" "malicious" "powerful" "stupid"))
"sentient, malicious, powerful, stupid"
> (foldr list (list 'a 'b 'c 'd 'e))
(a (b (c (d e))))
```

b. [5 points] Write a procedure, `(foldl proc lst)` that mimics `suml`, but with `proc` in place of `+`. That is, `(foldl proc (list v0 v1 v2 ... vn-1 vn))` should compute

```
(proc ... (proc (proc (proc v0 v1) v2) v3) ... vn)
```

For example,

```
> (foldl + (list 1 2 3 4))
10
> (foldl * (list 1 2 3 4))
24
> (foldl - (list 1 2 3 4))
-8
> (foldl (lambda (s1 s2) (string-append s1 ", " s2))
      (list "sentient" "malicious" "powerful" "stupid"))
"sentient, malicious, powerful, stupid"
> (foldl list (list 'a 'b 'c 'd 'e))
((((a b) c) d) e)
```

c. [10 points] Some computer scientists prefer an alternate version of `fold`, which we'll call `folder`. Rather than taking two parameters, their `folder` takes only one, `proc`, a binary procedure. The `folder` procedure then returns a new, variable-arity, procedure, that inserts `proc` between its arguments. For example,

```

> (define comma-splice (lambda (s1 s2) (string-append s1 ", " s2)))
> (comma-splice "Fred" "Barney")
"Fred, Barney"
> (comma-splice "George" "Jane" "Elroy" "Judy")
procedure comma-splice: expects 2 arguments, given 4: "George" "Jane" "Elroy" "Judy"
> (comma-splice "Scooby-Doo")
procedure comma-splice: expects 2 arguments, given 1: "Scooby-Doo"
> (define comma-splicer (folderr (lambda (s1 s2) (string-append s1 ", " s2))))
> (comma-splicer "Fred" "Barney")
"Fred, Barney"
> (comma-splicer "George" "Jane" "Elroy" "Judy")
"George, Jane, Elroy, Judy"
> (comma-splicer "Scooby-Doo")
"Scooby-Doo"

```

Implement `folderr` or `folderl`.

Problem 4: Not Quite Binary Search

Topics: Binary search, Randomness, Counting Steps

As you may recall, the binary search procedure quickly searches through sorted vectors by keeping track of a region of the vector in which a value should fall. At each “step”, it identifies the midpoint of the region, grabs the value at that region, compares it to the desired value, and narrows the region.

Some have criticized this procedure for being too predictable. Rather than choosing the middle of the region, they think we should choose a “random” dividing point.

Let’s see what happens with this technique for vectors of integers (so that you don’t have to worry about what comparison routine to use). In particular, we’ll define a procedure (`rbst val vals`) that searches for `val` in the sorted vector of integers, `vals`, and returns an index of `val` if it appears or `#f` if it does not appear. (Note that `rbst` stands for “random binary search for integers”.)

- a. [5 points] Document this alternate version of binary search.
- b. [5 points] Implement this alternate version of binary search.
- c. [10 points] Traditional binary search takes approximately $\log_2 n$ recursive calls for a vector of length n . Experimentally determine whether this version takes approximately the same number of steps.

You may find the following procedure useful for building sorted vectors:

```

(define random-sorted-vector
  (letrec ((kernel
            (lambda (vec pos len seed)
              (if (= pos len)
                  vec
                  (begin
                     (vector-set! vec pos seed)
                     (kernel vec (+ pos 1) len (+ seed (random 5)))))))
    (lambda (len)
      (kernel (make-vector len 0) 0 len (random 5)))))

```

For example,

```
> (random-sorted-vector 15)
#15(3 3 7 9 10 12 15 17 21 25 27 28 30 34 36)
> (random-sorted-vector 15)
#15(3 6 7 10 11 12 13 17 20 22 24 28 31 31 35)
> (random-sorted-vector 15)
#15(0 3 4 6 7 8 11 14 16 19 23 26 30 33)
```

You may find the following procedure useful for figuring out the value of $\log_2 n$

```
(define log2 (lambda (x) (/ (log x) (log 2))))
```

For example,

```
> (log2 16)
4.0
> (log2 1024)
10.0
> (log2 1000)
9.965784284662087
> (log2 (* 1024 1024))
20.0
```

Problem 5: More Higher-Order Procedures

Topics: Higher-order procedures, map

Although we've used the map procedure with two parameters, a procedure and a list, it can take an arbitrary number of parameters. For example,

```
> (map list (list 1 2 3) (list 'a 'b 'c) (list "he" "she" "it"))
((1 a "he") (2 b "she") (3 c "it"))
> (map + (list 1 2 3) (list 4 4 4) (list 2 3 5) (list 100 200 300))
(107 209 312)
```

Scheme also provides an apply procedure, which acts similarly to the streme-apply we've written previously.

```
> (apply + (list 1 2 3))
6
> (apply sqrt (list 4))
2
```

a. [10 points]: By conducting some experiments (that is, by entering a variety of expressions and seeing their results) and by reading documentation, figure out what map does when given more than two parameters. Explain it in your own words. (Make sure to note any restrictions on the parameters.) Include at least three illustrative examples.

b. [10 points]: By conducting some experiments (that is, by entering a variety of expressions and seeing their results) and by reading documentation, figure out what apply does. Explain it in your own words. (Make sure to note any restrictions on the parameters.) Include at least three illustrative examples.

Some Questions and Answers

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

Questions on Problem 1

What are the elements of `csc151`?

It's a list of students. Here's a straightforward way to make a three element list of random students.

```
(define csc151 (list (random-student) (random-student) (random-student)))
```

How can we generate a longer list?

Write a longer definition. Write a procedure that generates an arbitrary length list of random students. Whatever you find most natural.

How should we handle situations in which the two values are equal and `may-precede?` is `<?`

Arguably, `may-precede?` should be `<=` rather than `<` (which is more accurately `precedes?`). However, you can do whatever you see fit, provided you document what you've done.

What should the expressions for part c look like?

```
(sorted? csc151 (lambda (s1 s2) ...))
```

In what order should graduation year and last name be sorted?

Graduation year should be in increasing order. Within each year, students should be in order by last name, from alphabetically first to alphabetically last.

How do I get access to the code from `student-object.scm`?

You could copy and paste the code into your exam or you could load that code with

```
(load  
  "/home/rebelsky/Web/Courses/CS151/2006F/Examples/student-object.scm")
```

Questions on Problem 2

What types should the `map!` procedure support?

The `map!` procedure should accept any vector as its second argument. The first argument should be a procedure. It is up to you to document any restrictions on the procedure argument.

Does `map!` need to support empty vectors?

Certainly.

How do I get access to the code from `student-object.scm`?

See the answer to the previous question.

Do you realize that `(student ' :add-homework! 'plus)` returns void?

Yes.

So, how do we use `map!` to modify the vector?

Write an anonymous procedure that both adds the grade and returns the modified student.

May I use `vector->list`?

No.

Questions on Problem 3

Can you give me a hint as to the structure for `folderr`?

Sure. You're writing a procedure of one parameter, and it needs to return a variable-arity procedure, so the form should be something like the following:

```
(define folderr
  (lambda (proc)
    (lambda (params
      ...)))
```

I've found that in implementing variable arity procedures, I often need a kernel. What if I want to write a recursive kernel, too?

Well, if you insist upon writing a recursive kernel, you probably want something like the following:

```
(define folderr
  (letrec ((kernel
    (lambda (proc params)
      ...)))
    (lambda (proc)
      (lambda (params
        ...))))
```

The tone of your previous answer suggests that you don't think that a kernel is necessary.

I don't. However, if the most natural way for you solve this is to use a recursive kernel, then use a recursive kernel. "Whatever floats your boat", as they say.

Doesn't the comma belong inside the question mark in the previous answer?

I use British style for quotations, rather than American style, because I feel that British style is more logical.

Do we get extra credit for making both `folderl` and `folderr`?

No.

Do `foldr` or `foldl` have to handle empty lists?

No.

Questions on Problem 4

I've found it easier to count only recursive calls, and not the outermost call. This means that when it guesses right on the first call, it seems to take 0 steps. Is it okay if we don't count the outermost call to our procedure?

Yes, that's okay.

Questions on Problem 5

Other Questions

Why does Dr. Davis have fewer questions on part 1.c?

Perhaps she's nicer than I am.

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.

- Random “a” in the middle of Scheme code. [SR, 1 point]
 - In the list of questions and answers, Sam wrote “list or random students” instead of “list of random students”. [DB, 1 point]
 - Sam is repetitious: “a region *of of* the vector”. [JP, 1 point]
 - Sam continues to be repetitious; he used two open quotes instead of one. [AB, 1 point]
 - Sam wrote add-grade! when he meant to write add-homework!. [PR, 1 point]
 - In the Q&A section about 1c, Sam got the parameters to sorted? in the wrong order; (sorted? pred lst) rather than the correct (sorted? lst pred). [JB]
-

Copyright © 2006 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.