

Exam 3

Distributed: Friday, April 27, 2007

Due: 10:00 a.m., Friday, May 4, 2007

No extensions.

This page may be found online at

<http://www.cs.grinnell.edu/~rebelsky/Courses/CS151/2007S/Exams/exam.03.html>.

This exam is also available in PDF format.

Contents

- Preliminaries
- Problems
 - Problem 1: Tallying, Revisited
 - Problem 2: Fun with Higher-Order Procedures
 - Problem 3: All Sorts of Sorting
 - Problem 4: Finding the Nth Smallest Element of a List
- 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 80 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: Tallying, Revisited

Topics: Higher-Order Procedures, Vectors, Files, Sequential Search, Binary Trees, Deep Recursion

In a recent homework assignment, you wrote a procedure, `(tally pred? lst)` that, given a predicate and a list, computes the number of values in the list for which the predicate holds.

As you've noted before, Scheme provides a variety of ways to collect data. For example, in addition to collecting values in a list, we might collect them in a vector, file, or tree.

a. [10 points] Implement a procedure, `(tally-vector pred? vec)`, which, given a unary predicate and a vector, counts the number of values in the vector for which the predicate holds. For example,

```
> (tally-vector (1-s <= 5) (vector 6 4 1 2 6 9 4 4 1 0))
3
> (tally-vector symbol? (vector 2 'too "two" 'tutu))
2
> (tally-vector odd? (vector))
0
```

b. [10 points] Implement a procedure, `(tally-file pred? filename)`, that takes a unary predicate and the name of a file and counts the number of values in the file for which the predicate holds. For example, consider the following file, which might be named "stuff".

```
5 year old children say
(alphabet soup)
"tastes good"
(when it is not)
()
```

Various predicates will give different counts for that file

```

> (tally-file null? "stuff")
1
> (tally-file symbol? "stuff")
4
> (tally-file list? "stuff")
3
> (tally-file string? "stuff")
1
> (tally-file boolean? "stuff")
0

```

c. [5 points] Implement a procedure, `(tally-tree pred? tree)`, that takes a unary predicate and a tree and counts the number of leaves in the tree for which the predicate holds. For example,

```

> (tally-tree exact? (cons (cons 3+4i 3.0) (cons (cons 22/7 (cons 4.2 5)) 8.3)))
3
> (tally-tree complex? (cons (cons 3+4i 3.0) (cons (cons 22/7 (cons 4.2 5)) 8.3)))
6
> (tally-tree integer? (cons (cons 3+4i 3.0) (cons (cons 22/7 (cons 4.2 2)) 8.3)))
2
> (tally-tree integer? 5)
1
> (tally-tree integer? 'a)
0
> (tally-tree null? (cons (cons 3+4i 3.0) (cons (cons 22/7 (cons 4.2 2)) 8.3)))
0
> (tally-tree null? (list (list 3+4i 3.0) (list (list 22/7 (list 4.2 2))8.3)))
5

```

Problem 2: Fun with Higher-Order Procedures

Subject: Higher-order procedures, predicates

In our explorations of higher-order procedures, we have encountered a variety of interesting procedures. Here are some you might find useful, as well as some other standard procedures from the experienced Scheme programmer's toolbox.

```

(define l-s
  (lambda (binproc left)
    (lambda (right)
      (binproc left right))))

(define r-s
  (lambda (binproc right)
    (lambda (left)
      (binproc left right))))

(define o
  (lambda (f g)
    (lambda (x)
      (f (g x)))))

(define both
  (lambda (pred1? pred2?)
    (lambda (val)

```

```

        (and (pred1? val) (pred2? val))))))

(define either
  (lambda (pred1? pred2?)
    (lambda (val)
      (or (pred1? val) (pred2? val)))))

(define negate
  (lambda (pred?)
    (lambda (val)
      (not (pred? val)))))

(define all
  (lambda (pred?)
    (lambda (lst)
      (let kernel ((lst lst))
        (or (null? lst)
            (and (pred? (car lst))
                 (kernel (cdr lst))))))))))

(define all-real? (all real?))

(define any
  (lambda (pred?)
    (lambda (lst)
      (let kernel ((lst lst))
        (and (not (null? lst))
             (or (pred? (car lst))
                 (kernel (cdr lst))))))))))

(define any-odd? (any (both integer? odd?)))

(define member?
  (lambda (val lst)
    ((any (l-s equal? val)) lst)))

(define select
  (lambda (pred? lst)
    (cond
      ((null? lst) null)
      ((pred? (car lst))
       (cons (car lst) (select pred? (cdr lst))))
      (else (select pred? (cdr lst)))))

```

We use these procedures to more concisely and more elegantly define procedures, as in the case of `any-odd?` above. Your goal is to write the following definitions as concisely as possible, using the various procedures above.

a. [10 points] Write a procedure, `(select-Bs lst)`, that, given a list of real numbers, selects only those that are at least 80 and strictly less than 90.

b. [10 points] Write a procedure, `(make-filter pred?)`, that returns a filter. The filter is a procedure of one parameter (a list, `lst`) which returns a new list by removing all the elements of the `lst` for which `pred?` holds. For example,

```

> (define no-symbols (make-filter symbol?))
> (no-symbols (list 1 2 'a 'b 3 5))
(1 2 3 5)
> (no-symbols null)
()

```

c. [5 points] As you may recall, in an earlier assignment you wrote a procedure, `(intersect lst1 lst2)` in which `lst1` and `lst2` are lists of symbols (with no duplicates), representing sets. Now that you have the higher-order toolbox, you can write the solution much more concisely. Do so. You may assume that `lst1` and `lst2` are proper sets: That is, neither of the lists contains duplicate elements.

Hint! You may find `member?`, `select` and the sections useful.

Problem 3: All Sorts of Sorting

Topics: Sorting, Anonymous Procedures, Comparators

In our exploration of sorting, we discovered that the choice of the comparator used to determine whether one value naturally preceded another value was of key import in writing calls to sorting routines. Here are some potentially interesting problems. For each, use merge sort as the mechanism for sorting, relying on `mergesort.scm`, the sample implementation of merge sort.

a. [10 points] Suppose `words` is a list of strings. Write an expression to sort `words` by the length of each word.

b. [10 points] Suppose we represent homework grades in the class with a list of lists. Each element list begins with a string that represents the student's name and is followed by all of the grades, represented as integers. For example,

```

(define grades
  (list
    (list "Sam" 60 60 90 100 60 80)
    (list "Sammy" 95 95 95 95 90 80)
    (list "Samwise" 100 0 0 0 0 0)
    (list "Sammer" 60 70 80 90 100 110)))

```

Write an expression that sorts `grades` by average grade.

c. [5 points] As you may know, some printers use lists of grid points to determine where to place ink. Suppose we represent points by pairs with the x coordinate (an integer) as the car and the y coordinate (an integer) as the cdr. To improve the performance of our printer, we want to group rows together (so that all the values with a y of 0 appear before all values with a y of 1 appear before all values with a y of 2, and so on and so forth). In addition, our printer scans rows in opposite directions. That is, it scans row 0 left to right, row 1 right to left, row 2 left to right, and so on and so forth. Write an expression to sort a list of points, `points` to meet the scan behavior of the printer.

Problem 4: Finding the Nth Smallest Element of a List

Topics: Divide-and-conquer, algorithm analysis, Quicksort

As you may recall, one of the reasons that we have to use a random pivot in Quicksort is that it is expensive (in terms of computing time) to find the median element of a list. In fact, for an arbitrary n , it is expensive to find the n th-smallest element of the list.

There is one obvious strategy:

- Find the smallest element and remove it.
- Find the next smallest element and remove it.
- Find the next smallest element and remove it.
- ...

After we remove n elements, the smallest remaining element is the n th smallest. For a list of length m , this takes approximately $n*m$ comparisons.

Can we do better? Yes. It turns out that a variant of Quicksort is also ideal for finding the n th element of the list.

- Pick a random pivot.
- Partition the list into elements less than the pivot, equal to the pivot, and greater than the pivot.
- If there are more than n elements less than the pivot, find the n th smallest of the small elements. (That is, recurse.)
- If there are n or more elements that are either less than or equal to the pivot, we can discard those elements. Find the k th smallest element of the large elements, where k is computed from n , acknowledging that we have now discarded the elements smaller than or equal to the pivot. (That is, you'll need to use a value other than n in the recursive call.)
- Otherwise, the pivot is the n th smallest element.

a. [10 points] Document (`nth-smallest n lst precedes?`), a procedure that find the n th smallest element of `lst`, using `precedes?` to determine ordering.

b. [15 points] Implement (`nth-smallest n lst precedes?`).

You may find the following procedures helpful.

```
;;; Procedure:
;;;   random-element
;;; Parameters:
;;;   lst, a nonempty list of values
;;; Purpose:
;;;   "Randomly" select an element of lst.
;;; Produces:
;;;   val, an element of list.
;;; Preconditions:
;;;   lst is nonempty.
;;; Postconditions:
;;;   val is a member of lst - i.e., (member? val lst)
```

```

;;; Each element of lst can be selected with equal probability.
(define random-element
  (lambda (lst)
    (list-ref lst (random (length lst)))))

;;; Procedure:
;;; partition
;;; Parameters:
;;; lst, a list
;;; pivot, a value
;;; precedes?, a binary predicate
;;; Purpose:
;;; Partition lst into three lists,
;;; one for which (precedes? val pivot) holds,
;;; one for which (precedes? pivot val) holds, and
;;; one for which neither holds.
;;; Produces:
;;; (smaller-elements equal-elements larger-elements), a three-element list
;;; Preconditions:
;;; precedes? can be applied to pivot and any value of lst.
;;; Postconditions:
;;; (append smaller-elements equal-elements larger-elements)
;;; is a permutation of lst.
;;; (precedes? (list-ref smaller-elements i) pivot)
;;; holds for every i, 0 < i < (length smaller-elements).
;;; (precedes? pivot (list-ref larger-elements j))
;;; holds for every j, 0 < j < (length larger-elements).
;;; Neither (precedes? (list-ref equal-elements k) pivot)
;;; nor (precedes? pivot (list-ref equal-elements k))
;;; holds for every k, 0 < k < (length equal-elements)
(define partition
  (lambda (lst pivot precedes?)
    (letrec ((kernel
              (lambda (remaining smaller-elements equal-elements larger-elements)
                (cond
                 ((null? remaining)
                  (list (reverse smaller-elements)
                        (reverse equal-elements)
                        (reverse larger-elements)))
                 ((precedes? (car remaining) pivot)
                  (kernel (cdr remaining)
                          (cons (car remaining) smaller-elements)
                          equal-elements
                          larger-elements))
                 ((precedes? pivot (car remaining))
                  (kernel (cdr remaining)
                          smaller-elements
                          equal-elements
                          (cons (car remaining) larger-elements)))
                 (else
                  (kernel (cdr remaining)
                          smaller-elements
                          (cons (car remaining) equal-elements)
                          larger-elements))))))
      (kernel lst null null null))))

```

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

`select-Bs`

How concise is your answer to `select-Bs`?

I have eleven “words” in my answer, counting `define` as one and `select-Bs` as another.

Any lambdas?

No.

Is a 90 a B?

No.

Is an 80 a B?

Yes.

`make-filter`

Can you really write `make-filter` without a lambda?

Yes.

Do we have to write `make-filter` without a lambda?

I’d prefer that you try to do so. If you can’t, see if you can do it with just one lambda.

How concise is your answer to `make-filter`?

I have seven “words” in my answer, counting `define` as one and `make-filter` as another.

Can you give me some hints on writing `make-filter` that concisely?

Sure. First write `make-selector`, which works like `select`, except it takes one parameter, a predicate, and returns a procedure that selects all values that meet the predicate. Next, rewrite that procedure as concisely as possible. (Hint: Look for something that looks like `(lambda (val) (f constant val))` and replace it with `(l-s f constant)`.) Next, write `make-filter` in terms of `make-selector`. Finally, replace `make-selector` with the expression you used to define it. If you do all of those things in that order, you should end up with something close to what I wrote.

`intersect`

How concise is your answer to `intersect`?

My solution has ten words, including `define`, `intersect`, `lambda`, `lst1`, and `lst2`.

Can you do without the lambda?

Not if I restrict myself to to procedures I’ve provided in the exam.

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.

- Sam can't count integers. [PR, 1 point]
- Sam put double quotes at the end of a symbol. [EJ, 1 point]
- Sam is archaic and uses the deprecated term "import". [EJ, 0 points]
- Sam can't count symbols. [TK, 1 point]
- random-element uses the wrong article for "element". [MH, 1 point]
- Sam can't spell "probability". [MH, 1 point]

Five points down, now we're on fractional points. Each five errors earns a point.

Extra Credit Point #6

- Sam can't count (in the "Produces" section of "partition"). [MH]
- null, when read from a file, is a symbol and not the empty list. [CW]
- Another counting error. [PR]
- Sam seems to spell "select" as "remove" [CW]

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.