

Exam 2: Structures

Distributed: Friday, 27 October 2006

Due: 9:00 a.m., Friday, 3 November 2006

No extensions.

This page may be found online at

<http://www.cs.grinnell.edu/~rebelsky/Courses/CS151/2006F/Exams/exam.02.html>.

This exam is also available in PDF format.

Contents

- Preliminaries
- Problems
 - Problem 1: Searching in Vectors
 - Problem 2: From Tree of Symbols to String
 - Problem 3: Roommate Matching
 - Problem 4: Separating A List
- Some Questions and Answers
 - Questions on Problem 1
 - Questions on Problem 2
 - Questions on Problem 3.a
 - Questions on Problem 3.b
 - Questions on Problem 4
 - Other Questions
- 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: Searching in Vectors

Topics: Vectors, Recursion, Searching, Unit Testing.

As you may recall, we have found it useful to write a `member?` procedure for lists. This procedure takes two parameters, a value and a list, and returns `true` (`#t`) if the value appears in the list and `false` (`#f`) otherwise. It is useful to write a similar procedure for vectors. However, since indices are important for vectors, the procedure should return the index of the value if it appears, and `#f` if the value does not appear. We'll call this procedure `index-of`.

For example,

```
> (index-of "Samuel" (vector "Samuel" "Alexander" "Rebelsky"))
0
> (index-of "Rebelsky" (vector "Samuel" "Alexander" "Rebelsky"))
2
> (index-of "SamR" (vector "Samuel" "Alexander" "Rebelsky"))
#f
> (index-of 13 (vector 2 3 5 7 11 13 17 23))
5
> (index-of 9 (vector 2 3 5 7 11 13 17 23))
#f
```

- a. [5 points] Document `index-of`.
- b. [10 points] Implement `index-of`. You may not use `vector->list` in your implementation.
- c. [10 points] Write a test suite for `index-of`, using the Unit Testing framework we studied.

Problem 2: From Tree of Symbols to String

Topics: Strings, Trees, Deep Recursion

In past semesters, students have asked me how Scheme prints lists. To explain, I've been known to write a short procedure that converts lists of symbols to strings (because lists of symbols provide a sufficiently complex and sufficiently simple example). Here's the procedure:

```

;;; Procedure:
;;; symbols->string
;;; Parameters:
;;; symbols, a list of symbols [unverified]
;;; Purpose:
;;; Converts the list to a string similar to one that might be
;;; printed for the list.
;;; Produces:
;;; symstr, a string
;;; Preconditions:
;;; (none)
;;; Postconditions:
;;; symstr is a value such that if you write symstr to a file using
;;; write, then you get the same result as if you'd written symbols
;;; to a file using display.
;;; Process:
;;; We treat the empty string as a special case, and just print
;;; the open and close parens.
;;; For all other cases, we concatenate an open paren, the first
;;; symbol, the remaining symbols, and a close paren.
;;; We use a recursive kernel to process the remaining symbols.
;;; Because we've printed one symbol already, we can precede each
;;; remaining symbol with a space.
(define symbols->string
  (letrec ((kernel (lambda (symbols)
                     (cond
                      ((null? symbols) "")
                      (else (string-append " "
                                             (symbol->string (car symbols))
                                             (kernel (cdr symbols)))))))
    (lambda (symbols)
      (cond
       ((null? symbols) "()")
       (else (string-append "("
                             (symbol->string (car symbols))
                             (kernel (cdr symbols))
                             ")"))))))

```

a. [10 points] As you've probably noted, this procedure works only with lists of symbols. However, it should be possible to make it work with things very much like lists, except that they end with a symbol, rather than null. Rewrite it so that it accepts such structures.

```

> (symbols->string (list 'a 'b 'c))
"(a b c)"
> (symbols->string null)
"()"
> (symbols->string (cons 'a 'b))
"(a . b)"
> (symbols->string (cons 'a (cons 'b 'c)))
"(a b . c)"

```

Hint: Add another base case in the kernel, one that checks if `symbols` is a single symbol.

b. [5 points] Some folks think it's useful to make procedures like `symbols->string` work with singleton values in addition to pair structures. Extend `symbols->string` so that it accepts a single symbol as a parameter and returns a string for that symbol.

```
> (symbols->string 'a)
"a"
```

c. [10 points] Now that you've made `symbols->string` work with one form of pair structure and with singletons, you should be able to make it work with more general pair structures. In particular, it should work with trees of symbols and with lists of lists of symbols. Rewrite `symbols->string` so that it works with such values.

For example,

```
> (symbols->string (list 'a 'b 'c))
"(a b c)"
> (symbols->string null)
"()"
> (symbols->string (cons 'a 'b))
"(a . b)"
> (symbols->string (cons 'a (cons 'b 'c)))
"(a b . c)"
> (symbols->string (cons (cons 'a 'b) (cons 'c 'd)))
"((a . b) c . d)"
> (symbols->string (list (list 'a 'b) (list 'c 'd) (list 'e 'f)))
"((a b) (c d) (e f))"
> (symbols->string (list null (list null)))
"(( ) ( ))"
> (symbols->string (cons 'a (cons (cons (cons 'b 'c) 'd) 'e)))
"(a ((b . c) . d) . e)"
> (symbols->string 'a)
"a"
```

Hint: Consider the points in which the procedure calls `symbol->string` on the car of a pair. Since you no longer know that the car is a symbol, you'll need to do something else, since the car may be a symbol, a null, or a pair. Your primary task is to figure out what you should do. (To figure out what you might do, reflect on the reading on deep recursion.)

Problem 3: Roommate Matching

Topics: Files, Random Generation, Matching

Egelloc College wants to write a simple roommate matching procedure. For each student, we have last name, first name, year (1, 2, 3, or 4), smoker?, bedtime (early/late), political views (liberal/conservative), and favorite color. They have decided to store the values in a vector and have created a procedure that constructs such vectors.

```
(define student-info
  (lambda (last-name first-name year smokes? bedtime politics color)
    (cond
      ((not (string? last-name))
       (error "student-info: first parameter (last-name) must be a string"))
      ((not (string? first-name))
```

```

(error "student-info: second parameter (first-name) must be a string"))
((or (not (integer? year)) (< year 1) (> year 4))
(error "student-info: third parameter (year) must be an integer between 1 and 4"))
((not (boolean? smokes?))
(error "student-info: fourth parameter (smokes?) must be a boolean"))
((not (member bedtime (list 'early 'late)))
(error "student-info: fifth parameter (bedtime) must be 'early or 'late"))
((not (member politics (list 'liberal 'conservative)))
(error "student-info: sixth parameter (politics) must be 'liberal or 'conservative"))
((not (string? color))
(error "student-info: seventh parameter (color) must be a string"))
(else
(vector last-name first-name year smokes? bedtime politics color))))

```

They have also provided accessors for all of the fields.

```

(define get-last-name
  (lambda (student)
    (vector-ref student 0)))
(define get-first-name
  (lambda (student)
    (vector-ref student 1)))
(define get-year
  (lambda (student)
    (vector-ref student 2)))
(define smokes?
  (lambda (student)
    (vector-ref student 3)))
(define get-bedtime
  (lambda (student)
    (vector-ref student 4)))
(define get-politics
  (lambda (student)
    (vector-ref student 5)))
(define get-favorite-color
  (lambda (student)
    (vector-ref student 6)))

```

Here's a very short sample file of students (`sample-students.scm`):

```

#7("Smith" "Joe" 1 #f early conservative "blue")
#7("Jones" "Jane" 2 #t late liberal "yellow")
#7("Emilyson" "Jacob" 2 #f late liberal "purple")
#7("Smithson" "Jack" 1 #t late liberal "purple")
#7("Sonsmith" "Jenna" 2 #f early liberal "yellow")

```

a. [15 points] Write and test a procedure, `suggest-roommate`, that takes two parameters -- *student*, a student vector, and *file-of-students*, the name of a file containing a list of students, and (1) makes a list of all the good matches for student from the file of students, (2) randomly selects one of those matches, and (3) reports the name of that student in the form "Last-name, First-name".

A good match is one in which the years match and at least three out of the remaining four values match.

If there is no good match, have your procedure return #f.

For example, `suggest-roommate`, given a 2nd-year, smoking, early-to-bed, liberal, lover of the color yellow and the list above must return either "Jones, Jane" or "Sonsmith, Jenna", since each of those students are second year students, and Jane matches on smoking, politics, and color while Jenna matches on bedtime, politics, and color. Jacob Emilyson, the only other second year, matches only on politics, and so should not be considered.

```
> (suggest-roommate (student-info "Doe" "Dee" 2 #t 'early 'liberal "yellow") "sample-students.scm")
"Jones, Jane"
```

b. [10 points] Of course, Egelloc would never release information about its students to a contract programmer, so you will need to generate simulated data to test your procedure. (You can use the sample above, but it's pretty short, so you will still need to solve this part of the problem eventually.)

Write a procedure, `(generate-random-students filename n)` that generates n random student entries and shoves them in the file with the given name.

Problem 4: Separating A List

Topics: Husk/Kernel programming, List recursion, Local bindings

In grading, I often end up with a list of name/grade pairs for the students in my class. It would be convenient if I could separate that big list into four lists: students with A's, B's, C's, and D's. Assume that every pair contains a string as its car and an integer as its cdr.

a. [15 points] Write a procedure, `process-grades`, that takes a list of name/grade pairs as parameters and produces a list of four lists,

- The first list contains all students that deserve A's (have a grade of 90 or higher).
- The second list contains all students that deserve B's (have a grade between 80 and 89, inclusive).
- The third lists contains all students that deserve C's (have a grade between 70 and 79, inclusive).
- The final list contains all students that deserve D's (have a grade below 70).

The order of the names within the individual lists does not matter.

Here's an example of `process-grades` in action:

```
> (process-grades (list (cons "Sam" 90)
                        (cons "Eryn" 85)
                        (cons "Emily" 85)
                        (cons "Joe" 60)
                        (cons "Jane" 65)
                        (cons "Janet" 100)))
(("Sam" "Janet") ("Eryn" "Emily") () ("Joe" "Jane"))
```

You need not test preconditions for part a of the problem.

b. [10 points] Of course, it is useful to test preconditions. Hence for this part of the problem, you must wrap your answer from the previous step in a husk/kernel for precondition testing (in particular, that the list is of the appropriate form and that all the grades are in the range 0-100). You must make the kernel local.

You should report errors using the following protocol:

- If the parameter is not a list, report *grades: expects a list as a parameter*
- If the parameter contains an entry that is not of the form (cons string number), report *grades: expects each list element to be a string/number pair.*
- If the parameter contains an entry with a grade below 0 or over 100, report *grades: all grades must be in the range [0..100]. However, **name** has a grade of **grade**.*

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

My `index-of` works fine if the values are all symbols, but not if they're strings. Why?

I'd guess that you're using `eq?` rather than `equal?` to compare values. Recall that identical strings may occupy different areas of memory, so `eq?` does not suffice from comparing strings.

Should we support both lists and vectors as the second parameter of `index-of`?

No, you only need to support vectors.

Can you remind me of the structure of procedures that recurse over vectors?

Sure. Here's one that only extracts values from the vector. (You'd do something different if you were modifying the the vector.)

```
(define recursive-vector-proc
  (letrec ((kernel (lambda (pos vec len)
                     (if (= pos len)
                         (BASE-CASE)
                         (COMBINE (vector-ref vec pos)
                                  (kernel (+ pos 1) vec len))))))
    (lambda (vec)
      (kernel 0 vec (vector-length vec)))))
```

Do we have to document kernel and husk?

If the kernel is local, you need not document it.

How many P's?

The basic six P's.

Will we get extra credit for using more than six P's?

It depends on (a) the correctness of the extra documentation and (b) my mood. Note that incorrect extra documentation will incur a penalty.

Do we have to record *all* instances the value is in the vector, or just the first time it appears?

You need only report *one of the indices* if it appears multiple times. The first index is fine, but the last is also okay, as is any other one. Just make sure to document what you’ve done.

Questions on Problem 2

For part c, should we expect to do a significant rewrite of our answer for part b, or is it simply a case of making a few key changes?

You should be able to make only a few changes. As the problem suggests, you’ll need to change each call to `(symbol->string (car symbols))` to something else.

Questions on Problem 3.a

Was there an error in the sample file, wherein the symbols were quoted?

Um, yes. It is now fixed.

Can you remind me of the structure of a procedure that recurses over an input file, particularly since we learned about local variables and local procedures after we did input and output?

Sure. Here’s the general structure:

```
(define recursive-read-file-proc
  (letrec ((kernel (lambda (input-port)
                     (let ((val (read input-port)))
                       (cond
                        ((eof-object? val)
                         (close-input-port input-port)
                         (BASE-CASE))
                        (else
                         (COMBINE val (kernel input-port)))))))
    (lambda (fname)
      (kernel (open-input-file fname)))))
```

Can you give me some hints as to how to get started?

You might find it useful to define a separate `(good-match? student1 student2)` procedure. You might find it even more useful to make the first version of that procedure return true if the two students are the same year and false otherwise. (That is, to pay attention only to year.) Once you’ve defined this helper, make `suggest-roommate` extract just the good matches from the file.

Okay, I can now select all the “good” matches, randomly select one of them (particularly since you just taught us how), and extract the first and last name. Now I’m back to writing the `good-match?` helper. I’m trying to figure out how to make sure that at least three attributes match. Any suggestions?

One strategy is to write four additional helpers (or simply local variables), each of which evaluates one of the four attributes and returns (takes the value) 0 if the attributes don’t match and 1 if the attributes do match. I’ll leave it to your imagination how to use those helpers.

Questions on Problem 3.b

Can we select the first name randomly from a list, or do we need to do something more sophisticated?

Selecting randomly from a list is fine.

If we do something more sophisticated, will we receive extra credit?

If you do something more sophisticated and it's wrong, you will receive negative extra credit. If you do something more sophisticated and it's right, you may or may not receive extra credit, depending on what you've done and the mood I'm in.

I've written a `random-student` helper, and I want to test it. Can you show me a series of commands I'd use to write, say, 3 random students to a file?

Here goes ...

```
(define rs (open-output-file "random-students"))
(write (random-student) rs)
(newline rs)
(write (random-student) rs)
(newline rs)
(write (random-student) rs)
(newline rs)
(close-output-port rs)
```

Would it help to have some named students?

Yes. Here are some.

```
(define joe-smith #7("Smith" "Joe" 1 #f early conservative "blue"))
(define jane-jones #7("Jones" "Jane" 2 #t late liberal "yellow"))
(define dee-doe (student-info "Doe" "Dee" 2 #t 'early 'liberal "yellow"))
```

Questions on Problem 4

For this problem, I wrote four separate helpers. One extracts the A names, one extracts the B names, one extracts the C names, and one extracts the D names. What do you think of this strategy?

Not much. You traverse the list four times. I'd really prefer that you traverse the list only once.

Given that you don't like this nice, clean, straightforward strategy, how much are you going to take off if I use it anyway?

Not much. Let's see ... the problem is worth 15 points. That's a B-level solution. Hence, I'll take off 3 points.

Do you have a recommendation for how I should check whether *grade* is in the range 80..89, inclusive?

```
(<= 80 grade 89)
```

Other Questions

Why do you penalize us for doing incorrect extra work? It seems that such penalties will discourage us for trying extra things?

First, I want you to focus on doing the problems correctly, not on "what else can I do?" Second, I have encountered students in the past who simply throw in anything they can think of with the hope that doing so will earn them a bit of credit. I certainly want to discourage such activities.

How about if I add something Trogdor-related to the exam?

If it's pertinent, that may give you some points.

If someone in the other class catches an error in their version of the exam, and the same error appears in our version, do we get credit for it?

Not if I fix it before someone in this class tells me.

Are you concerned about the number of questions and answers that appear here?

No. A few questions are “resolve ambiguities” and, while a lot of those would concern me, a few aren’t bad. From my perspective, most of the questions are a variant of “Assess what I’ve done” or “Can you give me a hint?” I’ve chosen how much help to give in those cases and, instead of limiting the hint to one student, I’ve shared it with the class.

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 forgot to remove “*This exam is currently in draft format!*” from the released version. [SR, 1 point]
- Sam doesn’t know the difference between lists and vectors. [MP and TP, 2 points]
- purplese is not a color. [MP, 1 point]
- The symbols in the sample file should not be quoted [TP, 2 points].

Additional errors.

- Sam wrote `dr` instead of `cdr` in Problem 4. [SB]
- Sam wrote “Jenna matches on bedtime politics” rather than “Jenna matches on bedtime, politics”, giving a whole different meaning to the attribute. [TM]

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.