

Exam 1: Scheme Fundamentals

Distributed: Friday, February 16, 2007

Due: 10:00 a.m., Friday, February 23, 2007

No extensions.

This page may be found online at

<http://www.cs.grinnell.edu/~rebelsky/Courses/CS151/2007S/Exams/exam.01.html>.

This exam is also available in PDF format.

Contents

- Preliminaries
- Problems
 - Problem 1: Describing values
 - Problem 2: Mystery Predicates
 - Problem 3: Building Lists
 - Problem 4: Selecting Courses
- 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.

Unless I explicitly ask you to document your procedures, you need not write introductory comments.

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: Describing values

Topics: Types, Predicates, Strings

We've now encountered a wide variety of basic types in our exploration of Scheme, including integers (exact and inexact), real numbers, lists, strings, and even procedures. We've used predicates such as `list?` that let us ask whether a value is of a particular type. However, Scheme provides no obvious mechanism for reporting the type of an arbitrary value.

Write a procedure, (`describe val`), that returns a string describing the type of `val`. If `val` is not of a type we've learned about, report that `val` is "a value of unknown type". For example,

```
> (describe 1)
"an exact integer"
> (describe 3.4)
"an inexact real number"
> (describe (list 1 2 3))
"a list"
> (describe null)
"an empty list"
> (describe null?)
"a procedure"
> (describe (cons 1 2))
"a value of unknown type"
```

As you might expect, one of the goals of this question is for you to figure out what types you know about.

Note that our version of Scheme does not distinguish real numbers from rational numbers, so you need not do so either. I generally make it a practice to refer to exact non-integral real/rational numbers as rational and to inexact non-integral real/rational numbers as real.

Problem 2: Mystery Predicates

Topics: Conditionals, Booleans, Recursion, Style, Predicates

You may recall that I've written that I don't like to see if statements that explicitly return #t or #f since you can easily rewrite them using some combination of and, or, and not. For example,

- `(if test1 #t #f) => test1`
- `(if test1 #f #t) => (not test1)`
- `(if test1 #t test2) => (or test1 test2)`
- `(if test1 test2 #f) => (and test1 test2)`
- `(if test1 test2 test3) => (or (and test1 test2) (and (not test1) test3))`

For each of the following predicate procedures:

- Rewrite the procedure so it uses and, or, and not rather than if, #t, and #f.
- Figure out what the procedure does and then give the procedure a more meaningful name.

(a) (5 points)

```
(define mystery?  
  (lambda (val)  
    (if (symbol? val)  
        #t  
        (if (string? val)  
            #t  
            #f))))
```

(b) (5 points)

```
(define puzzle?  
  (lambda (a b c)  
    (if (< a b)  
        (if (< b c)  
            #t  
            #f)  
        (if (< b c)  
            #f  
            #t))))
```

(c) (5 points)

```
(define conundrum?  
  (lambda (lst)  
    (if (list? lst)  
        (if (null? lst)  
            #f  
            (null? (cdr lst)))  
        #f)))
```

(d) (10 points) Note that this procedure is recursive.

```
(define enigma?
  (lambda (lst)
    (if (null? lst)
        #t
        (if (symbol? (car lst))
            (enigma? (cdr lst))
            #f))))
```

Problem 3: Building Lists

Topics: Lists, List construction procedures, List recursion

Many of you have been a bit frustrated by the problem of choosing whether to use `list`, `cons`, or `append` when creating lists. Rather than having to remember the details each time, we might instead write a procedure that combines the capabilities of the three procedures, and perhaps even adds other capabilities.

Define a procedure, (`join val1 val2`), that follows the following four guidelines (labeled a-d)

a. If `join` is presented with two lists, it appends them using `append`. For example,

```
> (join (list 1 2 3) (list 'a 'b))
(1 2 3 a b)
> (join null (list "left" "right"))
("left" "right")
> (join (list #\a #\b #\c) null)
(#\a #\b #\c)
```

b. If `join` is presented with a simple value and a list as parameters (in that order), it prepends the value to the list. For example,

```
> (join #t (list #f #f #f))
(#t #f #f #f)
> (join "silly" null)
("silly")
```

c. If `join` is presented with a list and a simple value as parameters (in that order), it puts the value at the end of the list. For example,

```
> (join (list "Grinnell" "has" "no") "limits")
("Grinnell" "has" "no" "limits")
> (join (list 1 2 3 4) 5)
(1 2 3 4 5)
> (join null 'llun)
(llun)
```

d. If `join` is presented with two simple values as parameters, it puts them into a list.

```
> (join 'a 'b)
(a b)
> (join #t #f)
(#t #f)
```

Note that you should define a single `join` that has all four behaviors. If you find it helpful to write helper procedures (such as a procedure that puts a value at the end of a list), you may certainly do so.

The preceding refers to “simple” values. For the purposes of this problem, we’ll define a value as “simple” if it is not a list. That is,

```
(define simple?
  (lambda (val)
    (not (list? val))))
```

Problem 4: Selecting Courses

Topics: Recursion, Symbols, Membership

One reason we write programs is to help us manipulate and analyze large (or even medium-sized) collections of information. Suppose we want to create a system that students can use to help select courses. For such a system, we’ll need a way to represent information about courses and we need to design useful procedures to manipulate that information.

A complete course system is clearly beyond the scope of an exam, so let’s try a simple version of the problem.

We will represent each course at Grinnell as a list whose first element is a string that names the course and whose remaining elements are symbols that represent terms students have used to describe the class. For example, we might represent this course as

```
("CSC151" geeky challenging morning creative)
```

We can then represent a collection of courses as a list of these lists. (Note that failure to associate a term with a class does not suggest that the class does not have that characteristic; simply that it was not a term that first came to mind when our subject was asked to describe the class.)

```
(define some-courses
  (list
    (list "CSC151" 'geeky 'challenging 'morning 'creative 'numeric 'technology)
    (list "HUM101" 'challenging 'classic 'writing)
    (list "BIO150" 'workshop 'cool 'science)
    (list "MAT133" 'morning 'afternoon 'numeric 'large)
    (list "TUT100" 'required 'awesome 'morning 'challenging 'writing)
    (list "PHY131" 'workshop 'science)
    (list "MUS219" 'creative 'technology)
    (list "PHI111" 'morning 'challenging 'essential 'humanistic)
    (list "PHY457" 'quantum 'easy 'science)
    (list "ART136" 'creative)))
```

Write a procedure, (`find-courses category list-of-courses`), that finds the names of all courses that meet a particular characteristic. For example,

```
> (find-courses 'challenging some-courses)
("CSC151" "HUM101" "TUT100" "PHI111")
> (find-courses 'easy some-courses)
("PHY457")
> (find-courses 'limitless some-courses)
()
> (find-courses 'workshop some-courses)
("BIO150" "PHY131")
```

You may find it helpful to use the `member?` procedure you might have written for a recent lab. That procedure can be defined as follows:

```
(define member?
  (lambda (sym lst)
    (and (not (null? lst))
         (or (eq? sym (car lst))
             (member? sym (cdr lst))))))
```

You may also find it helpful to break this problem into parts. First, write a procedure that, given a characteristic and list of courses, returns a list of all courses (as lists) that contain the characteristic. Next, write a procedure that, given a list of courses, returns a list of the names of those courses. Finally, write `find-courses` by combining the two.

Some Questions and Answers

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

Problem 1

How specific do I have to be in describing characters? For example, do I need to distinguish digit characters?

You need not indicate what kind of character it is. The result "a character" suffices for all characters.

If the value is a number, do we have to make scheme figure out exactly what kind of number it is? i.e. we know the following procedures so far: `exact?`, `inexact?`, `positive?`, `negative?`, `zero?`, `even?`, `odd?`, `rational?`, `complex?`, and `real?`. Are there any of these we don't need on the exam?

You need to indicate the primary type (integer, real/rational, or complex) and its exactness. The remaining are not necessary.

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.

- Sam wrote “PHI457” when he meant “PHY457”. [KI, 1 point]
- Because Sam wrote problem 4 at midnight, he forgot to include the second parameter in some calls to `find-courses`. [AM, 1 point]
- In problem 4, “the a list” should be “a list”. [JS, 1 point]
- In problem 4, PHI131 got misnumbered as PHI111. [KI, 1 point]

- In problem 2, it is not the case that
`(if test1 test2 test3) => (or (and test1 test2) test3)`
rather
`(if test1 test2 test3) => (or (and test1 test2) (and (not test1) test3))` [SR, 1 point]

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.