

Exam 1: Scheme Fundamentals

Distributed: Friday, 22 September 2006

Due: 9:00 a.m., Friday, 29 September 2006

No extensions.

This page may be found online at

<http://www.cs.grinnell.edu/~rebelsky/Courses/CS151/2006F/Exams/exam.01.html>.

This exam is also available in PDF format.

Contents

- Preliminaries
- Problems
 - Problem 1: Determining Types
 - Problem 2: Simulating `if` and `cond` with `or` and `and`
 - Problem 3: Analyzing `Append`
 - Problem 4: Other List Operations
- Some Questions and Answers
- Errors
- Other Extra Credit

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: Determining Types

Topics: Types, Predicates

Your colleagues, Tyra and Tyler Typer, are thrilled that we've encountered a wide variety of basic types in our exploration of Scheme, including integers (exact and inexact), complex numbers, characters, strings, input-ports, and even procedures. However, they are concerned that Scheme provides no obvious mechanism for determining the type of a value.

They've turned to me to write such a procedure, and I've decided to delegate the work to you. Write a procedure, `(type-of val)` that returns the type of value `val` as a symbol. If the value has a type that you don't know, report a type of "unknown". For example,

```
> (type-of 1)
exact-integer
> (type-of 3.4)
inexact-real
> (type-of null)
list
> (type-of list?)
procedure
> (type-of (cons 1 2))
unknown
```

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

Problem 2: Simulating `if` and `cond` with `or` and `and`

Topics: Booleans, Conditionals

In the reading on conditionals, we noted that it is possible to simulate the behavior of `and` and `or` using `if` and `cond`. Bob and Bonnie Boole (no relation to George Boole) think the converse should be possible. That is, they think that you should be able to rewrite `if` and `cond` clauses using just `or` and `and`. Unfortunately, they are better philosophers than coders. Hence, they have turned to you to work out the details.

a. Consider the following typical form of `if`

```
(if test
    consequent
    alternate)
```

Write an equivalent piece of code using `and`, `not`, and `or`.

b. Consider the following prototypical `cond` expression.

```
(cond
  (test0 consequent0)
  (test1 consequent1)
  (test2 consequent2)
  (else alternate))
```

Write an equivalent piece of code using `and`, `not`, and `or`.

Problem 3: Analyzing Append

Topics: Lists, Recursion, Efficiency

My computer science teachers, I. H. Ate and A. P. Pend, regularly discouraged me from using `append` in my list-building procedures, particularly my recursive list-building procedures. They never told me why, but I finally figured it out through experimentation. Following the philosophy that you learn better by doing than by being told, I challenge you to replicate my experiments (with some guidance).

We begin by defining our own version of `cons` that lets us figure out how often it gets called in a typical procedure.

```
(define my-cons
  (lambda (head tail)
    (display 'cons)
    (newline)
    (cons head tail)))
```

This definition by itself is not very interesting. Hence, we will use it in a prototypical definition of `append`.

```
(define my-append
  (lambda (first second)
    (if (null? first)
        second
        (my-cons (car first) (my-append (cdr first) second)))))
```

We can now see their interaction.

```
> (my-append (list 'a 'b 'c) (list 'd 'e 'f))
cons
cons
cons
(a b c d e f)
```

a. If you use `my-append` to join a list of length n to a list of length m , how many times does `my-cons` get called?

Now, let's write a `reverse` procedure that uses `my-append` (the thing Professors Ate and Pend so disliked). Since we need to build a singleton list for appending, we also write `singleton`.

```
(define singleton
  (lambda (sym)
    (my-cons sym null)))

(define my-reverse
  (lambda (lst)
    (if (null? lst)
        null
        (my-append (my-reverse (cdr lst)) (singleton (car lst))))))
```

Now, let's try it

```
> (my-reverse (list 'a 'b 'c))
cons
cons
cons
cons
cons
cons
(c b a)
```

Yes, the result is correct. We also call `cons` six times to make the result. Does that make sense? Well, there are three calls to create the singleton lists, and then we have to join them together, so I guess that's okay. But let's be safe and check a little more. What happens if we try to reverse a list of length four?

```
> (my-reverse (list 'a 'b 'c 'd))
cons
cons
cons
cons
cons
cons
cons
cons
cons
cons
(d c b a)
```

Hmmm ... that's a bit strange. The result is correct and has four elements, but it took ten calls to `cons`. I guess we should explore a bit more.

```
> (my-reverse (list 'a 'b 'c 'd 'e))
cons
... thirteen outputs elided
cons
(e d c b a)
```

Wow! Fifteen calls to cons for a list of size five. What is going on?

b. In your own words, explain why it is taking so many more calls to cons in the larger lists. You may find it helpful to try larger lists.

After observing these problems, I rewrote `reverse` to use a helper that accumulates the reverse as we go.

```
(define my-reverse
  (lambda (lst)
    (my-reverse-helper null lst)))
(define my-reverse-helper
  (lambda (reversed-so-far remaining)
    (if (null? remaining)
        reversed-so-far
        (my-reverse-helper
         (my-append (singleton (car remaining)) reversed-so-far)
         (cdr remaining)))))
```

Is this any better? Let's see ... For a list of size three, it does six calls to cons. About the same. For a list of size four, it does eight calls to cons. Better. For a list of size five, it does ten calls to cons. Still better.

c. In your own words, explain why this new version is so much better.

Of course, it seems a bit silly to make a one-element list if we're going to immediately append it to another list. We should instead use `cons` (or, for testing, `my-cons`).

```
(define my-reverse
  (lambda (lst)
    (my-reverse-helper null lst)))
(define my-reverse-helper
  (lambda (reversed-so-far remaining)
    (if (null? remaining)
        reversed-so-far
        (my-reverse-helper
         (my-cons (car remaining) reversed-so-far)
         (cdr remaining)))))
```

d. Explain the benefits of this improvement.

Problem 4: Other List Operations

Topics: List operations, Recursion

Lisa and Lisle Lister note that in the problem above, and in homework 6, we were able to define our own versions of built-in list operations. When they asked me about this, I told them that most of the built-in list operations can be defined in terms of only a few, more primitive operations (`cons`, `car`, `cdr`, `null`, and `null?`). That comment inspired this problem.

a. Define `(my-cadr lst)`, which extracts the second element of a list, using only the five primitive operations above. (You need not use all five.)

b. Define `(my-caddr lst)`, which extracts all but the first two elements of a list, using only the five primitive operations above. (You need not use all five.)

c. Define `(my-length lst)`, which determines the length of `lst`, using the five primitives above, conditionals, and any numeric operations you deem appropriate. Note that `my-length` probably needs to be defined recursively.

d. Define `(my-list-ref index lst)`, which extracts the element of `lst` that is preceded by `index` elements. Again, you may use the five primitives above, conditionals, and any numeric operations you deem appropriate. Note that `my-list-ref` probably needs to be defined recursively.

If you'd prefer to define this as `(my-list-ref lst index)`, thereby matching the order of `list-ref`, that's fine, too.

Some Questions and Answers

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

Problem 1

Can we use `integer?` and `real?` and other predicates, or do we have to define them ourselves?

It is quite difficult to define these predicates yourself, so you should use the built-in ones.

Do we have to check for odd, even, and negative numbers?

No.

Will we get extra credit if we do?

If you consider negative values “extra”, then yes.

While doing problem 1 on the exam, I recalled that Scheme uses “real” and “rational” almost interchangeably. Would you prefer us using one or the other when writing type-of, or is it just a matter of personal preference?

I prefer “rational” for exact numbers and “real” for inexact, but it is up to you.

Problem 2

Do we have to use all three of `and`, `or`, and `not`, or only those that we deem necessary?

Only those you consider necessary.

Do you have to write working code, or just something equivalent?

Just something equivalent.

Can I use the `eq?` procedure?

No.

Problem 3

No questions received yet.

Problem 4

You specify for parts 3 and 4 that conditionals are ok. Does that preclude us from using `if` on parts 1 and 2, then?

You may use `if` on parts 1 and 2, but you should be able to do without it. (Note that you do not have to check preconditions for either procedure.)

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 “and `if` or `cond` clause” when he probably meant “an” .[MP, 1 point]
- “You must present you exam to me” should be “your exam”. [TP, 1 point]
- In transcribing a question, Sam forgot the symbol that indicates that the question is, indeed, a question. (That is, he forgot the question mark.) [PR, 1 point]
- Sam reversed the parameters to `index-of`. [AB, 1 point]

Other Extra Credit

I challenged you to find a reasonable test that `tri.2211.ss` failed to pass. A few people tried, but as one wrote “To be honest, the tests that I didn’t get to work were limitations of Scheme rather than the program not working.”

For those people’s valiant efforts, everyone gets two points of extra credit.

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.