

Mid-Semester Examination

Distributed: Monday, March 5, 2007

Due: 9:00 a.m., Friday, March 16, 2007

No extensions.

This page may be found online at

<http://www.cs.grinnell.edu/~rebelsky/Courses/CS302/2007S//Exams/midsem.html>.

This exam is also available in PDF format.

Contents

- Preliminaries
- Problems
 - Problem 1: The History of Conditionals [20 points]
 - Problem 2: Higher-Order Programming [20 points]
 - Problem 3: Implementing L
 - Problem 3A. Scope in `define` [10 points]
 - Problem 3B. Improving Symbolic Comparisons [10 points]
 - Problem 3C: `cond` Expressions [10 points]
 - Problem 3D: `lambda` Expressions [10 points]
 - Problem 4: Continuations and Coroutines [20 points]
- Citations
- Some Questions and Answers
- Errors

Preliminaries

There are four problems on the exam. Some problems have subproblems. Not all problems are worth the same amount of 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.

Experience shows that different people find different problems complex. Hence, if you get stuck on an early problem, try moving on to another problem and let your subconscious work on the early problem. (You'll also get a better sense of accomplishment if you can find at least one problem that you can solve early.)

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 eight hours, depending on how well you've learned topics and how fast you work. *You should not work more than twelve hours on this exam. Stop at twelve hours and write "There's more to life than CS" and you will earn at least 70 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 six 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: The History of Conditionals [20 points]

Topics: History, Design, Conditionals

In “Revenge of the Nerds” Paul Graham (2002) tells us that

When it was first developed, Lisp embodied nine new ideas. [...] 1. Conditionals. A conditional is an if-then-else construct. We take these for granted now, but Fortran I didn't have them. It had only a conditional goto closely based on the underlying machine instruction.

- a. Find out what the original Fortran I conditional looked like and describe it.
- b. Find out what the Algol-60 or Algol-58 conditional looked like and describe it.
- c. In your own words, explain why the Lisp conditional is superior to these two conditionals. (If you don't believe that it's superior, you can explain that, too.)
- d. Explain how you think Hoare would react to the Lisp conditional, using the criteria of *Hints on Programming Language Design* (1973).

Problem 2: Higher-Order Programming [20 points]

Topics: Higher-order Programming

Here is a small library of higher-order procedures (purposefully undocumented).

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

```

(define both
  (lambda (p1? p2?)
    (lambda (val)
      (and (p1? val) (p2? val)))))

(define either
  (lambda (p1? p2?)
    (lambda (val)
      (or (p1? val) (p2? val)))))

(define choose
  (lambda (t c a)
    (lambda (x)
      (if (t x) (c x) (a x)))))

(define constant
  (lambda (c)
    (lambda (x)
      c)))

(define id
  (lambda (x)
    x))

(define l-s
  (lambda (f l)
    (lambda (r)
      (f l r))))

(define r-s
  (lambda (f r)
    (lambda (l)
      (f l r))))

(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))

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

(define curry2
  (lambda (f)
    (lambda (l)
      (lambda (r)
        (f l r)))))

```

In the following three subproblems, you can use those procedures, as well as any of the standard Scheme procedures. You may not, however, use `lambda`.

a. Without using `lambda` or the `curry2` provided above, define a procedure (`curry2 f`) that has the same effect as the following definition.

```
(define curry2
  (lambda (f)
    (lambda (l)
      (lambda (r)
        (f l r))))))
```

b. Without using `lambda`, define a procedure, (`no-inexact-odds lst`) that removes all the inexact odd numbers from `lst`. For example,

```
> (no-inexact-odds (list 2 3 5))
(2 3 5)
> (no-inexact-odds (list 2 3.0 5))
(2 5)
> (no-inexact-odds (list 2.0 3.0 5.2))
(2.0 5.2)
> (no-inexact-odds (list 2.0 3.0 7 1 5.2))
(2.0 7 1 5.2)
> (no-inexact-odds (list 'one 3.0 -1))
(one -1)
```

c. Without using `lambda`, define a procedure, (`selector pred?`), that, given a unary predicate as a parameter, returns a procedure that takes a list as a parameter and returns a list of the elements of that list for which `pred?` holds. For example,

```
> (define numbers (selector number?))
> (numbers null)
()
> (numbers (list 1 2 3))
(1 2 3)
> (numbers (list 'a 'b 1 2 'c))
(1 2)
> (numbers (list 'a 'b 'c 'd 'e))
()
```

Problem 3: Implementing L

Topics:: The implementation of Lisp, `eval`, Scope, C, Code reading.

In a recent afternoon, I put together the skeleton for a simple implementation of Lisp, which I've called L. You can find the code for L at

<http://www.cs.grinnell.edu/~rebelsky/Courses/CS302/2007S/Examples/L/>. L has a very simple evaluator. To evaluate an L expression (which looks remarkably like a Lisp expression, but with some limitations), you run `leval` with the expression as a parameter.

```
$ leval '(quote a)'
(quote . (a . nil))
=> a
$ leval '(car (quote (a b)))'
(car . ((quote . ((a . (b . nil)) . nil)) . nil))
=> a
```

I've put the single quotes there to make sure that the shell treats the expression as a single parameter. There is no single-quote shorthand in L.

The L interpreter will evaluate multiple expressions in a row.

```
$ leval '(car (quote (a b)))' '(cons 1 2)'
(car . ((quote . ((a . (b . nil)) . nil)) . nil))
=> a
(cons . (1 . (2 . nil)))
=> (1 . 2)
```

The L interpreter also permits a special form, `(define name val)`, which updates the symbol table.

```
$ leval '(define a (cons 0 1))' '(cons a a)'
(define . (a . ((cons . (0 . (1 . nil))) . nil)))
=> (0 . 1)
(cons . (a . (a . nil)))
=> ((0 . 1) . (0 . 1))
$ leval '(define a (quote cons))' '(a 1 2)'
(define . (a . ((quote . (cons . nil)) . nil)))
=> cons
(a . (1 . (2 . nil)))
=> (1 . 2)
```

L, however, lacks some important parts of Lisp, including `cond`, `label`, and `lambda`.

In the following problems, you will answer some questions about the implementation of L and extend that implementation.

Problem 3A. Scope in `define` [10 points]

As you may have noticed, in L, `(define name exp)` returns the value of `exp`. For example,

```
$ leval '(cons (define a 5) 6)' '(cons a (quote nil))'
(cons . ((define . (a . (5 . nil))) . (6 . nil)))
=> (5 . 6)
(cons . (a . ((quote . (nil . nil)) . nil)))
=> (5 . nil)
```

However, the name assigned in the `define` is not available until the *next* expression is evaluated.

```
$ leval '(cons (define a 5) a)' '(cons a a)'
(cons . ((define . (a . (5 . nil))) . (a . nil)))
Error: Unknown symbol a.
=> (5 . <error>)
(cons . (a . (a . nil)))
=> (5 . 5)
```

Examine the code of L and explain why this “problem” occurs.

Problem 3B. Improving Symbolic Comparisons [10 points]

In “Revenge of the Nerds”, Paul Graham (2002) claims that one of the benefits of Lisp was “A symbol type. Symbols are effectively pointers to strings stored in a hash table. So you can test equality by comparing a pointer, instead of comparing each character.”

However, as you’ll see from the code for `l_eq`, my quick hack does a full string comparison.

Indicate how we could fix the implementation of `L` so that `l_eq` can simply do pointer comparison. (Don’t worry about making integers work correctly; `eq` is only supposed to work on symbols.)

Note that you need not make the changes to the implementation; rather, you should describe the changes to be made with sufficient detail that a competent programmer could follow those instructions. You should not change how symbols are represented.

Hint: You will probably need to do string comparison at some point; just don’t do it when we’re comparing symbols.

Problem 3C: cond Expressions [10 points]

Update `_l_eval` so that it supports `cond` expressions.

Problem 3D: lambda Expressions [10 points]

Update `_l_eval` so that it supports the application of `lambda` expressions to values.

Problem 4: Continuations and Coroutines [20 points]

One of the regular comments about continuations is that continuations provide an elegant mechanism for supporting one form of coroutining, cooperative multitasking. In such systems, each procedure includes regular calls to a suspend operation. When a procedure suspends, control passes to the next procedure.

In this problem, you will explore coroutining with continuations.

The traditional coroutining example is that of producers and consumers. The producer (or producers) and consumer (or consumers) share a common buffer. Producers add values to the buffer and consumers remove values from the buffer.

In Scheme, we might model a buffer as an object.

```
(define make-buffer
  (lambda (_capacity)
    (let ((contents (make-vector _capacity))
          (front 0)
          (back 0)
          (capacity _capacity)
          (size 0))
      (lambda (command . params)
        (cond
         ((eq? command 'empty?)
```

```

    (<= size 0))
  ((eq? command ':full?)
   (>= size capacity))
  ((eq? command ':get!)
   (let ((tmp (vector-ref contents front)))
     (set! front (modulo (+ front 1) capacity))
     (set! size (- size 1))
     tmp))
  ((eq? command ':put!)
   (vector-set! contents back (car params))
   (set! back (modulo (+ back 1) capacity))
   (set! size (+ size 1)))
  ((eq? command ':dump)
   (display "capacity: ") (display capacity) (newline)
   (display "size: ") (display size) (newline)
   (display "front: ") (display front) (newline)
   (display "back: ") (display back) (newline)
   (display "contents: ") (display contents) (newline))
  (else
   (error "Undefined command")))))))

```

>

Here is a generalized consumer.

```

(define consumer
  (lambda (buffer done? suspend consume)
    (let kernel ((count 0))
      (cond
        ((done? count)
         ((buffer ':empty?)
          (suspend)
          (kernel count)))
        (else
         (consume count (buffer ':get!))
         (kernel (+ count 1)))))))

```

Here is a generalized producer.

```

(define producer
  (lambda (buffer done? suspend create)
    (let kernel ((count 0))
      (cond
        ((done? count)
         ((buffer ':full?)
          (suspend)
          (kernel count)))
        (else
         (buffer ':put! (create count))
         (kernel (+ count 1)))))))

```

Here is a sample use of the producer and consumer in which we print out a message instead of suspending. We also use a random number generator to decide whether or not we're done (otherwise, we might loop forever).

```

(define pcl
  (lambda ()
    (let ((buf (make-buffer 10)))
      (producer buf
        (lambda (num) (= (random 20) 0))
        (lambda () (display "Suspending producer") (newline))
        (lambda (val) (random 100)))
      (consumer buf
        (lambda (num) (= (random 20) 0))
        (lambda () (display "Suspending consumer") (newline))
        (lambda (count val)
          (display (list 'consume count val))
          (newline))))))

```

If we put the producer first, as in the above, the producer will fill the buffer (which we won't see) and then repeatedly note that it is suspending itself (since the suspend operation does not work). It will then eventually give up. The consumer will then read from the buffer and print out the values. It will then repeatedly note that it is suspending itself and eventually give up.

Your goal is to figure out how to implement the `suspend` operation using continuations. This operation should, presumably, save the current continuation, get the continuation of the coroutine (or *a* coroutine), and resume that coroutine.

You may have to grab an initial continuation at some point.

Make sure to test your solution.

Citations

Graham, Paul (2002). Revenge of the Nerds. Online resource available at <http://www.paulgraham.com/icad.html>. (This version is reformatted for clarity.)

Hoare, C.A.R. (1973). Hints on Programming Language Design. Stanford Artificial Intelligence Laboratory Memo AIM-224 / STAN-CS-73-403. Computer Science Department, Stanford University. (An extended version of a keynote address at the second SIGACT/SIGPLAN Symposium on Principles of Programming Languages.)

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

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.

- "Odd number" rather than "odd numbers" [1 point]
- Points at to 95 [1 point]
- Missing prompt [SW, 1 point]
- The narrative of problem 2 mentions four subproblems, when there are only 3 [GG and SW, 1 point]
- It was unclear that the `curry2` solution could not involve the previously defined `curry2` [SW, 1 point]

Additional errors

- Sam wrote `_level` instead of `_l_eval` [EO]
- Invalid HTML [EO]