CS61A Final Exam Review Session
Thursday, May 10, 2007

The Basics

Final Examination: Tuesday, May 15 **5:00 pm—8:00 pm** **230 Hearst Gym**

Grading Status as of Thursday, May 10, 1:00 AM

Reader	**Status**
`cs61a-rb`	All grades posted
`cs61a-rc`	Homework 14 and 15 Remaining
`cs61a-rd`	Homework 15 and Project 4b Remaining
`cs61a-re`	All grades posted
`cs61a-rf`	Homework 14 and 15 Remaining

If you have missing grades (e.g. `---`), incorrect grades, or other complaints that are not past the grading deadline, let your reader know **as soon as possible**. Carbon copy your TA on this complaint.

The Metacircular Evaluator

1. Recall that `mceval.scm` tests true or false using the `true?` and `false?` procedure:
```
(define (true? x) (not (eq? x false)))
(define (false? x) (eq? x false))
```
Suppose we type the following definition into MCE:
```
MCE> (define true false)
```
What would be returned by the following expression:
```
MCE> (if (= 2 2) 3 4)
```
____

2. Suppose we type the following into `mc-eval`:
```
MCE> (define 'x (* x x))
```
This expression evaluates without error. What would be returned by the following expressions?
```
MCE> quote                                        MCE> (quote 10)
```
______________ ______________

3. Alyssa P. Hacker has noticed that many 61A students lose points by leaving out the empty frames that you get from invoking a procedure with no arguments. She asks all her friends how she can teach the students better so they won't make this mistake.

Ben Bitdiddle says, "Instead of teaching the students better, let's just modify the metacircular evaluator so that it doesn't create frames that would be empty. Then the students will be right." Modify the metacircular evaluator to implement Ben's suggestion.

(a) Is this primarily a change to eval or to apply?

(b) What specific procedure(s) will you change?

(c) Make the changes below

(d) Lem E. Tweakit notices that this change is not such a great idea. He thinks the modified metacircular evaluator will interpret certain Scheme programs incorrectly.

Under what circumstances would this modification to the evaluator change the meaning of Scheme programs run using the metacircular evaluator? (That is, what kind of program will produce different results using the modified version than using the original one?)

```
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
    ((variable? exp) (lookup-variable-value exp env))
    ((quoted? exp) (text-of-quotation exp))
    ((assignment? exp) (eval-assignment exp env))
    ((definition? exp) (eval-definition exp env))
    ((if? exp) (eval-if exp env))
    ((lambda? exp) (make-procedure (lambda-parameters exp) (lambda-body exp) env))
    ((begin? exp) (eval-sequence (begin-actions exp) env))
    ((cond? exp) (mc-eval (cond->if exp) env))
    ((application? exp) (mc-apply (mc-eval (operator exp) env)
                        (list-of-values (operands exp) env)))
    (else
     (error "Unknown expression type -- EVAL" exp))))

(define (mc-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence (procedure-body procedure)
                        (extend-environment (procedure-parameters procedure)
                                            arguments
                                            (procedure-environment procedure))))
        (else (error "Unknown procedure type -- APPLY" procedure))))

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (mc-eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))

(define (make-frame variables values)
  (cons variables values))
```

The Analyzing Evaluator

Which of the following interactions will execute faster or the same in the analyzing evaluator than in the original metacircular evaluator? Circle FASTER or SAME for each.

```
> (define (gauss-recur n) ;; sum of #s from 1 to n
     (if (= n 1)
         1
         (+ n (gauss-recur (- n 1)))))
> (gauss-recur 1000)
```

Analyzing will be: FASTER SAME

```
> (define (gauss n)
      (/ (* (+ n 1) n) 2)
> (gauss 1000)
```

Analyzing will be: FASTER SAME

Streams

1. What are the first seven terms of the following stream definition?

```
(define mystery (cons-stream 1 (cons-stream 2 (stream-map (lambda (x y) (+ x (* 2 y)))
                                                           mystery
                                                           (stream-cdr mystery)))))
```

________ ________ ________ ________ ________ ________ ________

2. Ben Bitdiddle has conveniently defined `stream-accumulate` for you below:

```
(define (stream-accumulate combiner null-value s)
    (if (stream-null? s)
        null-value
        (combiner (stream-car s) (stream-accumulate combiner null-value (stream-cdr s)
```

What happens when we do:

a) `(define foo (stream-accumulate + 0 integers))`

b) `(define bar (cons-stream 1 (stream-accumulate + 0 integers)))`

c)
```
(define baz (stream-accumulate (lambda (x y) (cons-stream x y))
                               the-empty-stream
                               integers))
```

The Lazy Evaluator

At lines A, B, C, and D, how many times have + and * been called in lazy evaluation and in applicative order evaluation?

```
(define (foo n m)
  (if (> n 10)
      (begin (display m) n)
      (begin (display n) m)))

Lazy> (define z (* 8 4))
A
Lazy> (define x ((lambda (x) x) (+ 2 2)))
B
Lazy> (define y (foo z x))
C
Lazy> y
D
```

The Nondeterministic Evaluator

A magic square is an arrangement of numbers in a square matrix, where the sum of each row, each column, and each-main diagonal is the same. For example, here is a 3x3 magic square:
2 7 6 (For the 3x3 square, each row, column, and diagonal sums to 15)
9 5 1
4 3 8

Write a procedure called `magic-square` using the nondeterministic evaluator to find 3x3 magic square configurations.

The Logic Evaluator

Rotating lists is fun, so let's do some. For the following, assume that only the rule append has been defined, as in the lecture:

Implement a rule rotate-forward so that:

(rotate-forward (1 2 3 4) ?what) ==> ?what = (2 3 4 1).

That is, the second list is the first list with the first element attached to the end instead. Assume the list is non-empty.

Let's get both sides of the story. We'd like a rule rotate-backward so that

(rotate-backward (1 2 3 4) ?what) ==> ?what = (4 1 2 3)

That is, the second list is the first list with the last element attached to the front instead. You may define other helper rules if you'd like.

Lexical and Dynamic Scope

```
> (define person 'bob)
> (define (say line)
     (display (se person 'says line)))
> (define (greet person)
     (say 'hello))
> (greet 'sam)
```

____________________ Scheme

____________________ Dynamic Scope

1. Draw an environment diagram for the preceding scheme code.

2. Draw an environment diagram for the preceding scheme code under dynamic scope.

Topics in the Past

The following are randomly selected topics from the past. If you have a question about topics not presented here, feel free to let us know during the review session.

1. Write a procedure, `(tree-member? x tree)` that takes in an element and a general tree, and returns `#t` if `x` is part of tree, and `#f` otherwise.

2. Write a procedure, `(num-sum exp)` that takes in a valid Scheme expression, and returns the sum of all numbers that occurs in that expression. For example,
   `(num-sum '(if (= 2 3) (lambda (x) (+ x 3)) 10)) ==> 18`

3. This is an attempt to write (remove-first! ls x), which destructive removes the first instance of x:
```
(define (remove-first! ls x)
    (cond ((null? ls) '())
          ((eq? (car ls) x) (set! ls (cdr ls)))
          ((eq? (cadr ls) x) (set-cdr! ls (cddr ls)))
          (else (remove-first! (cdr ls) x))))
```

   Suppose (define L '(a b c)). What happens to L when we do:

   a. `(remove-first! L 'a)`
      `L ==>` __________________________

   b. `(remove-first! L 'b)`
      `L ==>` __________________________

   c. `(remove-first! L 'z)`
      `L ==>` __________________________

4. Consider `foo`, a popular name for mystery functions.

```
(define foo
    (let ((L (list 1)))
      (lambda (a) (let ((M (cons a L)))
                   (lambda (b) (set! L (cons (+ a 1) L))
                               (set! a (+ a 2))
                               (set-car! (cdr M) (+ 3 (cadr M)))
                               (set! b (+ b 4))
                               (list a b L M))))))
```

Fill in the blanks below

```
> (define f (foo 1))
> (f 2)
____________________________
> (define g (foo 2))
> (g 1)
____________________________
> (f 2)
____________________________
> (g 1)
____________________________
```

5. There are just too many things to do and too much to keep track of during finals week. We would like to implement a system that helps us keep track of what we need to do. To that end, we propose a `job` class. A `job` object is a certain amount of work required; you can do some amounts of work until the work is done. A job also has an automatically assigned id. A job is "done" when there's no more work to do for that job. We also want to keep track of number of jobs we've created. The following illustrates how you would interact with the job object.

```
STk> (define job1 (instantiate job ‘(take out trash) 10))
job1
STk> (define job2 (instantiate job ‘(eat dinner) 5))
job2
STk> (define job3 (instantiate job ‘(study for cs61a) 100000))
job3
STk> (ask job1 ‘id)
0
STk> (ask job2 ‘id)
1
STk> (ask job2 ‘work-to-do)
5
STk> (ask job1 ‘do-work 10)
okay
STk> (ask job1 ‘work-to-do)
0
STk> (ask job1 ‘done?)
#t
STk> (ask job1 ‘do-work 10)
(but you are already done!)
STk> (ask job ‘number-of-jobs)
3
```

```
(define-class (job description work-to-do)

)
```