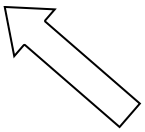
CS61A Midterm 3 Review Session

Monday, November 13, 2006

The Basics

Midterm 3: Wednesday, November 15 **7:00 pm—9:00 pm** **155 Dwinelle (Note the location!)**



1. Who is your TA? When is his section?
2. What is your login?
3. Do you go to 1 Pimentel for the midterm?
4. Should you sign the first page of the midterm under "read and sign this"?

Object-Oriented Programming

(From Midterm 3, Spring 2004) We would like to build a client class and a server class to simulate instant-messaging. (Unlike the IM program you saw in lab, this is just a simulation; the clients and servers are all just objects in the same program on the same computer.) Instances of the client class would connect to an instance of the server class, and the client objects would be able to send each other messages by way of the server.

A server has no instantiation variables. A client is instantiated with the name of the client and the server to connect to. Upon instantiation, it should connect to the given server.

You can ask a client for the most recent message that it has received. You can also ask a client to send a message by specifying the message to send and the name of the client to send to. For this problem, assume that there is a client by that name connected to the server (so you don't need to check this).

This is the desired interaction:

```
STk> (define s (instantiate server))
s
STk> (define brian (instantiate client 'brian s))
brian
STk> (define gap (instantiate client 'gap s))
gap
STk> (ask gap 'latest-message)
()
STk> (ask brian 'send-message '(where are the potstickers?) 'gap)
okay
STk> (ask gap 'latest-message)
(where are the potstickers?)
```

```
(define-class (client name server)
   (instance-vars (latest-message '()))
   (initialize ____________________________________
               ____________________________________
               ____________________________________)
   (method (receive-message msg)
      (set! latest-message msg))
   (method (send-message msg who)
      (ask server 'send-to-client msg who)))

(define-class (server)
   (instance-vars (clients '()))
   (method (accept-connection client)
      (set! clients (cons client clients)))
   (method (send-to-client msg name)
       ____________________________________
       ____________________________________
       ____________________________________))
```

Object-Oriented Programming

Sometimes, we want to send a message to every client that's connected to the server.

We write a subclass of the client class to allow this:

```
(define-class (broadcast-client name server)
   (parent (client name server))
   (method (broadcast msg)
   (ask server 'broadcast msg)))
```

The new broadcast method takes in a message to broadcast, and does not take in the name of the person to send to, since the message will be sent to everyone. The method simply calls the server object's broadcast method, which you will implement. Also, every time a new client connects to the server, we would like the server to send a message to every client that someone new has joined us. The desired interaction is like this:

This is the desired interaction:

```
STk> (define bs (instantiate broadcast-server))
bs
STk> (define brian (instantiate broadcast-client 'brian bs))
brian
STk> (define gap (instantiate broadcast-client 'gap bs))
gap
STk> (ask brian 'latest-message)
(gap has joined us)
STk> (define robert (instantiate broadcast-client 'robert bs))
robert
STk> (ask brian 'latest-message)
(robert has joined us)
STk> (ask gap 'latest-message)
(robert has joined us)
STk> (ask brian 'broadcast '(How is the midterm?))
okay
STk> (ask robert 'latest-message)
(how is the midterm?)
STk> (ask gap 'latest-message)
(how is the midterm?)
STk> (ask robert 'send-message '(should be much harder) 'brian)
okay
STk> (ask brian 'latest-message)
(should be much harder)
STk> (ask gap 'latest-message)
(how is the midterm?)
```

Fill in the blanks for the incomplete implementation of the broadcast-server class below that allows us to do this:

```
(define-class (broadcast-server name)
   (parent (server name))
   (method (accept-connection client)
      _____________________________________________________
      _____________________________________________________
      _____________________________________________________
      _____________________________________________________
   (ask self 'broadcast
      (cons (ask client 'name) '(has joined us))))
   (method (broadcast msg)
      (for-each (lambda (c) (ask c 'receive-message msg)))
```

Environment Diagrams

1.
```
(define (kons a b)
   (lambda (m)
      (if (eq? m 'kar) a b)))

(define p (kons (kons 1 2) 3))
```

2.
```
(define x 3)
(define y 4)
(define foo
   ((lambda (x) (lambda (y) (+ x y)))
    (+ x y)))
(foo 10)
```

3.
```
(define x 17)

(define f
   (let ((x 4))
      (lambda (y)
         (print x)
         (set! x y)
         y)))
```

Mutable Data

1.
```
(let ((x (list 1 2 3 4)))
   (set-car! (cddr x) x)
   x)
```

2.
```
(let ((x (list 1 2 3)))
   (set-car! x (cdr x))
   (set-cdr! (car x) 5)
   x)
```

3.
```
(let ((x (list 1 2 3)))
   (set-car! x (list 'a 'b 'c))
   (set-car! (cdar x) 'd)
   x)
```

4. Question 2 of MT3, Fall 1994
   Write `make-alist!`, a procedure that takes as its argument a list of alternating keys and values, like this:
   `(color orange zip 94720 name wakko)`
   and changes it, by mutation, into an association list, like this:
   `((color . orange) (zip . 94720) (name . wakko))`
   You may assume that the argument list has an even number of elements. The result of your procedure requires exactly as many pairs as the argument, so you will work by rearranging the pairs in the argument itself. Do not allocate any new pairs in your solution!

5. Question 2 of MT3, Spring 1996
   Write `list-rotate!` which takes two arguments, a nonnegative integer `n` and a list `seq`. It returns a mutated version of the argument list, in which the first `n` elements are moved to the end of the list, like this:
```
> (list-rotate! 3 (list 'a 'b 'c 'd 'e 'f 'g))
(d e f g a b c)
```
   You may assume that $0 \leq n < $ `(length seq)` without error checking.
   Note: Do not allocate any new pairs in your solution. Rearrange the existing pairs.

Vectors

1. You've seen vectors. You've seen tables. We want to implement tables with vectors. To make things easier, we're going to assume that there are two vector tables that manage keys and values. Assume that the corresponding key/value pair has the same index.

   Write `vector-lookup` for vector-tables. It acts just like lookup in tables. It takes a key and returns the corresponding value if the key is in the table and `#f` otherwise.

2. Write `vector-reverse!` that does the obvious thing. Do not create new vectors!
   Hint: You might want to write a helper called `vector-swap!` that takes in a vector and two indices and swap the elements at those indices.

Concurrency

1. What are the possible values of x after the following is executed:

```
(define x 10)
(parallel-execute (lambda () (set! x (+ 5 x)) (set! x (* x 3)))
                  (lambda () (if (> x 16)
                                 (set! x 100)
                                 (set! x (- x 20)))))
```

2. Question 13 of the Final Exam, Spring 2003
   Given the following definitions:

```
(define s (make-serializer))
(define t (make-serializer))
(define x 10)
(define (f) (set! x (+ x 3)))
(define (g) (set! x (* x 2)))
```

   Can the following expressions produce an incorrect result, a deadlock, or neither? (By "incorrect result" we mean a result that is not consistent with some sequential ordering of the processes.)

   (a) `(parallel-execute (s f) (t g))`

   (b) `(parallel-execute (s f) (s g))`

   (c) `(parallel-execute (s (t f)) (t g))`

   (d) `(parallel-execute (s (t f)) (s g))`

   (e) `(parallel-execute (s (t f)) (t (s g)))`