MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.5150 Spring 2025
Problem Set 8

Issued: Wed. 9 April 2025 **Due: Fri. 25 April 2025**

This is the LAST PROBLEM SET for 6.5151! Whew!
You have two weeks to work on this problem set.
There will be one more problem set for 6.5150 students only.

Everybody: Please work on your projects. Draft project reports are due on Wednesday, 30 April 2025. You will present your projects on the week of May 5 through May 9. Final project reports are due Friday May 9.

We cannot accept late problem sets or papers after the last class, which is Tuesday May 13.

Readings:
Online MIT/GNU Scheme Documentation,
Section 2.3: Dynamic Binding - parameterize and fluid-let
Section 12.4: Continuations - call-with-current-continuation

Here is a nice paper about continuations and threads:
http://repository.readscheme.org/ftp/papers/sw2003/Threads.pdf
In fact, there is an entire bibliography of stuff about this on:
http://library.readscheme.org/page6.html

Code: load.scm utils.scm queue.scm schedule.scm actors.scm syntax.scm time-share.scm

The Actor Model of Concurrency

We have examined traditional time-sharing in lecture, and the propagator system we will be playing with is another way to think about multiple computing agents acting in parallel: each propagator can be thought of as an independent computing agent.

In 1973 Carl Hewitt came up with an idea for formulating concurrency, based on message passing, called "actors." (Yes, that is the same Carl Hewitt who had the idea of match combinators.) Very crudely, Hewitt's idea is that when you call an actor (send it a message) the call immediately returns to the caller, but the message is put on a queue of work for the actor to do. Each actor processes the elements of its input queue, perhaps changing its local state as it does so. In Hewitt's actors the state update is accomplished by the actor "replacing itself" with a new behavior. If the intention of the

caller was to receive a reply it would have to pass its return continuation as an argument to the called actor as part of its calling message.

In Hewitt's ideal system there were no computational objects that were not actors, including integers. So to add "2" to "3" you send an "add" message with a return address to "2". You then receive a 2-adder to which you send a "3" and a return address which will receive "5". This made a very elegant object-oriented system, where everything was implemented by message passing. See the article http://en.wikipedia.org/wiki/Actor_model for more information.

Here I will be less orthodox and consider an actor-inspired system that adds an actor-like procedure to an ordinary Scheme-like lambda-calculus interpreter. This is more to my (GJS) liking, because it does not require everything to be an actor -- I hate grand theories and magic bullets that restrict a programmer to do things by someone's grand theory. The languages Erlang and Scala both provide mechanisms to support actor-like behaviors.

So, let's get into it. We provide an "actor" extension for MIT/GNU Scheme: You can load it with (load "load"). We initialize it with (init-actors). This is a work in progress. THERE ARE BUGS! However there is much that we can learn from this system.

In this system we can run ordinary programs in the usual way:

```
(define fib
  (lambda (n)
    (if (< n 2)
        n
        (+ (fib (- n 1))
           (fib (- n 2))))))
;Value: fib

(fib 10)
;Value: 55
```

But we can also define the Fibonacci computer in terms of actors, using "alpha expressions" rather than "lambda expressions":

```
(define fib1
  (alpha (n c)
         (if (< n 2)
             (c n)
             (let ((x 'not-ready) (y 'not-ready))
               (define wait-for-xy
                 (alpha (k)
                        (if (boolean/or
                             (eq? x 'not-ready)
                             (eq? y 'not-ready))
                            (wait-for-xy k)
                            (k (+ x y)))))
               (wait-for-xy c)
               (fib1 (- n 1)
                     (lambda (v) (set! x v)))
               (fib1 (- n 2)
                     (lambda (v) (set! y v)))))))
;Value: fib

(fib1 10 write-line)
;Value: task-added

55
```

Here fib1 is an actor that takes a number and a continuation to call with the value. If the number is not less than 2 it sets up two state variables, x and y, to capture the values of the recursive calls. It also defines an actor procedure that busy-waits until x and y are available. The continuations of the recursive calls are just assignments to the capture variables. In cases where we don't want to start a concurrent process, we use lambda rather than alpha.

We see that the call (fib1 10 write-line) returns immediately with the value task-added and later we see the answer 55 appearing. We also see that this is very slow, because there are many calls to fib busy waiting! The busy waiter should really yield instead of looping, as follows:

If we write:

```
(define fib2
  (alpha (n c)
         (if (< n 2)
             (c n)
             (let ((x 'not-ready) (y 'not-ready))
               (define wait-for-xy
                 (alpha (k)
                        (if (boolean/or
                             (eq? x 'not-ready)
                             (eq? y 'not-ready))
                            (begin (yield)
                                   (wait-for-xy k))
                            (k (+ x y)))))
               (wait-for-xy c)
               (fib2 (- n 1)
                     (lambda (v) (set! x v)))
               (fib2 (- n 2)
                     (lambda (v) (set! y v)))))))
```

This code runs many times faster.

Implementation

The implementation is really quite simple. Each actor is an executable to-do procedure, (an ordinary Scheme procedure) associated with a data structure to hold it, its queue of tasks, and its runnability. This association is an "apply hook" (see refrence manual section 12.1). When called with arguments the apply-hook procedure (not the to-do) adds a task to its queue of tasks. Each task is a thunk that when called applies the to-do procedure to the arguments supplied.

```
(define-record-type <actor>
    (make-actor to-do task-queue runnable)
    actor?
  (to-do get-actor-to-do)
  (task-queue get-actor-task-queue set-actor-task-queue!)
  (runnable actor-runnable? set-actor-runnable!))

(define (make-actor-procedure to-do)
  (let ((actor (make-actor to-do (queue:make) #f)))
    (let ((proc
           (make-apply-hook
            (lambda args
              (add-to-tasks! actor
                             (lambda () (apply to-do args))))
            actor)))
      (set! all-actors (cons actor all-actors))
      proc)))

(define (actor-procedure? x)
  (and (apply-hook? x)
       (actor? (apply-hook-extra x))))
```

Notice that we keep a list of all the actors, so we can initialze them, as necessary.

To add a task to an actor we must add it to the actor's task queue and we must make the actor runnable. This must all be done atomically, because of concurrancy.

```
(define (add-to-tasks! actor task)
  (atomically
    (lambda ()
      (queue:add-to-end! (get-actor-task-queue actor) task)
      (add-to-runnable! actor)))
  'task-added)

(define (add-to-runnable! actor)
  (if (actor-runnable? actor)
      'already-runnable
      (begin (set-actor-runnable! actor #t)
             (queue:add-to-end! runnable-actors actor)
             'made-runnable)))
```

The alpha expression syntax is implemented as a macro:

```
(define-syntax alpha
  (er-macro-transformer
   (lambda (form rename compare)
     (let ((bound-variables (cadr form))
           (body (cddr form))
           (r-make-actor-procedure
            (rename 'make-actor-procedure))
           (r-lambda (rename 'lambda)))
       `(,r-make-actor-procedure
         (,r-lambda ,bound-variables ,@body))))))
```

There is some basic initialization:

```
(define all-actors '())
(define runnable-actors)

(define (init-actors)
  (for-each (lambda (actor)
              (set-actor-task-queue! actor
                                     (queue:make))
              (set-actor-runnable! actor #f))
            all-actors)
  (set! runnable-actors (queue:make))
  (start-time-sharing))
```

So now all that is left is the scheduling and time sharing. To schedule we find a runnable actor with task to be done: The procedure switch-tasks is the scheduler. It either returns a task to be performed or it signals that there are no tasks to be performed or that a runnable actor actually has no work to do.

```
(define (switch-tasks)
  (define (get-task)
    (if (queue:empty? runnable-actors)
        nothing-to-do
        (let ((actor (queue:get-first! runnable-actors)))
          (let ((actor-task-queue (get-actor-task-queue actor)))
            ;; If there are no more tasks for this actor,
            ;; it is not runnable.
            (if (queue:empty? actor-task-queue) ; Nothing
                (begin (set-actor-runnable! actor #f)
                       'try-again)        ;perhaps another actor?
                (let ((task (queue:get-first! actor-task-queue)))
                  ;; Got the first task, and removed it.
                  ;; If there are no more tasks for this actor,
                  ;; it is not runnable.
                  ;; If there are more tasks for this actor,
                  ;; requeue it.
                  (if (queue:empty? actor-task-queue)
                      (set-actor-runnable! actor #f)
                      (queue:add-to-end! runnable-actors actor))
                  ;; Return task to be executed.
                  task))))))
  (let lp ((task (atomically get-task)))
    (if (eq? task 'try-again)
        (lp (atomically get-task))
        task)))

(define nothing-to-do
  (list 'nothing-to-do))

(define (nothing-to-do? task)
  (eq? task nothing-to-do))
```

And now for the time-sharing of tasks. There is a preemptive timer interrupt that runs this whole mess. There is an MIT/GNU Scheme specific detail: register-timer-event is the MIT/GNU Scheme mechanism for delivering a timer interrupt. When the time specified by its first argument expires, it invokes the second argument. This part of the system was very hard to get right: GJS was up all night debugging it!

```
;;;; Preemptive scheduling for time-sharing.

(define time-sharing:quantum 1)
(define time-sharing-enabled? #t)
(define time-sharing? #f)

;;; This is an MIT/GNU Scheme specific detail.
;;; register-timer-event is the MIT/GNU Scheme mechanism for
;;; delivering a timer interrupt.  When the time specified
;;; by its first argument expires, it invokes the second
;;; argument.

(define (start-time-sharing)
  (set! time-sharing? time-sharing-enabled?)
  (define (setup-interrupt)
    (if time-sharing?
        (register-timer-event time-sharing:quantum
                              on-interrupt)))
  (define (on-interrupt)
    (setup-interrupt)
    (yield))

  (setup-interrupt))                    ; Initialize interrupt

(define (stop-time-sharing)
  (set! time-sharing? #f))

;;; SWITCH-TASKS can return either NOTHING-TO-DO or a
;;; thunk that represents a piece of work to be done.

(define (yield)
   (let ((next-task (switch-tasks)))
     (if (not (nothing-to-do? next-task))
         (begin
           ;; The interrupt routine, including yield, is
           ;; not interruptable, but we must allow the
           ;; task to be run to be interruptable!  A mess.
           (unblock-thread-events)
           (set-interrupt-enables! interrupt-mask/all)
           (next-task)))))
```

```
;;; without-interrupts is an MIT Scheme mechanism that turns
;;; off all timer interrupts.  It executes its thunk
;;; atomically.

(define (atomically thunk)
  (without-interrupts thunk))

(define (double-check-lock check do if-not)
  (let ((outside
         (atomically
          (lambda ()
            (if (check)
                (begin (do)
                       (lambda () 'ok))
                if-not)))))
    (outside)))
```

Problem 9.1:

Here is a simple program that is implemented with actors.

```
  (define (foo n)
    (define (buzz m)
      (if (not (= m 0)) (buzz (- m 1))))
    (define iter
      (alpha (l i)
        (if (not (= i 0))
            (begin
              (if (eq? l 'a) (buzz (* 100 i)) (buzz (* 100 (- n i))))
              (pp (list l i))
              (iter l (- i 1))))))
    (iter 'a n)
    (iter 'b n))
```

When I ran this program I got the following output:

```
(foo 10)
;Value: task-added

(b 10)
(b 9)
(a 10)
(a 9)
(a 8)
(b 8)
(b 7)
(a 7)
(a 6)
(a 5)
(b 6)
(b 5)
((b 4)
a 4)
(a 3)
(a 2)
(b 3)
(b 2)
(b 1)
(a 1)
```

You may get a different order than I got. Why do we get this output in such a strange order?

Part a: Explain what you see here.

Part b: The fact that the overlap of two printing (PP) calls appeared here is due to the lack of locking of the output port, so each process has exclusive access to it. An actor-like way to fix this is to make PP an actor, so its commands are serialized as separate tasks. Write the code to implement that, and dmonstrate it.

Problem 9.2:

Note that we have made a choice in the handler for application of actor procedures: we defer only the execution of the body. Is this choice a good one? Suppose we wanted to also defer the evaluation of the operands?

Explain, in a short clear paragraph your opinion on this matter.

Problem 9.3:

The procedure double-check-lock is used in set-variable-value! and define-variable!.

```
(define (double-check-lock check do if-not)
  (let ((outside
         (atomically
          (lambda ()
            (if (check)
                (begin (do)
                       (lambda () 'ok))
                if-not)))))
    (outside)))
```

Why is it needed? Explain both why it is needed and how it works.

Problem 9.4:

The first "actor implementation" of the Fibonacci procedure was

```
(define fib1
  (alpha (n c)
    (if (< n 2)
        (c n)
        (let ((x 'not-ready) (y 'not-ready))
          (define wait-for-xy
            (alpha (k)
                   (if (boolean/or (eq? x 'not-ready)
                                   (eq? y 'not-ready))
                       (wait-for-xy k)
                       (k #t))))
          (fib1 (- n 1) (lambda (v) (set! x v)))
          (fib1 (- n 2) (lambda (v) (set! y v)))
          (wait-for-xy (lambda (ignore) (c (+ x y))))))))
```

a. We explained why is this so slow compared with the traditional version of fib defined as a doubly-recursive Scheme function in the same interpreter.

Perhaps it is worthwhile to consider the following alternative implementation of Fibonnaci:

```
(define fib3
  (alpha (n c)
    (if (< n 2)
        (c n)
        (let ((x 'not-ready) (y 'not-ready))
          (define check-if-done
            (lambda ()
              (if (boolean/or (eq? x 'not-ready)
                              (eq? y 'not-ready))
                  #f
                  (c (+ x y)))))
          (fib3 (- n 1)
                (lambda (v) (set! x v) (check-if-done)))
          (fib3 (- n 2)
                (lambda (v) (set! y v) (check-if-done)))))))
```

The essential difference is that the end test in the first version is implemented as an actor loop, whereas in the second version it is implemented as a Scheme procedure that terminates by returning #f rather than calling a continuation. This returns very fast.

However, if we try to do the same with an actor version of check-if-done, we get a weird result!

```
(define fib4
  (alpha (n c)
    (if (< n 2)
        (c n)
        (let ((x 'not-ready) (y 'not-ready))
          (define check-if-done
            (alpha (c)
              (if (boolean/or (eq? x 'not-ready)
                              (eq? y 'not-ready))
                  #f
                  (c (+ x y)))))
          (fib4 (- n 1)
                (lambda (v)
                  (set! x v)
                  (check-if-done c)))
          (fib4 (- n 2)
                (lambda (v)
                  (set! y v)
                  (check-if-done c)))))))

(fib4 10 write-line)
;Value: task-added

55
55
```

b. Explain why we get two copies of 55. Can you fix this?

Futures

The procedures that were illustrated in Problem 9.4 share an idea. An actor is started; when it finishes it sets a return value in the caller. The caller must check when the results of pending computations are completed for it to proceed to use those results. This pattern should be abstracted. One way is by the introduction of "future"s. (This abstraction was introduced by Robert H. Halstead, Jr. in the early 1980's. Sometimes futures are called "promises", but this word usually refers to delay thunks.) With futures, the code for Fibonacci looks like:

```
(define fib
  (alpha (n c)
    (if (< n 2)
        (c n)
        (let ((xp (future (lambda (k) (fib (- n 1) k))))
              (yp (future (lambda (k) (fib (- n 2) k)))))
          (wait xp
            (lambda (x)
              (wait yp (lambda (y) (c (+ x y))))))))))

(fib 10 write-line)
;Value: task-added
55
```

Notice that the future converts the continuation into an object whose fulfillment one can wait for.
A future is a mutable data structure. It contains a continuation, a "done" flag, a value. When a future gets fulfilled, its value is stored, and the done flag is set.

Problem 9.5:

Here we implement futures. The implementation should be added to the guest interpreter.

a. Your first job is to construct the data structure for a future. We recommend using record structures for this.
b. Implement procedures FUTURE and WAIT. Make sure that your WAIT does not hang: it should return immediately but put a continuation on a queue of things to do. Assume here that a future is waited on only once. Make sure that you protect critical regions with ATOMICALLY.
c. What goes wrong with your implementation if the assumption is violated?
d. Improve your implementation to relax the assumption.
