6.001 Recitation 22: Streams

RI: Gerald Dalley, dalleyg@mit.edu, 4 May 2007
http://people.csail.mit.edu/dalleyg/6.001/SP2007/

"The three chief <u>virtues</u> of a programmer are: Laziness, Impatience and Hubris (Larry Wall)."

Reviewing Laziness

The whole point of lazy evaluation is to put off doing work as long as you can. There are two ways of doing that. One is to be explicit:

```
; Returns a promise to evaluate exp in the current environment
; -- the promise is a ``thunk'' that remembers the expression and the current environment
(delay exp)
; Returns the value of exp in the original environment
(force promise)
```

It's easier if everything is implicit. In this case, all arguments to functions are delayed, and values are forced when needed. How many cases can you think of?

-
-
-
-

Streaming Basics

Lazy evaluation has a bunch of interesting applications, but one that we're most interested in is constructing streams, (usually) infinite data structures. A stream is just like a list, but the `cdr` of each `cons` cell is lazy-memoized – it's a memoized thunk.

For now, assume that we have the following Scheme procedures and/or special forms:

```
; Creates a stream pair, evaluating x, but wrapping y in a thunk
(define s (cons-stream x y))

; Returns the car part of the stream (x in this case).
; Not a thunk (unless x evaluated to a thunk).
(stream-car s)

; Returns the cdr part of the stream pair (same as (delay y)).
; This is _always_ a thunk or an evaluated thunk.
(stream-cdr s)
```

We'll use the following notation to graphically represent various Scheme objects[1]:

---
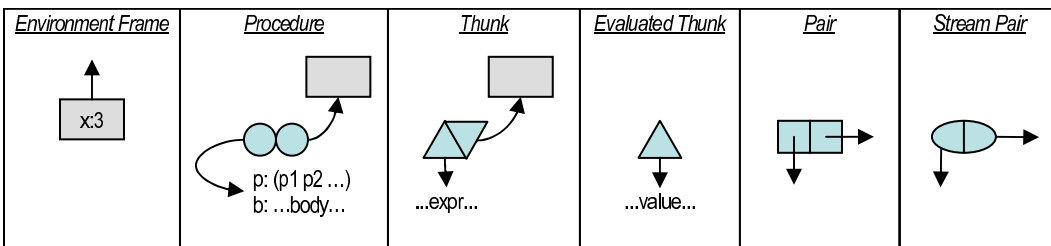
[1]Our representations of thunks as a double-triangle and evaluated thunks as a single triangle are non-standard, so don't be surprised if other instructors use other notation.

One Infinity

Let's walk through a simple example of evaluating streams using the environment model. Here we'll create perhaps the simplest infinite stream possible: a stream of ones.

```
(define ones (cons-stream 1 ones))
```
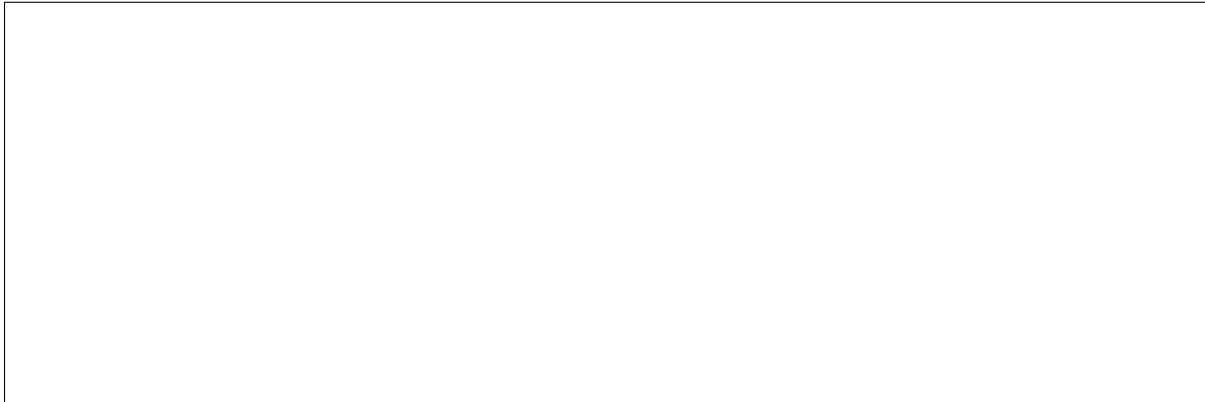
Draw the environment diagram resulting from the evaluation of this expression.

Now suppose we evaluate

```
(stream-cdr ones)
```

Draw the environment diagram resulting from the evaluation of this expression.

Integer Stream

We can now look at a slightly more interesting example: a stream of all positive integers. We'll first define a helper procedure:

```
(define (ints-from-n x) (cons-stream x (ints-from-n (+ x 1))))
```

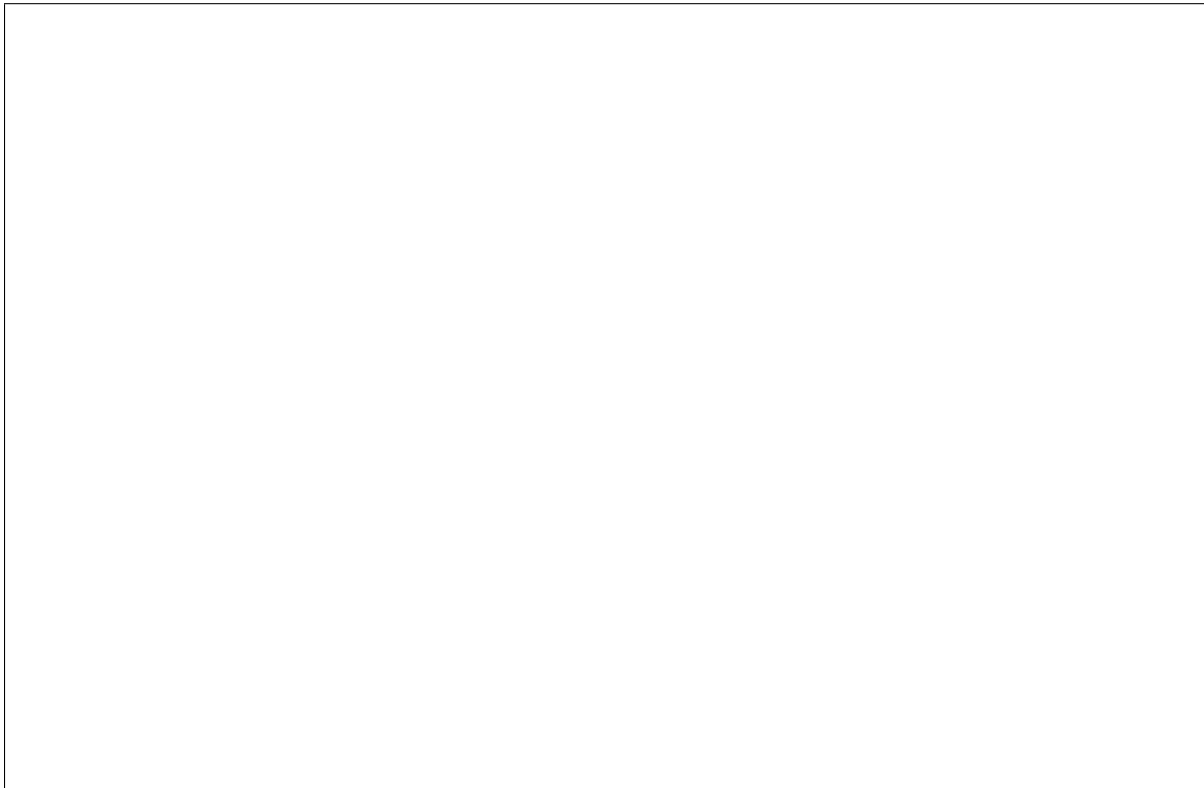Draw the environment diagram resulting from the evaluation of this expression.

Draw the environment diagram resulting from the following expressions:

```
(define ints (ints-from-n 1))

(display (stream-cdr ints)) ; Q: does display cause an infinite loop?

(stream-cdr (stream-cdr ints))
```

Cartography

Soon after introducing pairs and lists, we found that we could save a lot of effort by creating some standard higher-order procedures such as `map`, `filter`, and `foldr`. In lecture, we saw an implementation of `stream-filter` and `stream-accum`. Let's implement `map-streams`:

```scheme
; Like map, but for streams.  Can operate on any number of streams.
; If any input streams are finite, the resulting stream's length is the length
; of the shortest input stream.
(define (map-streams f . args)
```

Using `map-streams`, implement `add-streams`:

```scheme
; Does element-wise addition of the input streams
(define (add-streams . args)
```

Fibonacci

Create a stream, `fibstream` that is stream of Fibonacci numbers. The Fibonacci series goes

```
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 ...)
```

It starts with two 1's, then each successive value is the sum of the previous two. Hint: use `map-streams`.

```scheme
(load "streams.scm")(load "assert.scm")
(define fibstream
```

Bonus Problem

Think about how you could modify the project 5 meta-circular evaluator to support streams. Hint 1: copy the thunk code from the lazy evaluator in the online tutor for last week. Hint 2: you only need to change 3 lines of code and add about 10 new lines (not including the thunk implementation).

Solutions

Reviewing Laziness

The whole point of lazy evaluation is to put off doing work as long as you can. There are two ways of doing that. One is to be explicit:

```scheme
; Returns a promise to evaluate exp in the current environment
; -- the promise is a ''thunk'' that remembers the expression and the current environment
(delay exp)
; Returns the value of exp in the original environment
(force promise)
```

It's easier if everything is implicit. In this case, all arguments to functions are delayed, and values are forced when needed. How many cases can you think of?

- Used as an argument to a primitive procedure.
- Used in a conditional.
- Used as an operator.
- Printing the value to screen (very implementation-dependent).

Streaming Basics

Lazy evaluation has a bunch of interesting applications, but one that we're most interested in is constructing streams, (usually) infinite data structures. A stream is just like a list, but the `cdr` of each `cons` cell is lazy-memoized – it's a memoized thunk.
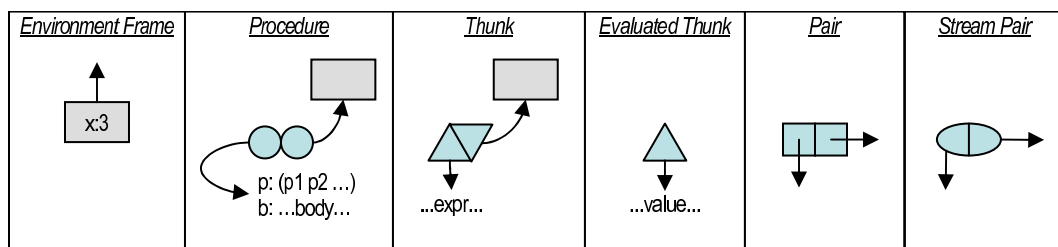
For now, assume that we have the following Scheme procedures and/or special forms:

```scheme
; Creates a stream pair, evaluating x, but wrapping y in a thunk
(define s (cons-stream x y))

; Returns the car part of the stream (x in this case).
; Not a thunk (unless x evaluated to a thunk).
(stream-car s)

; Returns the cdr part of the stream pair (same as (delay y)).
; This is _always_ a thunk or an evaluated thunk.
(stream-cdr s)
```

We'll use the following notation to graphically represent various Scheme objects[2]:

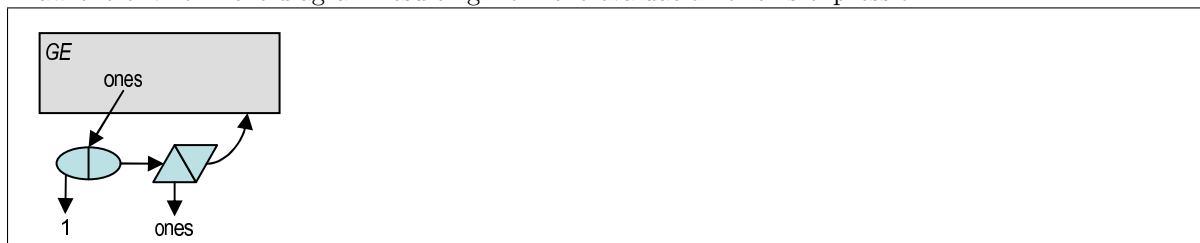| Environment Frame	Procedure	Thunk	Evaluated Thunk	Pair	Stream Pair

One Infinity

Let's walk through a simple example of evaluating streams using the environment model. Here we'll create perhaps the simplest infinite stream possible: a stream of ones.

```scheme
(define ones (cons-stream 1 ones))
```

---

[2]Our representations of thunks as a double-triangle and evaluated thunks as a single triangle are non-standard, so don't be surprised if other instructors use other notation.

Draw the environment diagram resulting from the evaluation of this expression.

Now suppose we evaluate

```
(stream-cdr ones)
```

Draw the environment diagram resulting from the evaluation of this expression.
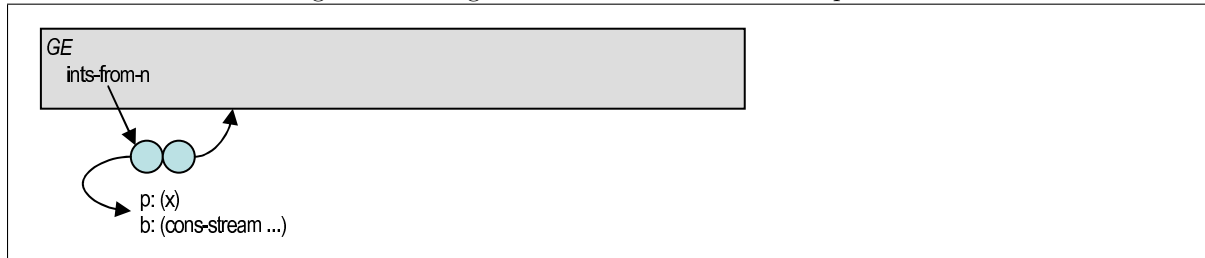
Integer Stream

We can now look at a slightly more interesting example: a stream of all positive integers. We'll first define a helper procedure:
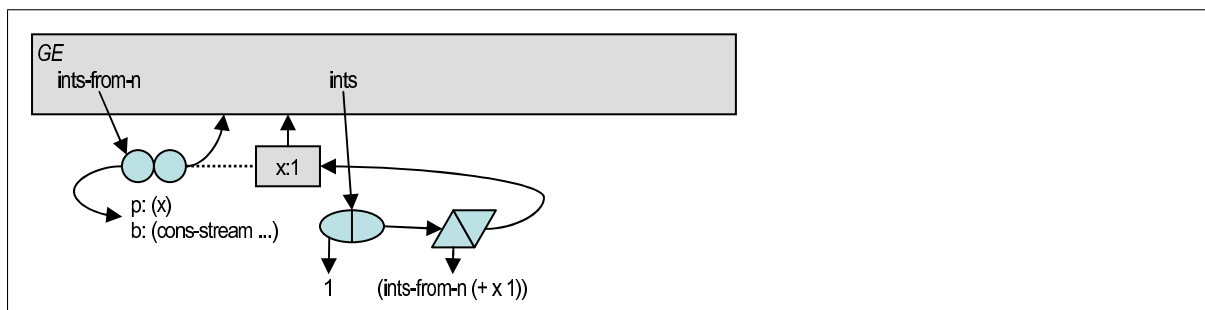
```
(define (ints-from-n x) (cons-stream x (ints-from-n (+ x 1))))
```

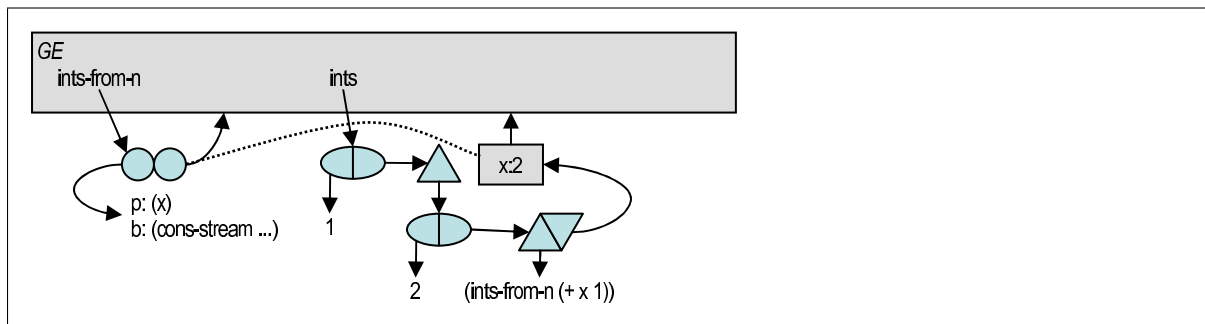Draw the environment diagram resulting from the evaluation of this expression.

Draw the environment diagram resulting from the following expressions:
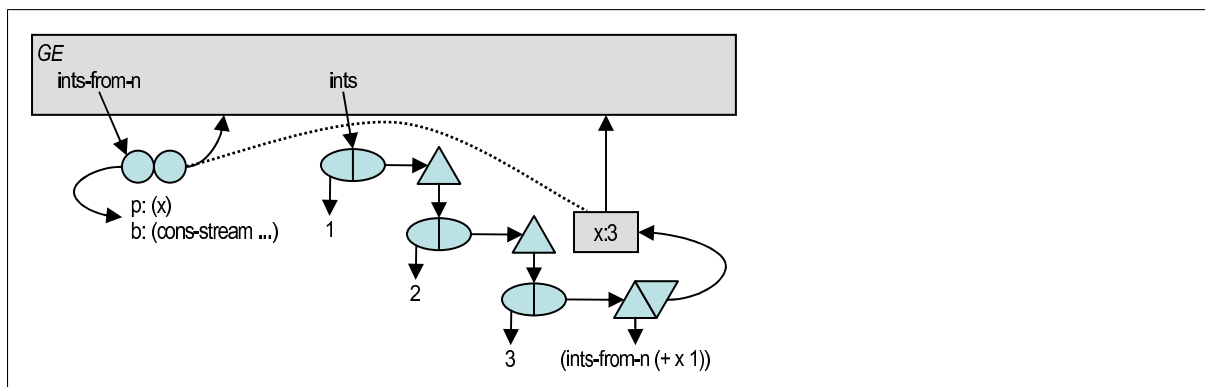
```
(define ints (ints-from-n 1))
```



```
(display (stream-cdr ints))  ; Q: does display cause an infinite loop?
```



```
(stream-cdr (stream-cdr ints))
```

Cartography

Soon after introducing pairs and lists, we found that we could save a lot of effort by creating some standard higher-order procedures such as `map`, `filter`, and `foldr`. In lecture, we saw an implementation of `stream-filter` and `stream-accum`. Let's implement `map-streams`:

```
; Like map, but for streams.  Can operate on any number of streams.
; If any input streams are finite, the resulting stream's length is the length
; of the shortest input stream.
(define (map-streams f . args)
  (if (not (null? (filter empty-stream? args)))
      the-empty-stream
      (cons-stream (apply f (map stream-car args))
                   (apply map-streams (cons f (map stream-cdr args))))))
```

Using `map-streams`, implement `add-streams`:

```
; Does element-wise addition of the input streams
(define (add-streams . args)
  (apply map-streams + args))
```

Fibonacci

Create a stream, `fibstream` that is stream of Fibonacci numbers. The Fibonacci series goes

```
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 ...)
```

It starts with two 1's, then each successive value is the sum of the previous two. Hint: use `map-streams`.

```
(load "streams.scm")(load "assert.scm")
(define fibstream
  (cons-stream
   1
   (cons-stream
    1
    (map-streams + (stream-cdr fibstream) fibstream))))

(assert-equal
 (print-stream fibstream 20)
 '(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765))
```

Bonus Problem

Think about how you could modify the project 5 meta-circular evaluator to support streams. Hint 1: copy the thunk code from the lazy evaluator in the online tutor for last week. Hint 2: you only need to change 3 lines of code and add about 10 new lines (not including the thunk implementation).

```
;;
;; eval.scm - 6.001 Spring 2007
;;
;;    Modified from project 5's meta-circular evaluator to add streams.
;;    Look for comments with "GED" to find the important modifications.
;;

(define (m-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) (m-eval (cond->if exp) env))
        ((let? exp) (m-eval (let->application exp) env))
        ; GED -- cons-stream must be a special form since we don't have any other
        ;        lazy evaluation in this interpreter.
```

```scheme
          ((cons-stream? exp) (eval-cons-stream exp env))
          ((application? exp)
           ; GED -- We must get the actual operator procedure to apply it.  It' can't
           ;         apply a procedure if it doesn't actually have that procedure.
           (m-apply (actual-value (operator exp) env)
                    (list-of-values (operands exp) env)))
          (else (error "Unknown expression type -- EVAL" exp))))

(define (m-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         ; GED -- primitive procedures don't understand our thunks, so we must force
         ;         the actual values before applying the primitive.
         (apply-primitive-procedure procedure
                                    (map force-it 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))))

;;
;; this section includes syntax for evaluator
;; selectors and constructors for scheme expressions
;;

(define (tagged-list? exp tag)
  (and (pair? exp) (eq? (car exp) tag)))

(define (self-evaluating? exp)
  (or (number? exp) (string? exp) (boolean? exp)))

(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))

(define (variable? exp) (symbol? exp))
(define (assignment? exp) (tagged-list? exp 'set!))
(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))
(define (make-assignment var expr)
  (list 'set! var expr))

(define (definition? exp) (tagged-list? exp 'define))
(define (definition-variable exp)
  (if (symbol? (cadr exp))   (cadr exp)   (caadr exp)))
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp) (cddr exp))))  ; formal params, body
(define (make-define var expr)
  (list 'define var expr))

(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters lambda-exp) (cadr lambda-exp))
(define (lambda-body lambda-exp) (cddr lambda-exp))
(define (make-lambda parms body) (cons 'lambda (cons parms body)))

(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp) (cadddr exp))
(define (make-if pred conseq alt) (list 'if pred conseq alt))

(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))
(define first-cond-clause car)
```

```
(define rest-cond-clauses cdr)
(define (make-cond seq) (cons 'cond seq))

(define (let? expr) (tagged-list? expr 'let))
(define (let-bound-variables expr) (map first (second expr)))
(define (let-values expr) (map second (second expr)))
(define (let-body expr) (cddr expr)) ;differs from lecture--body may be a sequence
(define (make-let bindings body)
  (cons 'let (cons bindings body)))

(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions begin-exp) (cdr begin-exp))
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))
(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
(define (make-begin exp) (cons 'begin exp))

(define (application? exp) (pair? exp))
(define (operator app) (car app))
(define (operands app) (cdr app))
(define (no-operands? args) (null? args))
(define (first-operand args) (car args))
(define (rest-operands args) (cdr args))
(define (make-application rator rands)
  (cons rator rands))

(define (and? expr) (tagged-list? expr 'and))
(define and-exprs cdr)
(define (make-and exprs) (cons 'and exprs))
(define (or? expr) (tagged-list? expr 'or))
(define or-exprs cdr)
(define (make-or exprs) (cons 'or exprs))

;;
;; this section is the actual implementation of meval
;;


(define (list-of-values exps env)
  (cond ((no-operands? exps) '())
        (else (cons (m-eval (first-operand exps) env)
                    (list-of-values (rest-operands exps) env)))))

(define (eval-if exp env)
  (if (actual-value (if-predicate exp) env)
      (m-eval (if-consequent exp) env)
      (m-eval (if-alternative exp) env)
      ))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (m-eval (first-exp exps) env))
        (else (m-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (m-eval (assignment-value exp) env)
                       env))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (m-eval (definition-value exp) env)
    env))
```

```scheme
(define (let->application expr)
  (let ((names (let-bound-variables expr))
        (values (let-values expr))
        (body (let-body expr)))
    (make-application (make-lambda names body)
                      values)))

(define (cond->if expr)
  (let ((clauses (cond-clauses expr)))
    (if (null? clauses)
        #f
        (if (eq? (car (first-cond-clause clauses)) 'else)
            (make-begin (cdr (first-cond-clause clauses)))
            (make-if (car (first-cond-clause clauses))
                     (make-begin (cdr (first-cond-clause clauses)))
                     (make-cond (rest-cond-clauses clauses)))))))

(define input-prompt ";;; M-Eval input:")
(define output-prompt ";;; M-Eval value:")

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (if (eq? input '**quit**)
        'meval-done
        ; GED -- technically, we don't need to force the actual value to be
        ;         computed: we could could just print out the thunk, but this
        ;         would be ugly.
        (let ((output (actual-value input the-global-environment)))
          (announce-output output-prompt)
          ; GED -- the following line was not changed.  Embedded thunks get printed out
          ;         using our internal representation.  If you're enterprising, think about
          ;         how you might modify the printer to hide that representation.  Hint 1:
          ;         play with the DrScheme implementation of streams to see the results of
          ;         it hiding the representation.  Hint 2: consider using a version of
          ;         tree-map or tree-fold.
          (display output)
          (driver-loop)))))

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))

(define (announce-output string)
  (newline) (display string) (newline))

(define *meval-warn-define* #t) ; print warnings?
(define *in-meval* #f)          ; evaluator running

;;
;;
;; implementation of meval environment model
;;

; double bubbles
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? proc)
  (tagged-list? proc 'procedure))
(define (procedure-parameters proc) (second proc))
(define (procedure-body proc) (third proc))
(define (procedure-environment proc) (fourth proc))


; bindings
(define (make-binding var val)
  (list var val))
```

```scheme
(define binding-variable car)
(define binding-value cadr)
(define (binding-search var frame)
  (if (null? frame)
      #f
      (if (eq? var (first (first frame)))
          (first frame)
          (binding-search var (rest frame)))))
(define (set-binding-value! binding val)
  (set-car! (cdr binding) val))

; frames
(define (make-frame variables values)
  (cons 'frame (map make-binding variables values)))
(define (frame-variables frame) (map binding-variable (cdr frame)))
(define (frame-values frame) (map binding-value (cdr frame)))
(define (add-binding-to-frame! var val frame)
  (set-cdr! frame (cons (make-binding var val) (cdr frame))))
(define (find-in-frame var frame)
  (binding-search var (cdr frame)))

; environments
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())

(define (find-in-environment var env)
  (if (eq? env the-empty-environment)
      #f
      (let* ((frame (first-frame env))
             (binding (find-in-frame var frame)))
        (if binding
            binding
            (find-in-environment var (enclosing-environment env))))))

; drop a frame
(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 args supplied" vars vals)
          (error "Too few args supplied" vars vals))))

; name rule
(define (lookup-variable-value var env)
  (let ((binding (find-in-environment var env)))
    (if binding
        (binding-value binding)
        (error "Unbound variable -- LOOKUP" var))))

(define (set-variable-value! var val env)
  (let ((binding (find-in-environment var env)))
    (if binding
        (set-binding-value! binding val)
        (error "Unbound variable -- SET" var))))

(define (define-variable! var val env)
  (let* ((frame (first-frame env))
         (binding (find-in-frame var frame)))
    (if binding
        (set-binding-value! binding val)
        (add-binding-to-frame! var val frame))))

; primitives procedures - hooks to underlying Scheme procs
(define (make-primitive-procedure implementation)
  (list 'primitive implementation))
(define (primitive-procedure? proc) (tagged-list? proc 'primitive))
```

```scheme
(define (primitive-implementation proc) (cadr proc))
(define (primitive-procedures)
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'set-car! set-car!)
        (list 'set-cdr! set-cdr!)
        (list 'null? null?)
        (list '+ +)
        (list '- -)
        (list '< <)
        (list '> >)
        (list '= =)
        (list 'display display)
        (list 'not not)
        ; GED -- newline is useful, so we add it...not a big deal
        (list 'newline newline)
        ; GED -- note that stream-car doesn't need to be a special form.
        ;        It actually could be a compound procedure, but it's a little
        ;        more convenient to make it a primitive.
        (list 'stream-car car)
        ; GED -- stream-cdr also doesn't need to be a special form, but it
        ;        cannot be a compound procedure.  In this version of the evaluator,
        ;        thunks are not first-class objects, so there's no way to use
        ;        real Scheme's force-it in a lambda expression for this interpreter.
        ;        By making stream-cdr a special form, m-apply automatically calls
        ;        force-it for us, so this primitive is really simple.
        (list 'stream-cdr cdr)
        ; ... more primitives
        ))

(define (primitive-procedure-names) (map car (primitive-procedures)))

(define (primitive-procedure-objects)
  (map make-primitive-procedure (map cadr (primitive-procedures))))

(define (apply-primitive-procedure proc args)
  (apply (primitive-implementation proc) args))

; used to initialize the environment
(define (setup-environment)
  (let ((initial-env (extend-environment (primitive-procedure-names)
                                         (primitive-procedure-objects)
                                         the-empty-environment))
        (oldwarn *meval-warn-define*))
    (set! *meval-warn-define* #f)
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    (set! *meval-warn-define* oldwarn)
    initial-env))

(define the-global-environment (setup-environment))

(define (refresh-global-environment)
  (set! the-global-environment (setup-environment))
  'done)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   Representing Thunks
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```scheme
; GED -- the thunk implementation was just grabbed from the lazy evaluator in the
;         online tutor.  Recall that this is a memoized implementation, where we
;         evaluate the thunk's expression at most one time.

(define (actual-value exp env)
  (force-it (m-eval exp env)))

(define (delay-it exp env) (list 'thunk exp env))

(define (thunk? obj) (tagged-list? obj 'thunk))
(define (thunk-exp thunk) (cadr thunk))
(define (thunk-env thunk) (caddr thunk))

(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value (thunk-exp obj)
                                     (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
        ((evaluated-thunk? obj) (thunk-value obj))
        (else obj)))

(define (evaluated-thunk? obj)
  (tagged-list? obj 'evaluated-thunk))

(define (thunk-value evaluated-thunk)
  (cadr evaluated-thunk))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   cons-stream expressions
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; GED -- this is new code to handle the cons-stream special form.  To keep things simple,
;         we'll use a real cons cell with a thunk in the cdr part to represent a stream
;         pair.  When combined with our implementation of m-apply and the stream-cdr
;         primitive, this does the right thing.

(define (cons-stream? exp) (tagged-list? exp 'cons-stream))
(define (cons-stream-car-part exp) (second exp))
(define (cons-stream-cdr-part exp) (third  exp))
(define (eval-cons-stream exp env)
  (cons (m-eval (cons-stream-car-part exp) env)
        (delay-it (cons-stream-cdr-part exp) env)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   testing (GED)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Helper procedure that evaluates a sequence of expressions using our
; stream evaluator in its global environment and returns the value of
; the final expression.
(define (evl expressions)
  (car (last-pair (map (lambda e (m-eval (car e) the-global-environment))
                       expressions))))

; Make a useful compound procedure for converting the first n elements of
; a stream into a list.  Note that print-stream does not need to be a primitive.
(evl '((define (print-stream s n)
         (if (= n 0)
             '()
```

```
                (cons (stream-car s) (print-stream (stream-cdr s) (- n 1)))))))))

; Adds two streams, element-wise.  Assumes s1 and s2 are infinite.
(evl '((define (add-streams s1 s2)
         (cons-stream (+ (stream-car s1)
                         (stream-car s2))
                      (add-streams (stream-cdr s1)
                                   (stream-cdr s2))))))

(display "testing␣ones-------------------------------------------") (newline)
(evl '((define ones (cons-stream 1 ones))
       (display ones) (newline)                 ; -> (1 #thunk)
       (display (stream-car ones)) (newline)    ; -> 1
       (display (stream-cdr ones)) (newline)))  ; -> (1 #evaluated-thunk)

(display "testing␣ints--------------------------------------------") (newline)
(evl '((define ints (add-streams ones (cons-stream 0 ints)))
       (display ints) (newline)                 ; -> (1 #thunk)
       (display (stream-cdr ints)) (newline)    ; -> (2 #thunk)
       (print-stream ints 10)))                 ; -> (1 2 3 4 5 6 7 8 9 10)
```