

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Spring Semester, 2004

**Quiz II**

**Closed Book – one sheet of notes, both sides**

Separately, we have distributed an answer sheet. You may use the space on the exam booklet for whatever temporary work you find useful, but you **MUST** enter your answers into the answer sheet. **Only the answer sheet will be graded.** Each problem that requires an answer has been numbered. Place your answer at the corresponding number in the answer sheet.

Note that any procedures or code fragments that you write will be judged not only on correct function, but also on clarity and good programming practice.

The answer sheet asks for your section number and tutor. For your reference, here is the table of section numbers.

Section	Time	Room	Instructor	Tutor
R01	WF 9 AM	36-155	Berthold Horn	Joshua Tauber
R02	WF 10 AM	34-301	Randy Davis	David Ziegler
R03	WF 10 AM	36-155	Berthold Horn	Sam Daitch
R04	WF 10 AM	34-304	Ben Vandiver	Iuliu Vasilescu
R05	WF 11 AM	34-301	Randy Davis	Murali Vajapeyam
R06	WF 2 PM	26-322	Ben Vandiver	Iuliu Vasilescu/Murali Vajapeyam
R07	WF 12 PM	34-301	Daniela Rus	Bryan Russell/Joshua Tauber
R08	WF 1 PM	34-303	Barbara Cutler	Manish Jethwa
R09	WF 1 PM	34-301	Daniela Rus	Bryan Russell
R10	WF 2 PM	34-303	Barbara Cutler	Manish Jethwa/Sam Daitch
R11	WF 11 AM	26-314	Michael Collins	Kristina Foster/David Ziegler
R12	WF 12 PM	26-314	Michael Collins	Kristina Foster
ESG				

**Part 1: (12 points)**

For each of the following expressions you are to (1) draw the box and pointer diagram corresponding to the list or pair structure created, and (2) write the printed result of evaluating the last expression in the sequence. If the evaluation results in an error, write “error” and describe the error.

```
(define x (cons 7 (list 8 9)))
(set-car! (cdr x) 10)
x
```

**Question 1. Draw the box and pointer diagram for x.**

**Question 2. Write the printed result for x.**

---

```
(define a (list 'big 'red 'dog))
(define b (cadr a))
(define c (cddr a))
(set! b 'blue)
(set-car! c 'cat)
a
```

**Question 3. Draw the box and pointer diagram for a, b and c.**

**Question 4. Write the printed result for a.**

---

```
(define y 7)
(define z (let ((x (list 'a (quote (b c)) y)))
            (set! y (cdr x))
            (set-cdr! x (car (cdr x)))
            x))
z
```

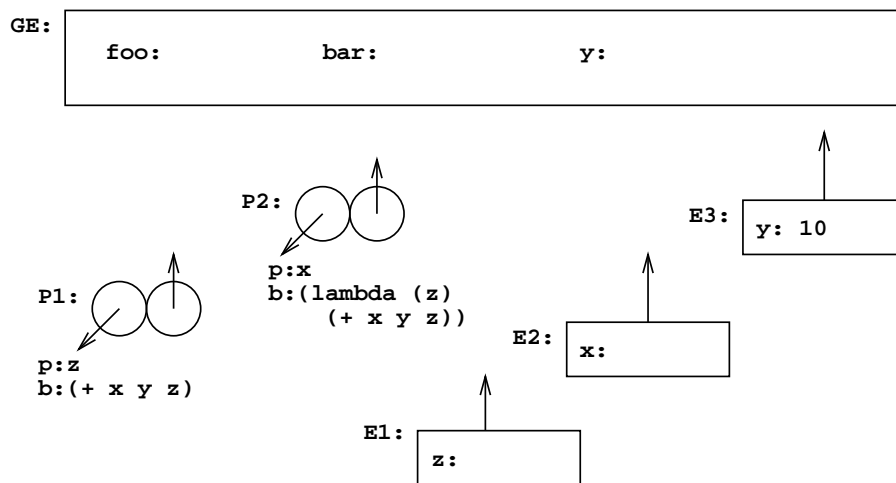
**Question 5. Draw the box and pointer diagram for x, y and z.**

**Question 6. Write the printed result for z.**

**Part 2: (19 points)**

The following expressions are evaluated in the order shown, generating the (incomplete) environment pictured below that. You may assume that primitive procedures such as `+`, `*`, and so on already exist (but are not shown) in the global environment (GE).

```
(define (foo x)
  (lambda (z)
    (+ x y z)))
(define bar (foo 20))
(define y 10)
(bar 30)
```



For each question below, write what the value is or pointer points to. Possible values include primitive or list-structured data such as `105`, `book`, or `(1 2 3)`, and procedures (indicate by writing `P1` for procedure `P1`, `P2` for procedure `P2`, or “other procedure” for a procedure not shown). Possible environment pointers include `E1`, `E2`, `E3`, `GE` for the global environment, or “other frame” for an environment frame that is not shown.

**Question 7.** What does the environment pointer of procedure `P1` point to?

**Question 8.** What does the environment pointer of procedure `P2` point to?

**Question 9.** What is the enclosing environment of frame `E1`?

**Question 10.** What is the enclosing environment of frame `E2`?

**Question 11.** What is the value of `foo` in the global environment?

**Question 12.** What is the value of `bar` in the global environment?

**Question 13.** What is the value of `y` in the global environment?

**Question 14.** What is the value of `z` in frame `E1`?

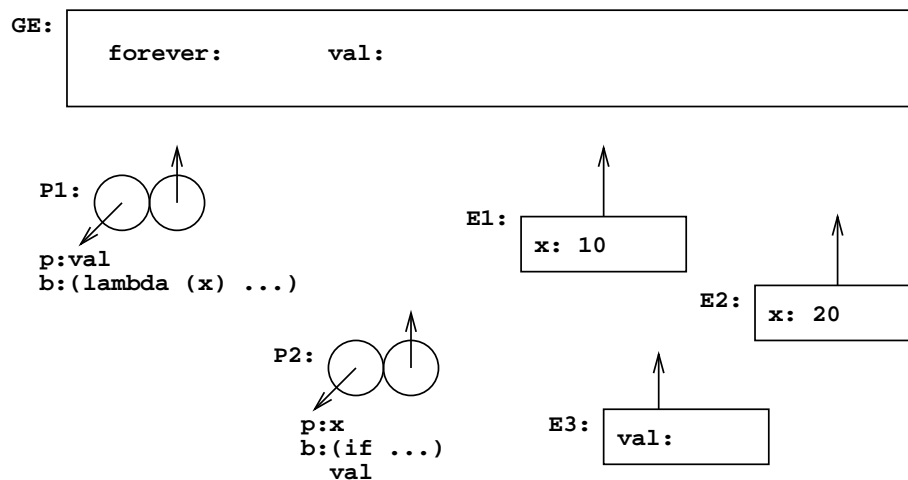
**Question 15.** What is the value of `x` in frame `E2`?

**Question 16.** What is the printed result from last expression above: `(bar 30)`?

**Part 2, cont'd.**

The following expressions are evaluated in the order shown, generating the (incomplete) environment pictured below that. You may assume that primitive procedures such as `+`, `*`, and so on already exist (but are not shown) in the global environment (GE).

```
(define val nil)
(define forever
  (let ((val nil))
    (lambda (x)
      (if (null? val) (set! val x)
          val))))
(forever 10)
(forever 20)
```



For each question below, write what the value is or pointer points to. Possible values include primitive or list-structured data such as `105`, `book`, or `(1 2 3)`, and procedures (indicate by writing P1 for procedure P1, P2 for procedure P2, or “other procedure” for a procedure not shown). Possible environment pointers include E1, E2, E3, GE for the global environment, or “other frame” for an environment frame that is not shown.

**Question 17.** What does the environment pointer of procedure P1 point to?

**Question 18.** What does the environment pointer of procedure P2 point to?

**Question 19.** What is the enclosing environment of frame E1?

**Question 20.** What is the enclosing environment of frame E2?

**Question 21.** What is the enclosing environment of frame E3?

**Question 22.** What is the value of `forever` in the global environment?

**Question 23.** What is the value of `val` in the global environment?

**Question 24.** What is the value of `val` in frame E1?

**Question 25.** What is the printed result from last expression above: `(forever 20)`?

**Part 3: (14 points)**

Alyssa P. Hacker makes an extension to the underlying implementation of our Scheme object oriented system. Our OO system as described in lecture 15 and used in the adventure game project is replicated (in slightly shortened form) at the end of this quiz booklet.

The change Alyssa wants to make is to give every `root-object` in the system (and thus all objects that inherit from `root-object`) its own “unique identifier” or UID. She modifies the class definition for `root-object` as shown below.

```
(define make-root-object
  (let ((next-uid 1))
    (lambda (self)
      (define uid next-uid)
      (set! next-uid (+ 1 next-uid))
      (lambda (message)
        (case message
          ((TYPE) (lambda () '(root)))
          ((IS-A) (lambda (type) (if (memq type (ask self 'TYPE)) #t #f)))
          ((UID) (lambda () uid))
          (else (no-method)))))))
```

Immediately after the above definition is evaluated, Alyssa tests her implementation with the following two expressions (evaluated in the order shown below):

```
(define x (create-instance make-root-object))
(define y (create-instance make-root-object))
(ask x 'uid) ;Value: 1
(ask y 'uid) ;Value: 2
```

Alyssa is happy with the results so far. However, she knows that the two test cases above are not enough. In an effort to be more complete, she defines the following simple class hierarchy, which she will use to further test her code.

```

(define (create-A) (create-instance make-A))
(define (make-A self)
  (let ((root (make-root-object self)))
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend 'A root)))
        (else (get-method message root))))))

(define (create-B) (create-instance make-B))
(define (make-B self)
  (let ((root (make-root-object self)))
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend 'B root)))
        (else (get-method message root))))))

(define (create-C) (create-instance make-C))
(define (make-C self)
  (let ((A-part (make-A self))
        (B-part (make-B self)))
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend 'C B-part A-part)))
        (else (get-method message B-part A-part))))))

```

Alyssa then evaluates the following expressions in sequence. You can assume that the following expressions are evaluated immediately after the prior `root-object` tests above, so that the internal value of the `next-uid` variable is 3 at the beginning of evaluation of the sequence of tests below:

```

(define a1 (create-A))
(define b1 (create-B))
(define c1 (create-C))

```

**Question 26.** What is the result of `(ask a1 'is-a 'A)?`

**Question 27.** What is the result of `(ask a1 'is-a 'B)?`

**Question 28.** What is the result of `(ask a1 'is-a 'C)?`

**Question 29.** What is the result of `(ask a1 'is-a 'root)?`

**Question 30.** What is the result of `(ask a1 'uid)?`

**Question 31.** How many times is `make-root-object` called as a result of the `(define a1 (create-A))` expression above?

**Question 32.** What is the result of `(ask c1 'is-a 'A)?`

**Question 33.** What is the result of `(ask c1 'is-a 'B)?`

**Question 34.** What is the result of `(ask c1 'is-a 'C)?`

**Question 35.** What is the result of `(ask c1 'is-a 'root)?`

**Question 36.** What is the result of `(ask c1 'uid)?`

**Question 37.** How many times is `make-root-object` called as a result of the `(define c1 (create-C))` expression above?

**Part 4: (14 points)**

Ben Bitdiddle has taken the implementation of the “better queue” discussed in detail in the lecture on tagged data structures, and converted it to use our object-oriented system. His implementation is shown below. Recall that the idea of a queue is that many items may be put on the queue; one can then get items from the queue, and items are retrieved in the same order in which they were put on the queue. The new methods include the predicate `EMPTY?` to see if the queue is empty, `PUT-ITEM` to add an item to the rear of the queue, and `GET-ITEM` to get (and remove from the queue) the item at the front of the queue. In addition, the `SHOW` method returns a list of the items currently on the queue.

```
;; better queue
;;
(define (create-queue)
  (create-instance make-queue))

(define (make-queue self)
  (let ((root-part (make-root-object self))
        (front-ptr nil)
        (rear-ptr nil))
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend 'queue root-part)))
        ((EMPTY?) (lambda () (null? front-ptr)))
        ((PUT-ITEM) (lambda (item)
                      (let ((new-pair (cons item nil)))
                        (cond ((ask self 'empty?)
                               (set! front-ptr new-pair)
                               (set! rear-ptr new-pair))
                              (else
                               (set-cdr! rear-ptr new-pair)
                               (set! rear-ptr new-pair)))
                          #t))))
        ((GET-ITEM) (lambda ()
                      (cond ((ask self 'empty?) #f)
                            (else (let ((item (car front-ptr)))
                                    (set! front-ptr (cdr front-ptr))
                                    item))))))
        ((PUT-FRONT) YOUR-CODE-HERE)
        ((SHOW) (lambda () front-ptr))
        (else (get-method message root-part))))))
```

An example using this implementation shows how the queue should behave:

```
(define q (create-queue))
(ask q 'put-item 'a)    ;Value: #t
(ask q 'put-item 'b)    ;Value: #t
(ask q 'get-item)      ;Value: a
(ask q 'put-item 'c)    ;Value: #t
(ask q 'show)          ;Value: (b c)
(ask q 'get-item)      ;Value: b
(ask q 'get-item)      ;Value: c
(ask q 'show)          ;Value: nil
(ask q 'get-item)      ;Value: #f
```

Your first task is to show that you understand the internal data structure used by the `queue` implementation above. Assume the following three expressions have been evaluated in sequence:

```
(define q2 (create-queue))
(ask q2 'put-item 'a)
(ask q2 'put-item 'b)
```

**Question 38:** Draw the box and pointer diagram for `front-ptr` and `rear-ptr` after the last expression above has been evaluated.

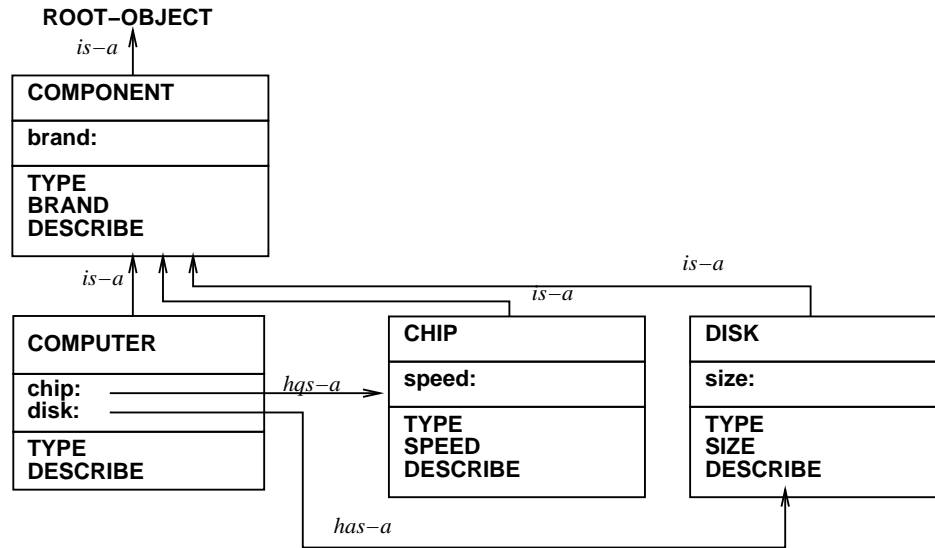
Ben has not written the implementation for the `PUT-FRONT` method. The specification for the behavior of this method is that it should put the argument item at the *front* of the queue, rather than at the *rear* of the queue as the normal `PUT-ITEM` method does. The `PUT-FRONT` method should return `#t` on success. An example illustrates the desired behavior:

```
(define q3 (create-queue))
(ask q3 'put-item 'a)    ;Value: #t
(ask q3 'put-front 'b)  ;Value: #t
(ask q3 'put-item 'c)    ;Value: #t
(ask q3 'show)          ;Value: (b a c)
```

**Question 39:** Write the method for the `PUT-FRONT` message.

**Part 5: (21 points)**

Alyssa P. Hacker has a new start-up company in her garage to bring in components such as chips and disks, and assemble them into computers. She has specified and implemented the class hierarchy shown pictorially below for `component`, `chip`, `disk`, and `computer`. The implementation uses our OO system as described in lecture 15 and used in the adventure game project, which is replicated (in slightly shortened form) at the end of this quiz booklet.



The code that Alyssa has written to implement the classes in this class diagram is given on the next several pages. We will work through this code class by class, asking you to evaluate what is returned by example uses of the code as we go. The purpose of this part of the quiz is for you to show that you can read, understand, and extend object oriented programs. As you do so, the class diagram on this page may be helpful.

```
;; component
;;
(define (create-component brand)
  ; symbol -> component
  (create-instance make-component brand))

(define (make-component self brand)
  (let ((root-part (make-root-object self)))
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend 'component root-part)))
        ((BRAND) (lambda () brand))
        ((DESCRIBE) (lambda () (list 'component (ask self 'brand))))
        (else (get-method message root-part))))))

; Easy way to get a description of a list of components:
(define (describe-all components)
  (map (lambda (x) (ask x 'DESCRIBE)) components))
```

**Question 40:** What is returned by the last expression in the following sequence:

```
(define x (create-component 'acme-power-supply))
(ask x 'describe)
```

**Question 41:** What is returned by the last expression in the following sequence:

```
(define y (list (create-component 'acme-power-supply)
                (create-component 'ronco-power-cord)))
(describe-all y)
```

```

;; chip
;;
;; A computer chip is an electronic component, with
;; some speed (in GHz).
;;
(define (create-chip brand speed)
  (create-instance make-chip brand speed))

(define (make-chip self brand speed)
  (let ((component-handler (make-component self brand)))
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend 'chip component-handler)))
        ((SPEED) (lambda () speed))
        ((DESCRIBE) (lambda () (list 'chip speed 'GHz)))
        (else (get-method message component-handler))))))

;; disk
;;
;; A disk is a component, with some disk size (in gigabytes)
;;
(define (create-disk brand size)
  (create-instance make-disk brand size))

(define (make-disk self brand size)
  (let ((component-handler (make-component self brand)))
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend 'disk component-handler)))
        ((SIZE) (lambda () size))
        ((DESCRIBE) (lambda () (list 'disk size 'Gb)))
        (else (get-method message component-handler))))))

```

**Question 42:** What is returned by the last expression in the following sequence:

```

(define chip1 (create-chip 'pentium 3.2))
(ask chip1 'describe)

```

**Question 43:** What is returned by the last expression in the following sequence:

```

(define disk1 (create-disk 'frisbee 80))
(ask disk1 'is-a 'component)

```

```

;; computer
;;
;; A computer is a component which must consist of
;; a chip and disk.
;;
(define (create-computer brand chip disk)
  (create-instance make-computer brand chip disk))

(define (make-computer self brand chip disk)
  (let ((component-handler (make-component self brand)))
    (if Q44-EXPRESSION-HERE
        (error "make-computer: wrong component types!")
        (lambda (message)
          (case message
            ((TYPE) (lambda () (type-extend 'computer component-handler)))
            ((DESCRIBE)
             (lambda ()
              (list 'computer (ask self 'brand) (ask chip 'describe) (ask disk 'describe))))
            (else (get-method message component-handler))))))))))

```

Alyssa wants an error to be generated whenever the input arguments are not of the correct type. For example:

```

(create-computer 'clunker (create-chip 'pentium 3.4) (create-chip 'athlon 3.6))
;make-computer: wrong component types!
;Type D to debug error, Q to quit back to REP loop:

```

**Question 44:** Implement this type checking, by writing the Q44-EXPRESSION-HERE that is missing in the make-computer procedure implementing the computer class.

**Question 45:** What is returned by the last expression in the following sequence:

```

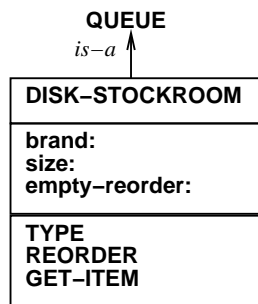
(define chip1 (create-chip 'pentium 3.2))
(define disk1 (create-disk 'frisbee 80))
(define computer1 (create-computer 'clunker chip1 disk1))
(ask computer1 'describe)

```

**Part 6: (20 points)**

This part of the quiz asks you to show that you can put together a bigger system out of smaller pieces. Specifically, you will use both `queues` from Part 4 of this quiz and `disks` from Part 5 of this quiz to implement something interesting to help out Alyssa P. Hacker's startup business. For this reason, it is best to do Part 6 of the quiz *after* you have done Parts 4 and 5. Note that Part 6 does not depend on your answers or solutions from Parts 4 and 5, but you will need to understand the classes and implementations in those parts.

Alyssa is building lots of computers – her startup company has really taken off. She keeps running out of disk drives to assemble into computers. Your job is to implement a `disk-stockroom` as pictured in the class diagram below. A `disk-stockroom` acts as a `queue` of `disk` components. The stockroom has a `REORDER` method that takes one argument, the number of disks to reorder. The `REORDER` method should internally use `create-disk` when it needs to build a new disk (of the brand and size specified when the stockroom was created). Alyssa can also use `GET-ITEM` to get a disk from the `disk-stockroom` when she needs one. If the `disk-stockroom` does not have a disk, rather than return `#f` as the usual `queue` would do, the `disk-stockroom` should `REORDER` a default number of disks (this default is the `empty-reorder` number specified when the `disk-stockroom` was created), and then return the requested `disk`.



The desired behavior of the `disk-stockroom` is illustrated in the example below:

```

(define (describe-all-queue q) (describe-all (ask q 'SHOW)))
(define supply (create-disk-stockroom 'frisbee 100 4))
(describe-all-queue supply)           ;Value: nil
(ask supply 'REORDER 1)                ;Value: #t
(describe-all-queue supply)           ;Value: ((disk 100 gb))
(ask supply 'EMPTY?)                  ;Value: #f
(ask (ask supply 'GET-ITEM) 'describe) ;Value: (disk 100 gb)
(ask supply 'EMPTY?)                  ;Value: #t
(ask (ask supply 'GET-ITEM) 'describe) ;Value: (disk 100 gb)
(ask supply 'EMPTY?)                  ;Value: #f
(describe-all-queue supply)           ;Value: ((disk 100 gb) (disk 100 gb) (disk 100 gb))
  
```

**Question 46: Write the code to implement the `disk-stockroom` class:**

```

(define (create-disk-stockroom brand size empty-reorder)
  (create-instance make-disk-stockroom brand size empty-reorder))

(define (make-disk-stockroom self brand size empty-reorder) ...)
  
```

**BACKGROUND INFORMATION — THIS PAGE CONTAINS NO QUESTIONS**

```

;;; Scheme Object-Oriented System

; make-instance
;
; instance is an object which holds the "self" of a normal object
; instance. It handles type requests, but otherwise passes all
; messages along to the handler procedure that it contains.
;
(define (make-instance)
  (let ((handler #f))
    (lambda (message)
      (case message
        ((SET-HANDLER!) (lambda (handler-proc) (set! handler handler-proc)))
        (else (get-method message handler))))))

(define (create-instance maker . args)
  (let* ((instance (make-instance))
        (handler (apply maker instance args)))
    (ask instance 'SET-HANDLER! handler)
    instance))

; root-object
;
; It contains the TYPE and IS-A methods. All classes should inherit
; (directly or indirectly) from root.
;
(define (make-root-object self)
  (lambda (message)
    (case message
      ((TYPE) (lambda () '(root)))
      ((IS-A) (lambda (type) (if (memq type (ask self 'TYPE)) #t #f)))
      (else (no-method))))))

; ask
;
; We "ask" an object to invoke a named method on some arguments.
;
(define (ask object message . args)
  (let ((method (get-method message object)))
    (cond ((method? method) (apply method args))
          (else (error "No method for" message)))))

```

**BACKGROUND INFORMATION — THIS PAGE CONTAINS NO QUESTIONS**

**BACKGROUND INFORMATION — THIS PAGE CONTAINS NO QUESTIONS**

```

; get-method
;
; Objects have methods to handle messages. Gets the indicated method
; from the object or objects. This procedure can take one or more
; objects as arguments, and will return the first method it finds
; based on the order of the objects.
;
(define (get-method message . objects)
  (define (try objects)
    (if (null? objects)
        (no-method)
        (let ((method ((car objects) message)))
          (if (not (eq? method (no-method)))
              method
              (try (cdr objects))))))
    (try objects))

(define (method? x)
  (cond ((procedure? x) #T)
        ((eq? x (no-method)) #F)
        (else (error "Object returned this non-message:" x))))

(define no-method
  (let ((tag (list 'NO-METHOD)))
    (lambda () tag)))

; type-extend
;
; Use this inside each make-<object> class definition,
; in the TYPE method for that class, in order to add
; to the type information for that new class.
;
(define (type-extend type . parents)
  (cons type
        (remove-duplicates
         (append-map (lambda (parent) (ask parent 'TYPE)) parents))))

(define (remove-duplicates lst)
  (if (null? lst)
      nil
      (cons (car lst) (remove-duplicates (filter (lambda (x) (not (eq? x (car lst)))) lst)))))

(define (filter predicate lst)
  (cond ((null? lst) '())
        ((predicate (car lst))
         (cons (car lst) (filter predicate (cdr lst))))
        (else (filter predicate (cdr lst)))))

```

**BACKGROUND INFORMATION — THIS PAGE CONTAINS NO QUESTIONS**