

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

**Recitation 15**  
**Message-Passing Objects**

## Scheme

### 1. Special Forms

- (a) *case* - (*case* *expr* *clauses*)  
 Works like *cond*, except the test of each clause is a list of numbers and symbols to compare against the value of *expr*.

### 2. Procedures

- (a) (*for-each* *proc* *list*)  
 Applies *proc* to each element of *list* **in order**, returning an unspecified value.
- (b) (*association-procedure* *pred* *select*)  
 Returns an association procedure that is similar to *assv*, except that *select* (a procedure of one argument) is used to select the key from the association, and *pred* (an equivalence predicate) is used to compare the key to the given item.

## Problems

A message-passing object definition:

```
(define (make-binary-operation name op)
  (lambda (message)
    (case message
      ((NAME)
       name)
      ((OPERATE)
       (lambda (a b)
         (op a b)))
      (else
       (error "binop can't" message))))))

(define binop (make-binary-operation 'glue (lambda (x y) (append x y))))
((binop 'OPERATE) '(1) '(2))
```

### 1. Stack object implementation

- (a) Complete the skeleton for the stack object given below. The skeleton comprises everything but the method definitions.

```
(define (make-stack)
  (let ((vals '())))
```

- (b) Add a method called `EMPTY?` which returns `#t` if the stack is empty.
- (c) Add a method called `CLEAR` which empties the stack of any elements it may contain.
- (d) Add a method called `PEEK` which returns the top element of the stack, leaving the stack unchanged. If the stack is empty, signal a “stack underflow” error.
- (e) Add a method called `PUSH` which allows an element to be added to the top of the stack.
- (f) Add a method called `POP` which removes and returns the top element of the stack. Remember to program defensively.
2. Write a procedure called `push-all` which takes a stack and a list and pushes all the elements of the list onto the stack. It should return the stack.

```
(define (push-all stack lst)
```

3. Write a procedure called `pop-all` which takes a stack and pops elements off it until it becomes empty, adding each element to the output list.

```
(define (pop-all stack)
```

4. Write reverse.

```
(define (reverse lst)
```

### 5. Calculator object implementation

```
(define (make-calculator) ; an RPN calculator
  (let ((stack (make-stack))
        (ops (list (make-binary-operation '+ +)
                    (make-binary-operation '- -)
                    (make-binary-operation '* *)
                    (make-binary-operation '/ /))))
    (oplookup
     (lambda (message)
       (case message
         (else (error "calculator doesn't" message))))))

  (define c (make-calculator))

  (c 'ANSWER) ; empty-stack
  ((c 'NUMBER-INPUT) 4) ; pushed
  (c 'ANSWER) ; 4
  ((c 'NUMBER-INPUT) 5) ; pushed
  (c 'ANSWER) ; 5
  ((c 'OPERATION-INPUT) '+) ; pushed
  (c 'ANSWER) ; 9
  ((c 'NUMBER-INPUT) 7) ; pushed
  ((c 'OPERATION-INPUT) '-') ; pushed
  (c 'ANSWER) ; 2
  (c 'CLEAR) ; cleared
  (c 'ANSWER) ; empty-stack
```

- (a) Complete the definition of `oplookup` so it is a procedure that when given an operation name and the `ops` list, will return the operation with the given name.

- (b) Write a method called `ANSWER`, which returns the current value on the top of the stack.
- (c) Write a method called `CLEAR`, which removes all the numbers from the stack.
- (d) Write a method called `NUMBER-INPUT`, which puts the number onto the stack.
- (e) Write a method called `OPERATION-INPUT`, which takes an operation name as input, looks up the operation, removes two numbers from the stack, and puts the result of the operation back onto the stack.