MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring 2006

Recitation 17 — 4/14/2006
Message Passing Objects

Special Forms

1. *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* with `eqv?`.

Procedures

1. `(for-each` *proc list*`)`
   Similar to `map`: Applies *proc* to each element of *list*. However, `for-each` works for its side effects, and applies the procedure **in order** from first element to last, and does not specify a return value.
2. `(apply` *proc args*`)`
   Applies *proc* to *args*. It's like having written `(proc arg0 arg1 arg2 ...)`.

Variable number of arguments

1. `.` *args* - In order to implement variable-number-of-arguments procedures (like `+`,`list`,`append`, or `map`). End the parameter list of a `lambda` with `. args`. The variable `args` will be bound to a list of all the remaining arguments.
   Examples:

```
((lambda (x . args) (append x args)) '(1 2) 3 4 5)
(define (do-stuff x y . rest) ...)
(define (add . args) (fold-right + 0 args))
(define add (lambda args (fold-right + 0 args)))
((lambda args (cons 'yay args)) 3 4)
((lambda args (cons 'yay args)))
```

Message Passing Pairs

Consider the following possible definitions for cons, car and cdr.

```
(define (cons x y)
  (lambda (msg . rest)
    (case msg
      ((PAIR?) #t)
      ((CAR) x)
      ((CDR) y)
      ;;set-car!, set-cdr! go here
      (else (error "pair cannot" msg)))))
```

```
(define (car p)
  (p 'CAR))
(define (cdr p)
  (p 'CDR))
(define (pair? p)
  (and (procedure? p) (p 'PAIR?)))
```

1. Complete the definition for cons to handle `set-car!` and `set-cdr!`, and write new definitions for `set-car!` and `set-cdr!`.

   The addition to `cons` is:

   ```
   ((SET-CAR!) (set! x (first rest)))
   ((SET-CDR!) (set! y (first rest)))
   ```

   The definitions for `set-car!` and `set-cdr!` are:

   ```
   (define (set-car! p nval)
     (p 'SET-CAR! nval))
   (define (set-cdr! p nval)
     (p 'SET-CDR! nval))
   ```

Message Passing Bank Account

```
(define (make-bank-account)
  (let ((balance 0))
    (lambda (msg . rest)
      (case msg
        ((BALANCE) balance)
        ((DEPOSIT)
         (set! balance (+ balance (car rest)))
         balance)
        ((WITHDRAW)
         (let ((v (car rest)))
           (if (>= balance v)
               (set! balance (- balance v))
               (error "cannot withdraw" v))
           balance))
        (else
         (error "no action for message" msg))
        ))))

(define checking (make-bank-account))
(define savings (make-bank-account))

(checking 'DEPOSIT 10)
(savings 'DEPOSIT 100)
(checking 'WITHDRAW 2.50)
```

1. Draw an environment diagram corresponding to the final state of global environment after evaluating the above expressions. You do not need to include procedures and frames that are no longer accessible such as procedure objects created by a `let` that are no longer accessible, and frames from procedure applications that are inaccessible.

2. Add additional methods to the bank account.

   `ACCRUE-INTEREST`, which takes an interest rate, computes the amount of interest and deposits the correct additional amount.

   `TRANSFER`, which takes an amount, and another account to transfer to, and adjusts the balances in the two accounts accordingly.

   `EVIL-ATM-WITHDRAW`, which takes an amount to withdraw, ensures that it is a multiple of $20, and withdraws that amount along with a $2 service charge. Make sure the account does the right thing when the balance is enough for the withdrawal but not the withdrawal plus fee.