
CS61A**Midterm 2 Review****(v1.1)**

Basic Info

Your login:

Your section number:

Your TA's name:

Midterm 2 is going to be held on Tuesday 7-9p, at 1 Pimentel.

What will Scheme print?

What will the Scheme interpreter print in response to each of the following expressions? If the return value is a pair, then draw the corresponding box and pointer diagram.

1. `((caddr '(1 + * 2)) 3 4)`
2. `(list? (cons 1 (cons 2 (cons nil nil))))`
3. `(list (list (list (cons 1 nil))))`
4. `(list 'one (cons 'foo (list '())))`
5. `(filter number? '(1 (a 2) ((b) 3) c 4 d))`
6. `(cddadr '(a (((b)) c d e f) g))`
7. `(map (lambda (x) (cons 'foo x)) '(1 (2) (3)))`

Lists

1. Write a procedure, (**flatten** *ls*) that takes in a deep list and flattens all the elements into a flat list. For example,
`(flatten '(1 (2 (3 4 ((5 6) 7) 8) 9) 10)) ==> (1 2 3 4 5 6 7 8 9 10)`

2. Write a predicate procedure **deep-car?** that takes a symbol and a deep-list (possibly including sublists to any depth) as its arguments. It should return true if and only if the symbol is the car of the list or of some list that's an element, or an element of an element, etc.

```
> (deep-car? \ 'a \ (a b c))
#t
> (deep-car? \ a \ '(b (c a) a d))
#f
> (deep-car? \ a \ '((x y) (z (a b) c) d))
#t
```

Fill in the blanks to complete the definition.

```
(define (deep-car? symbol lst)
  (if (pair? lst)
      (or (eq? symbol
                (car lst))
          (helper symbol (cdr lst)))
      #f))

(define (helper symbol lsts)
  (cond ((null? lsts) #f)
        ((deep-car? symbol (car lsts)) #t)
        (else #f)))
```

Trees

1. Write a procedure, `(tree-member? x tree)` that takes in an element and a general tree, and returns `#t` if `x` is part of tree, and `#f` otherwise.
2. Write `double-datum?`, which takes a tree as its argument, and returns true if there exists a node anywhere in the tree that has the same datum as one of its children (immediate children), otherwise false.

Scheme-1

Here is code for `scheme-1`

```
(define (eval-1 exp)
  (cond ((constant? exp) exp)
        ((symbol? exp) (eval exp))
        ((quote-exp? exp) (cadr exp))
        ((if-exp? exp)
         (if (eval-1 (cadr exp))
             (eval-1 (caddr exp))
             (eval-1 (cadddr exp))))
        ((lambda-exp? exp) exp)
        ((pair? exp)
         (apply-1 (eval-1 (car exp))
                   (map eval-1 (cdr exp))))
        (else (error "bad expr: " exp))))

(define (apply-1 proc args)
  (cond ((procedure? proc)
         (apply proc args))
        ((lambda-exp? proc)
         (eval-1 (substitute (caddr proc) ; the body
                              (cadr proc) ; the formal parameters
                              args ; the actual arguments
                              ()))) ; bound-vars
        (else (error "bad proc: " proc))))
```

1. Suppose we moved the moved the `cond` clause beginning with `pair?` (the one that handles function calls) to the beginning of the `cond`, making it the first clause:

```
(define (eval-1 exp)
  (cond ((pair? exp) ...)
        ((constant? exp) ...)
        ((symbol? exp) ...)
        ...
        (else (error ...))))
```

With the above changes, show what Scheme-1 would print given the following inputs. If the result is an error, just write error.

Scheme-1: 3

Scheme-1: +

Scheme-1: (if #t 1 2)

Scheme-1: ((lambda (x) (* x x)) 3)

Scheme-1: (+ 3 4)

Scheme-1: (lambda (x) (* x x))

2. Suppose we changed it so that we no longer called `substitute` inside of `apply-1`:

```
(define (apply-1 proc args)
  (cond ((procedure? proc)
        (apply proc args))
        ((lambda-exp? proc)
         (eval-1 (caddr proc))) ; evaluate the body without using substitute
        (else (error "bad proc: " proc))))
```

With the above changes, show what Scheme-1 would print given the following inputs. If the result is an error, just write error.

Scheme-1: ((lambda (x) (* x x)) 3)

Scheme-1: ((lambda (x) (+ 2 3)) 2)

Scheme-1: (+ 3 4)

Scheme-1: (lambda (x) (* x x))

Scheme-1: (lambda (x) (+ 3 4))

Scheme-1: ((lambda (cons) (cons 1 2)) +)

Data Directed Programming and Message Passing

1. Instead of attaching one tag to an object, suppose we wanted to attach multiple tags. After all, a tomato is not only a food, its also a projectile! So well rename `attach-tag` to `attach-tags` (taking a list of tags as its first argument), and rename `type-tag` to `type-tags`. Here is the original version of `operate`:

```
(define (operate op obj)
  (let ((proc (get (type-tag obj) op)))
    (if proc
        (proc (contents obj))
        (error "No such operator for this type"))))
```

Rewrite `operate` so that it looks for each of an objects type tags in the table, using the first one for which a procedure is found for this operator.

2. Below are definitions of `m-car` and `m-cdr` (the `m` here stands for message-passing).

```
(define (m-car pair) (pair 'car))
(define (m-cdr pair) (pair 'cdr))
```

- (a) Write a definition of `m-cons` that behaves just like regular cons when used in conjunction with `m-car` and `m-cdr`.

- (b) If you type `list1` at the STk prompt, what would it print out?

```
(define list1 (m-cons 1 (m-cons 2 (m-cons 3 '()))))
```

- (c) We would like to be able to convert a message-passing list into an ordinary Scheme list. Write the function `m-list-to-regular-list` that takes a message-passing list (hmm, I havent yet defined what a message-passing list is; how should it be defined?) and returns a regular list of the same elements.