Lambda

(lambda (x) (* x 2))

((lambda (y) (* (+ y 2) 8)) 10)

96

((lambda (b) (* 10 ((lambda (c) (* c b)) b))) ((lambda (e) (+ e 5)) 5))

1000

((lambda (a) (a 3)) (lambda (z) (* z z)))

9

((lambda (n) (+ n 10))

  ((lambda (m) (m ((lambda (p) (* p 5)) 7))) (lambda (q) (+ q q))))

80

((lambda (x) (x x)) (lambda (y) 4))

4

((lambda (y z) (z y)) (* (lambda (a) (a 3 5))))

15

Foo

Write a procedure, foo, that, given the call below, will evaluate to 10.

((foo foo foo) foo 10)

(define (foo x y) y)

Bar

Write a procedure, bar, that given the call below, will evaluate to 10 as well.

(bar (bar (bar 10 bar) bar) bar)

(define (bar x y) x)

First-Satisfies

Something easy: write first-satisfies that takes in a predicate procedure and a sentence, and returns the first element that satisfies the predicate test. Return false if none satisfies the predicate test. For example, (first-satisfies even? '(1 2 3 4 5)) returns 2, and (first-satisfies number? ' (a clockwork orange)) returns false.

(define (first-satisfies pred? sent)
  (cond ((empty? sent) #f)
        ((pred? (first sent)) (first sent))
        (else (first-satisfies pred? (bf sent))))))

Hide and Go Seek

Were going to play hide-and-go-seek. Lets say, a seeker is a procedure that takes in a sentence, and seeks out a certain word in the sentence. It returns the word if the word is found, or false otherwise. For example, if we have a 4-seeker, a seeker that seeks out the number 4, then

(4-seeker '(1 2 3 4 5)) ==> 4

(4-seeker '(1 2 3)) ==> #f

A seeker-producer is a procedure that takes in an element x and returns a procedure (a seeker) that takes in a sentence sent and returns x if the element x is in the sentence sent, and false otherwise.

a. Make a call to seeker-producer to find out if 4 is in the list (9 3 5 4 1 0). seeker-producer is the only procedure you can use! What does it return?
b. Implement seeker-producer, using the handy-dandy procedure member?.

(define (seeker-producer x)
  (lambda(sent) (if (member? x sent) x #f)))

c. Implement seeker-producer, using an internal define, but not using member?.

(define (seeker-producer x)
  (define (helper sent)
    (cond ((empty? sent) #f)
          ((equal? x (first sent)) x)
          (else (helper (bf sent))))
    helper)

d. Implement seeker-producer, not using internal defines or member?.

(define (seeker-producer x)
  (lambda(sent)
    (cond ((empty? sent) #f)
          ((equal? x (first sent)) x)
          (else ((seeker-producer x) (bf sent))))))

e. Of course, its not much of a game if we cant hide! A hider of a word is a procedure that takes in a sentence and hides the word behind an asterisk if it exists. For example, if we have a 4-hider, a hider that hides the number 4, then

(4-hider '(1 2 3 4 5)) ==> (1 2 3 4*5)

Write a procedure hider-producer that takes in an element y, and returns a procedure (a hider) that takes in a sentence sent and returns the same sentence with element y hidden behind an asterisk, if it exists.
Youll probably want to use every to help you.

(define (hider-producer x)
  (lambda(sent)
    (every (lambda(w)
             (if (equal? w x) (word * w) w)
             sent))))

f. Oh no! Now a hider can fool your seeker! Consider this call:

(4-seeker (4-hider '(1 2 3 4 5)))
==> #f (make sure you know why!)

Surely you will not be outdone by yourself. Write a procedure, super-seeker-producer that takes in a procedure produced by seeker-producer (that is, a seeker), and returns a super-seeker that will not be fooled by hider:

((super-seeker-producer 4-seeker) (4-hider '(1 2 3 4 5))) ==> 4

You can use every if you want. You might also find these procedures useful:

(define (hidden? w) (equal? (first w) *))
(define (unhide w)
  (if (hidden? w) (bf w) w))

(define (super-seeker-producer seeker)
  (lambda(sent) (seeker (every unhide sent)))))