LAMBDAS AND HIGHER ORDER FUNCTIONS 2

COMPUTER SCIENCE 61AS

The Basics of Lambdas

1. What does a lambda expression always return?

   Solution: A procedure.

2. Express the following expressions using lambda instead of their named counterparts.

   a. `square`

   Solution: `(lambda (x) (* x x))`

   b. `(square 4)`

   Solution: `( (lambda (x) (* x x)) 4)` Notice there are two parenthesis at the beginning! The inner one belongs to the expression for `square`, while the outer one calls `square`.

   c. `sum-of-squares`

   Solution: `(lambda (x y) (+ (* x x) (* y y)))`

   d. `(sum-of-squares 3 (+ 2 2))`

   Solution: `( (lambda (x y) (+ (* x x) (* y y))) 3 (+ 2 2))`

What Will Racket Print?

What do the following expressions evaluate to?

1. `(lambda (x) (* x 2))`

   Solution: `#[closure arglist=(x) 237a08]`

2. `((lambda (x) (* x 2)) 10)`

   Solution: 20

3. ```
   ((lambda (b) (* 10 ((lambda (c) (* c b)) b)))
    ((lambda (e) (+ e 5)) 5))
   ```

   Solution: 1000

4. `((lambda (x) (x x)) (lambda (y) 4))`

   Solution: 4

5. `((lambda (x y) (y x)) * (lambda (a) (a 3 5)))`

   Solution: 15

6. ```
   ((lambda (n) (+ n 10))
    ((lambda (m) (m ((lambda (p) (* p 5)) 7)))
     (lambda (q) (+ q q))))
   ```

   Solution: 80

Practice with Lambdas

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

   `((foo foo foo) foo 10)`

   Solution: `(define (foo x y) y)`

2. Write a procedure, `bar`, that given the call below, will evaluate to 10.

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

Solution: `(define (bar x y) x)`

3. What does the following evaluate to? (This one is hard!)

```
((lambda (f x) (f f x))
 (lambda (k n)
     (if (< n 2)
         1
         (* n (k k (- n 1)))))
 4)
```

Solution: This one is really tricky! This call will return 24, the factorial of 4. Note that the procedure is recursive – it calls itself, without using a define statement! Such things are called Y-combiners; but don't worry about it for now.

The Basics of Higher Order Functions

1. What is a higher-order function? What are some examples you've seen so far?

Solution: A higher-order function either takes in a function as an argument, outputs a function, or both! Some HOFs you've seen so far are `keep` and `every`.

2. Recall the procedure `keep`, which takes in a predicate procedure and a sentence, and throws away all words of the sentence that don't satisfy the predicate.

   Explain why `(keep (< 6) '(4 5 6 7 8))` doesn't work. Then, re-write the expression so it works (use a lambda!).

Solution: `(< 6)` evaluates to `#t`; it's not a function! `keep` expects its first argument to be a function. This can be fixed in multiple ways. Two such ways are below:

```
(keep (lambda (x) (< x 6)) '(4 5 6 7 8))

OR

(define (lessthan x)          ;; Here's another higher-order function!
  (lambda (n) (< n x)))

(keep (lessthan 6) '(4 5 6 7 8))
```

Practice with Higher Order Functions

1. Write `accumulate`. `Accumulate` takes in a combiner function, an initial value, and a sentence.

```
(accumulate +  0 '(1 2 3 4))
10

(accumulate *  1 '(1 2 3 4))
24

(accumulate word 'while '(my guitar gently weeps))
whilemyguitargentlyweeps
```

Solution:

```
(define (accumulate fn initial sent)
    (if (empty? sent)
        initial
        (fn (accumulate fn initial (butlast sent)) (last sent))))

OR

(define (accumulate fn initial sent)
    (if (empty? sent)
        initial
        (fn initial (accumulate fn (first sent) (bf sent)))))
```

2. Write a procedure `f-expt`, (f-expt func power) that returns a procedure which is equivalent to `func` applied `power` times. Assume `func` takes in only a single argument. For example, `((f-expt 1+ 3) 2)` is 5, because `(1+ (1+ (1+ 2)))` is 5.

Solution: There are a few tricky parts about this problem. First, the base case isn't that obvious. Remember, we need to return a function. What function corresponds to a `power` of 0? It's the identity function! This can be written `(lambda (x) x)`. This is simply a function that has the same input as output.

The next tricky part is the recursive call. Intuitively, you might guess that we want to call `f-expt` with the same `func` and `(- power 1)`. You might also guess that we need some way to combine the result of different recursive calls. Combining all of this together, we get the code below.

```
(define (f-expt func power)
    (if (= power 0)
        (lambda (x) x)     ;; Identity function
        (lambda (x) (func ((f-expt func (- power 1)) x))) ))
```

3. Write a procedure `curry`. `Curry` takes in a function (that takes in two arguments) and a value. It returns a function that takes in one argument.

```
((curry sum-of-squares 3) 4)
25

((curry sum-of-squares 3) 9)
100
```

Solution:

```
(define (curry fn x)
    (lambda (y) (fn x y)))
```

4. We're going to play hide-and-go-seek. Let's 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 `#f` 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 `#f` otherwise.

a. Make a call to `seeker-producer` to find out if 4 is in the sentence `'(9 3 5 4 1 0)`. `seeker-producer` is the only procedure you can use! What does it return?

Solution: `((seeker-producer 4) '(9 3 5 4 1 0))`. This should return 4.

b. Implement seeker-producer, without using internal defines or `member?`. (Hint: think lambdas and recursion!)

```
(define (seeker-producer x)
```

Solution:

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

OR

(define (seeker-producer x)
    (lambda (sent)
        (if (empty? (keep (lambda (y) (equal? x y)) sent))
            #f
            x)))
```

5. Of course, it's not much of a game if we can't 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.

   You'll probably want to use `every` to help you.

   ```
   (define (hider-producer x)
   ```

Solution:

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