The Basics of Lambdas

1. What does a lambda expression always return?

2. Express the following expressions using lambda instead of their named counterparts.
   a. square
   b. (square 4)
   c. sum-of-squares
   d. (sum-of-squares 3 (+ 2 2))

What Will Racket Print?

What do the following expressions evaluate to?
   1. (lambda (x) (* x 2))
2. \(((\text{lambda } (x) (* x 2)) \text{ 10})\)

3. \(((\text{lambda } (b) (* 10 ((\text{lambda } (c) (* c b)) b)))
   ((\text{lambda } (e) (+ e 5)) 5))\)

4. \(((\text{lambda } (x) (x x)) \text{ (lambda } (y) 4))\)

5. \(((\text{lambda } (x y) (y x)) * (\text{lambda } (a) (a 3 5)))\)

6. \(((\text{lambda } (n) (+ n 10))
   ((\text{lambda } (m) (m ((\text{lambda } (p) (* p 5)) 7)))
   (\text{lambda } (q) (+ q q)))\)

---

**Practice with Lambdas**

1. Write a procedure, foo, that given the call below, will evaluate to 10.
   \(((\text{foo foo foo) foo 10})\)

2. Write a procedure, bar, that given the call below, will evaluate to 10.
   \((\text{bar (bar (bar 10 bar) bar) bar})\)

3. What does the following evaluate to? (This one is hard!)
   \(((\text{lambda } (f x) (f f x))
   (\text{lambda } (k n)\))\)
The Basics of Higher Order Functions

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

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!).

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

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

\[
((\texttt{curry sum-of-squares 3}) 4) \quad 25
\]

\[
((\texttt{curry sum-of-squares 3}) 9) \quad 100
\]

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 \( \text{sent} \) and returns \( x \) if the element \( x \) is in the sentence \( \text{sent} \), and \#f otherwise.

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

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

\[
(\text{define (seeker-producer x)}
\]

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 \text{hider-producer} that takes in an element \( y \), and returns a procedure (a hider) that takes in a sentence \( \text{sent} \) and returns the same sentence with element \( y \) hidden behind an asterisk, if it exists.

You’ll probably want to use \text{every} to help you.

\[
(\text{define (hider-producer x)}
\]