Review: Lambda

\[ ((\text{lambda } (x_1 \ldots x_n) \text{ exp}) \; v_1 \ldots v_n) \Rightarrow \text{ exp'} \]

where \( \text{exp'} \) is \( \text{exp} \) with all occurrences of \( x_1 \) replaced by \( v_1 \), all occurrences of \( x_2 \) replaced by \( v_2 \), and so on.

For example, the next step here would be:

\[ ((\text{lambda } (x \; y) \; (\times \; (\oplus \; y \; 4) \; x)) \; 5 \; 6) \Rightarrow (\times \; (\oplus \; 6 \; 4) \; 5) \]
Group Problem - Stepping with Lambda

Step through the following program:

```
(((lambda (x y) (lambda (x) (* x y))) 5 6) 10)
```
Warm-up Problems - Abstract List Functions

Recall the abstract list functions \texttt{filter}, \texttt{map}, and \texttt{build-list}. Use abstract list functions to do the following the tasks without explicit recursion:

Double each element in \texttt{(list 1 2 3 4 5)}:
⇒ \texttt{(list 2 4 6 8 10)}

Keep all the elements in \texttt{(list 1 2 3 4 5 6 7)} that are divisible by 3:
⇒ \texttt{(list 3 6)}

Create a list of odd numbers from 1 to 12:
⇒ \texttt{(list 1 3 5 7 9 11)}
Group problem - factors

Write a function called factors that consumes a positive integer, n, and produces a list of its factors from 1 to n inclusive. Do not use explicit recursion.

(factors 30) ⇒ (list 1 2 3 5 6 10 15 30)
Review: foldr

Recall that foldr is the built-in function that abstracts recursion on lists, where the first element of the list is combined with the recursive call on the rest.

Here is an implementation of foldr as my-foldr:

```scheme
(define (my-foldr combine base lst)
  (cond
    [(empty? lst) base]
    [else (combine (first lst)
                   (my-foldr combine base (rest lst)))
         (my-foldr combine base (rest lst)))]))
```

Sum the numbers in (list 1 2 3 4):

```scheme
(my-foldr + 0 (list 1 2 3 4)) ⇒ 10
```
Warm-up Problems - foldr

Do the following tasks without explicit recursion. The only abstract list function you may use is foldr:

Find the minimum element of (list 6 1 2 9 2):
⇒ 1

Count the number of even numbers in (list 1 2 3 4 5 6 7):
⇒ 3

Recall:

- The combine function passed to foldr must take 2 arguments:
  - The first one corresponds to the first element of the list
  - The second corresponds to the recursive result from the rest of the list
- The base argument passed to foldr corresponds to the result from the empty list
Group Problem - Stepping with foldr

Step through the following program:

```
(foldr (lambda (x y) (cond [(even? x) (cons x y)]
                          [else y])) empty '(1 1 2 3 5 8))
```
Assignment Review: ifoldr

In Assignment 08, you will write your own abstract list function, ifoldr, that abstracts lockstep recursion on a list and a natural number. Recursing on the natural number follows a count-up pattern, so ifoldr is equivalent to recursing on a list while keeping track of the index of the current element.

The combine function will take 3 arguments instead of 2, where the extra argument corresponds to the position of the current element in the list.
Assignment Review: ifoldr

Here is an example:

Keep the first 3 elements of (list 'a 'b 'c 'd 'e):

(ifoldr (lambda (pos frst rror)
            (cond [(< pos 3) (cons frst rror)]
                  [else rror]))
    empty (list 'a 'b 'c 'd 'e))

⇒ (list 'a 'b 'c)
Assignment Review: remove-letters

In the assignment, you have been provided an implementation for remove-letters, which uses another provided function remove-at. remove-at consumes a Nat called i and a list, and removes the element at index i from the list. It uses ifoldr.

remove-letters consumes a string, s, and produces a list of strings, each with one letter removed from s.

We will re-write these functions together. Here are some examples using remove-letters:

(remove-letters "abc") ⇒ (list "bc" "ac" "ab")

(remove-letters " ") ⇒ empty
Assignment Review: remove-letters

(define (remove-at i lst)
  (ifoldr (lambda (pos frst rror)
              (cond [(= i pos) rror]
                    [else (cons frst rror)]))
         empty lst))

(define (remove-letters s)
  (local [(define loc (string->list s))]
         (build-list (length loc)
                     (lambda (i) (list->string (remove-at i loc))))))
Group Problem: separate

Write a function `separate` that consumes a value called `sep` and a list. It inserts `sep` between identical items in the list.

You may not use explicit recursion.

\[
\begin{align*}
(\text{separate } '(a b c) \ 'x) & \Rightarrow '(a b c) \\
(\text{separate } '(a b b c) \ 'x) & \Rightarrow '(a b x b c) \\
(\text{separate } '(a a b b c c) \ 'x) & \Rightarrow '(a x a b x b c x c)
\end{align*}
\]