Review: Lambda

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

where \( \exp' \) is \( \exp \) 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:

\[ ((\text{lambda} \ (x \ y) \ (\ast \ (+ \ y \ 4) \ x)) \ 5 \ 6) \Rightarrow (\ast \ (+ \ 6 \ 4) \ 5) \]

Group Problem - Stepping with Lambda

Step through the following program:

\[ (((\text{lambda} \ (x \ y) \ (\text{lambda} \ (x) \ (\ast \ x \ y))) \ 5 \ 6) \ 10) \]
Warm-up Problems - Abstract List Functions
Recall the abstract list functions filter, map, and build-list. Use abstract list functions to do the following the tasks without explicit recursion:

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

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

Create a list of odd numbers from 1 to 12:
⇒ (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:
(define (my-foldr combine base lst)
  (cond
   [(empty? lst) base]
   [else (combine (first lst)
                 (my-foldr combine base (rest lst)))]
)

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

```
(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))))))
```

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

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.

(separate '(a b c) 'x) ⇒ '(a b c)
(separate '(a b b c) 'x) ⇒ '(a b x b c)
(separate '(a a b b c c) 'x) ⇒ '(a x a b x b c x c)