Review: Lambda

\[(\text{local } [(\text{define (name-used-once } x_1 \ldots x_n \text{) } \text{exp})])\]

\[\text{name-used-once}\]

can also be written

\[(\text{lambda} (x_1 \ldots x_n \text{) exp})\]

\[\text{lambda}\] can be thought of as “make-function”.

It can be used to create a function which we can then use as a value – for example, as the value of the first argument to filter, map, or foldr.

Warm-up Problems: Abstract List Functions

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

Double each element in \(\text{(list 1 2 3 4 5)}\):

\[\Rightarrow (\text{list 2 4 6 8 10})\]

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

\[\Rightarrow (\text{list 3 6})\]

Create a list of odd numbers from 1 to 12:

\[\Rightarrow (\text{list 1 3 5 7 9 11})\]
Group Problem: increasing-lists
Without using explicit recursion, write a function called increasing-lists that consumes a positive integer n and produces a list of n lists of natural numbers, where the ith list contains the first i + 1 natural numbers.

(increasing-lists 1) ⇒ '(0)
(increasing-lists 4) ⇒
'(0
(0 1)
(0 1 2)
(0 1 2 3))

Group Problem: map-lofn
Write a function map-lofn which consumes a (listof Any) and a list of functions. The functions in the consumed list will have the contract Num → Any. map-lofn produces a list of lists, where each sublist contains the result of applying each function from the consumed list to each number in the consumed (listof Any). You may not use explicit recursion in your solution.

(check-expect (map-lofn (list 3.5 'four 18 "q" 0)
                        (list sqr add1 zero?))
                (list (list 12.25 324 0)
                      (list 4.5 19 1)
                      (list false false true)))

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 given 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 given 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
  (list 1 1 2 3 5 8))

**Group Problem: list-replace**

Recall the function list-replace from Tutorial 4. The function list-replace consumes a target value, a replacement value, and a (listof Any). The function produces a new list, which is identical to the consumed list, except all occurrences of the target value (if any) are replaced with the replacement value. Write the function list-replace without using explicit recursion, and without using foldr.

(list-replace 5 "five" (cons 5 (cons 'a (cons 5 empty))))
⇒ (cons "five" (cons 'a (cons "five" empty)))

(list-replace "five" 5 (cons 5 (cons 'a empty)))
⇒ (cons 5 (cons 'a empty))
Group Problem: list-replace
Now rewrite the function `list-replace` without using explicit recursion, and using `foldr`. You may not use `map`, `filter`, or `build-list`.

```
(list-replace 5 "five" (cons 5 (cons 'a (cons 5 empty))))
⇒ (cons "five" (cons 'a (cons "five" empty)))
```

```
(list-replace "five" 5 (cons 5 (cons 'a empty)))
⇒ (cons 5 (cons 'a empty))
```