COMP 333 Spring 2022
Racket Programming Project: List Functions

Write a Racket program to define the following functions

(rotate-list-left x)
x: list
value: list
description
       move all elements of the list one element to the left, head wraps around to tail
hints
       if list is empty or one element, value is the original list
       else append the cdr of the list to the list consisting of the car

(rotate-list-left-n x n)
x: list
n: integer
value: list
       call rotate-list-left n times
hint
       if n is 0, value is x
       else call the function recursively, replace x with (rotate-list-left x), n with n – 1.

(count-list-elements x)
x: list
value: integer
description
       return number of elements in list
       there is a built-in function to do this, but we're defining our own for practice
hint
       if list is empty, value is 0
       else 1 + recursive call on the cdr of x

(list-element-n x n)
x: list
n: integer
value: list element
description
　　　find the nth element in the list, elements are numbered 0 through length – 1.
trace
　　　(list-element-n '(a b c d e) 2)　　　　　'c
hint
　　　If n is 0, value is car of x
　　　Else call function recursively, replace x with cdr x, replace n with n - 1

(list-minus-element-n x n)
x: list
n: integer
value: list
description
　　　list with the nth element removed
hint
　　　if n is 0, value is cdr of x
　　　else cons car x to value of recursive function call, replace x with cdr x, n with n - 1
trace
　　　(list-minus-element-n '(a b c d e) 2)　　　'(a b d e)

(rotate-list-right x)
x: list
value: list
description
        similar to rotate-list-left but right rotation
hint
        find element n-1, cons to list-minus-element n-1

(reverse-list x)
x: list
value: list
description
        list with elements in reverse order
hint
        if list is empty or singleton, value is x
        else append recursive call on cdr of x to list of car of x

(cons-to-all a x)
a: element
x: list of lists
value: list of lists
description
        use cons to append a to the head of each element of x
        note that each element of x is a list
        use map to apply operation to each element of a list

Save this one for last, a little harder than the others

(permute x)
x: list
value: list of lists
description
        generate all permutations of the original list
        value is the list of lists of all permutations
hint
        (permute '())   '()
        (permute '(a))   '((a))
        (permute '(a b))   '((a b) (b a))
        (permute '(a b c))   '((a b c) (a c b) (b a c) (b c a) (c a b) (c b a))

        This case can be broken down into three parts
            (cons-to-all 'a (permute '(b c)))
            (cons-to-all 'b (permute '(a c)))
            (cons-to-all 'c (permute '(a b)))
        and appending all three lists together
        This case is traced for list of length 3, but this is the general case
hint
        If list has 1 element, return list of original list version of that one item
        If list has 2 elements, return list of original list and reverse of original list
        Else general case which calls a helper function defined below

Helper Functions for Permute Function

The general task for generating permutations breaks down into a couple of simpler tasks which can be implemented with helper functions ph-1 (permute helper #1) and ph-2 (permute helper #2). For example, in computing permutations for a list of length 3
　　　(a b c)
we can break the list down into 3 substeps

　　　a + (b c)　　　　　　element 0 + list with element 0 removed
　　　b + (a c)　　　　　　element 1 + list with element 1 removed
　　　c + (a b)　　　　　　element 2 + list with element 2 removed

Here, for each position in the list, we separate the element at position n from the remainder of the list (list with element at n removed). We already wrote functions list-element-n and list-minus-element-n that find these values from the original list and value of n. Next we generate permutations of the remainder list. This is done with a recursive call to the original permute function.

　　　a + ((b c) (c b))　　　a + permutations of (b c)
　　　b + ((a c) (c a))　　　b + permutations of (a c)
　　　c + ((a b) (b a))　　　c + permutations of (a b)

At this stage, note that we make a recursive call to **permute** but working on a subset of the original list. Also note once again that the value of calling permute for a list produces a "list of lists". In other words, each permutation is a list, and the collection of all permutations is a list of those lists. Next, we cons the removed element back onto each permutation in the list using cons-to-all.

　　　((a b c) (a c b))
　　　((b a c) (b c a))
　　　((c a b) (c b a))

This also produces lists of lists. Finally, we merge or append all the lists together to get the final list of permutations in one big list of lists.

　　　((a b c) (a c b) (b a c) (b c a) (c a b) (c b a))

This trace show how it works for a list of length 3. It works the same way for lists of any length n. But it will be hard to trace the function in detail for longer lists. Using this trace, now take a look at the helper functions. Note that the original **permute** function and **ph-1** are mutually recursive (permute calls ph-1 and ph-1 calls permute). This makes the two functions harder to trace, so you will need to test them carefully on small examples to begin with.

(ph-1 x n)

        find nth element of x

        find x minus nth element

        cons-to-all nth element to value of permute x minus nth element

(ph-2 x n)

        evaluate ph-1 for all positions in the list 0 through length – 1

        append all resulting lists into final list

The general case for the permute function is then to call ph-2

- permute calls ph-2
- ph-2 calls ph-1
- ph-1 calls permute (on a smaller version of the original list)

For testing, it will be difficult to test all three functions in their complete form. However you can write a temporary version of permute that works for lists up to length 3 or 4, then test the function with ph-1 and ph-2 in that limited case. Once this first version is working, you can introduce the more general definition and test it for longer lists.

Test Cases For All Functions

As you develop your functions, try them out on your own test cases. But in order to simplify my grading and evaluation of your program, I will test your code against the following test cases. Confirm that your code runs against these test cases before submitting.

```
> (rotate-list-left '())
'()
> (rotate-list-left '(a))
'(a)
> (rotate-list-left '(a b c))
'(b c a)
>

> (rotate-list-left-n '(a b c) 0)
'(a b c)
> (rotate-list-left-n '(a b c d e) 2)
'(c d e a b)
> (rotate-list-left-n '(a b c d e) 5)
'(a b c d e)
>

> (count-list-elements '())
0
> (count-list-elements '(a))
1
> (count-list-elements '(a b c d e))
5
>

> (list-element-n '(a b c d e) 0)
'a
> (list-element-n '(a b c d e) 4)
'e
> (list-element-n '(a b c d e) 1)
'b
>

> (list-minus-element-n '(a b c d e) 0)
'(b c d e)
> (list-minus-element-n '(a b c d e) 1)
'(a c d e)
> (list-minus-element-n '(a b c d e) 2)
'(a b d e)
```

```
> (list-minus-element-n '(a b c d e) 4)
'(a b c d)
>


> (rotate-list-right '(a b c d e))
'(e a b c d)
> (rotate-list-right '(a))
'(a)
> (rotate-list-right '(a b))
'(b a)
> (rotate-list-right '(a b c d e f g))
'(g a b c d e f)
>

> (reverse-list '(a))
'(a)
> (reverse-list '(a b))
'(b a)
> (reverse-list '(a b c d e))
'(e d c b a)
>

> (cons-to-all 'a '((b c) (d e) (f g)))
'((a b c) (a d e) (a f g))
>
```

```
> (permute '(a b))
'((a b) (b a))                                        2! = 2 permutations
> (permute '(a b c))
'((c a b) (c b a) (b a c) (b c a) (a b c) (a c b))    3! = 6 permutations
> (permute '(a b c d))
'((d c a b)  (d c b a)  (d b a c)  (d b c a)  (d a b c)    4! = 24 permutations
  (d a c b)  (c d a b)  (c d b a)  (c b a d)  (c b d a)
  (c a b d)  (c a d b)  (b d a c)  (b d c a)  (b c a d)
  (b c d a)  (b a c d)  (b a d c)  (a d b c)  (a d c b)
  (a c b d)  (a c d b)  (a b c d)  (a b d c))
>
```

Here are all test cases organized into one test suite, to make it easier for you to add it to your program once you're ready to test. You can also test individual functions separately as you complete them.

```
(rotate-list-left '())                ; '()
(rotate-list-left '(a))               ; '(a)
(rotate-list-left '(a b c))           ; '(b c a)
(rotate-list-left-n '(a b c) 0)       ; '(a b c)
(rotate-list-left-n '(a b c d e) 2)   ; '(c d e a b)
(rotate-list-left-n '(a b c d e) 5)   ; '(a b c d e)
(count-list-elements '())             ; 0
(count-list-elements '(a))            ; 1
(count-list-elements '(a b c d e))    ; 5
(list-element-n '(a b c d e) 0)       ; 'a
(list-element-n '(a b c d e) 4)       ; 'e
(list-element-n '(a b c d e) 1)       ; 'b
(list-minus-element-n '(a b c d e) 0) ; (b c d e)
(list-minus-element-n '(a b c d e) 1) ; '(a c d e)
(list-minus-element-n '(a b c d e) 2) ; '(a b d e)
(list-minus-element-n '(a b c d e) 4) ; '(a b c d)
(rotate-list-right '(a b c d e))      ; '(e a b c d)
(rotate-list-right '(a))              ; '(a)
(rotate-list-right '(a b))            ; '(b a)
(rotate-list-right '(a b c d e f g))  ; '(g a b c d e f)
(reverse-list '(a))                   ; '(a)
(reverse-list '(a b))                 ; '(b a)
(reverse-list '(a b c d e))           ; '(e d c b a)
(cons-to-all 'a '((b c) (d e) (f g))) ; '((a b c) (a d e) (a f g))

(permute '(a b))                      ; '((a b) (b a)) 2! = 2 permutations
(permute '(a b c))                    ; '((c a b) (c b a) (b a c) (b c a) (a b c) (a c b)) 3! = 6 permutations
(permute '(a b c d))

; '((d c a b) (d c b a) (d b a c) (d b c a) (d a b c)      4! = 24 permutations
; (d a c b) (c d a b) (c d b a) (c b a d) (c b d a)
; (c a b d) (c a d b) (b d a c) (b d c a) (b c a d)
; (b c d a) (b a c d) (b a d c) (a d b c) (a d c b)
; (a c b d) (a c d b) (a b c d) (a b d c))
```