Recursive List Functions in Racket

Because Racket lists are defined recursively, it’s natural to process them recursively.

Typically (but not always) a recursive function recf on a list argument L has two cases:

- **base case:** what does recf return when L is empty? (Use null? to test for an empty list.)
- **recursive case:** if L is the nonempty list (cons Vfirst Vrest) how are Vfirst and (recf Vrest) combined to give the result for (recf L)?

Note that we always apply recf directly to Vrest (and nothing else)!

Recursive List Functions: Divide/Conquer/Glue (DCG) strategy for the general case [in words]

**Step 1 (concrete example):** pick a concrete input list, typically 3 or 4 elements long. What should the function return on this input?

E.g. A `sum` function that returns the sum of all the numbers in a list:

```
(sum '(5 7 2 4)) ⇒* 18
```

**Step 2 (divide):** without even thinking, **always** apply the function to the rest of the list. What does it return?

```
(sum '(7 2 4)) ⇒* 13
```

**Step 3 (glue):** How to combine the first element of the list (in this case, 5) with the result from processing the rest (in this case, 13) to give the result for processing the whole list (in this case, 18)?

```
(+ 5 (sum '(7 2 4))) ⇒* 18
```

**Step 4 (generalize):** Express the general case in terms of an arbitrary input:

```
(define (sum nums)
  ...
  (+ (first nums) (sum (rest nums))) ...)
```

Recursive List Functions: Divide/Conquer/Glue (DCG) strategy for the general case [in diagram]

```
(sum '(5 7 2 4)) ⇒* 18
```

**Divide:** what should function return for rest of list? (wishful thinking!)

```
(sum '(7 2 4)) ⇒* 13
```

**Glue:** how to combine the first element of the list with the result of recursively processing rest of the list to get the desired result for the whole list?

**Solution for concrete example:**

```
(+ 5 (sum '(7 2 4)))
```

**Generalization of concrete solution:**

```
(+ (first nums) (sum (rest nums)))
```
In this case, $V_{null}$ should be 0, which is the identity element for addition.

But in general it depends on the details of the particular combiner determined from the general case. So solve the general case before the base case!

$$
\text{Recursive List Functions: base case via singleton case}
$$

Deciding what a recursive list function should return for the empty list is not always obvious and can be tricky. E.g. what should $(\text{sum } '(()))$ return?

If the answer isn’t obvious, consider the ”penultimate case” in the recursion, which involves a list of one element:

$$(\text{sum } '((4))) \Rightarrow 4$$

In this case, $V_{null}$ should be 0, which is the identity element for addition.

But in general it depends on the details of the particular combiner determined from the general case. So solve the general case before the base case!

In $(\text{sum (list 7 2 4))},$ the list argument to $\text{sum}$ is $(\text{cons 7 (cons 2 (cons 4 null)))})$.

Replace $\text{cons}$ by $+$ and $\text{null}$ by 0 and simplify:

$$(+\ 7\ (+\ 2\ (+\ 4\ 0)))$$

$$\Rightarrow\ (+\ 7\ (+\ 2\ 4))$$

$$\Rightarrow\ (+\ 7\ 6)$$

$$\Rightarrow\ 13$$
Generalizing the sum definition

(define (recf ns)
  (if (null? ns)
      nullval
      (combine (first ns)
                (recf (rest ns)))))

Your turn

Define the following recursive list functions and test them in Racket:

(product ns) returns the product of the numbers in ns

(min-list ns) returns the minimum of the numbers in ns
  Hint: use min and +inf.0 (positive infinity)

(max-list ns) returns the minimum of the numbers in ns
  Hint: use max and -inf.0 (negative infinity)

(all-true? bs) returns #t if all the elements in bs are truthy; otherwise returns #f. Hint: use and

(some-true? bs) returns a truthy value if at least one element in bs is truthy; otherwise returns #f. Hint: use or

(my-length xs) returns the length of the list xs
### Recursive Accumulation Pattern Summary

<table>
<thead>
<tr>
<th></th>
<th>combine</th>
<th>nullval</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>product</td>
<td></td>
<td></td>
</tr>
<tr>
<td>min-list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>max-list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>all-true?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>some-true?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>my-length</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Define these using Divide/Conquer/Glue

\[ \text{(snoc 11 '}(7\ 2\ 4)) \]
\[ '}(7\ 2\ 4\ 11) \]

\[ \text{(my-append '}(7\ 2\ 4) '}(5\ 8)) \]
\[ '}(7\ 2\ 4\ 5\ 8) \]

\[ \text{(append-all '}(7\ 2\ 4) (9) (5\ 8)) \]
\[ '}(7\ 2\ 4\ 9\ 5\ 8) \]

\[ \text{(my-reverse '}(5\ 7\ 2)) \]
\[ '(4\ 2\ 7) \]

### Mapping Example: map-double

(map-double ns) returns a new list the same length as ns in which each element is the double of the corresponding element in ns.

\[ > \text{(map-double (list 7 2 4))} \]
\[ '(14 4 8) \]

(define (map-double ns)
    (if (null? ns)
        ; Flesh out base case
        ; Flesh out general case
        )
)

### Understanding map-double

We’ll call this the mapping pattern

\[ \begin{array}{c}
    \text{7} \\
    \downarrow 2 \\
    \text{14} \\
    \end{array} \quad \begin{array}{c}
    \text{2} \\
    \downarrow 2 \\
    \text{4} \\
    \end{array} \quad \begin{array}{c}
    \text{4} \\
    \downarrow 2 \\
    \text{8} \\
    \end{array} \]

\[ \text{We’ll call this the mapping pattern} \]
Generalizing \texttt{map-double}

\[
\text{mapF} \left( \text{list } V_1 \ V_2 \ldots \ V_n \right)
\]

Expressing \texttt{mapF} as an accumulation

\[
\begin{align*}
\text{(define (mapF xs))} \\
\quad \text{(if (null? xs)} \\
\qquad \text{null} \\
\qquad \text{((λ (fst subres)} \\
\qquad \quad \text{((λ (fst subres)} \text{)} \text{; Flesh this out} \\
\qquad \quad \text{(first xs)} \\
\qquad \text{(mapF (rest xs))))))}
\end{align*}
\]

Some Recursive Listfuns Need Extra Args

\[
\begin{align*}
\text{(define (map-scale factor ns))} \\
\quad \text{(if (null? ns)} \\
\qquad \text{null} \\
\qquad \text{(cons (* factor (first ns))} \\
\qquad \quad \text{(mapF (rest ns))))}
\end{align*}
\]

Filtering Example: \texttt{filter-positive}

\[
\begin{align*}
\text{(filter-positive ns) returns a new list that contains only the} \\
\text{positive elements in the list of numbers \textit{ns}, in the same relative order} \\
\text{as in \textit{ns}.} \\
\quad \text{> (filter-positive (list 7 -2 -4 8 5))} \\
\quad \text{'(7 8 5)}
\end{align*}
\]

\[
\begin{align*}
\text{(define (filter-positive ns))} \\
\quad \text{(if (null? ns)} \\
\qquad \text{; Flesh out base case} \\
\qquad \text{; Flesh out recursive case} \\
\quad \text{)}
\end{align*}
\]
Understanding filter-positive

(filter-positive (list 7 -2 -4 8 5))

We'll call this the filtering pattern

Generalizing filter-positive

(filterP (list V1 V2 ... Vn))

Expressing filterP as an accumulation

(define (filterP xs)
  (if (null? xs)
      null
      ((lambda (fst subres)
          
          ) ; Flesh this out
        (first xs)
        (filterP (rest xs)))))