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 "blindly" apply `recf` to `Vrest`!

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]
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 (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, Vnull 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!

Putting it all together: base & general cases

\[
\text{(sum nums)} \text{ returns the sum of the numbers in the list nums}
\]

\[
\text{(define (sum ns)}
\]

\[
\text{(if (null? ns)}
\]

\[
\text{0)
}\]

\[
\text{(+ (first ns)}
\]

\[
\text{(sum (rest ns))))}
\]

Understanding \text{sum}: Approach #1

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

We’ll call this the recursive accumulation pattern

Understanding \text{sum}: Approach #2

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 null by 0 and simplify:

\[
(+ 7 (+ 2 (+ 4 0)))
\]

\[
⇒ (+ 7 (+ 2 4))
\]

\[
⇒ (+ 7 6)
\]

\[
⇒ 13
\]
Generalizing sum: Approach #1

(recf (list 7 2 4))

7 \rightarrow 2 \rightarrow 4 \rightarrow \bullet

Generalizing sum: Approach #2

In (recf (list 7 2 4)), the list argument to recf is

(cons 7 (cons 2 (cons 4 null)))

Replace cons by combine and null by nullval and simplify:

(combine 7 (combine 2 (combine 4 nullval))))

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>0</td>
</tr>
<tr>
<td>product</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>min-list</td>
<td>min</td>
<td>+inf.0</td>
</tr>
<tr>
<td>max-list</td>
<td>max</td>
<td>-inf.0</td>
</tr>
<tr>
<td>all-true?</td>
<td>and</td>
<td>#t</td>
</tr>
<tr>
<td>some-true?</td>
<td>or</td>
<td>#f</td>
</tr>
<tr>
<td>my-length</td>
<td>( \lambda (\text{fst subres}) (+ 1 \text{subres}) )</td>
<td>0</td>
</tr>
</tbody>
</table>

Define these using Divide/Conquer/Glue

> (snoc 11 '(7 2 4)) '(7 2 4 11)
> (my-append '(7 2 4) '(5 8)) '(7 2 4 5 8)
> (append-all '((7 2 4) (9) () (5 8))) '(7 2 4 9 5 8)
> (my-reverse '(5 7 2 4)) '(4 2 7 5)

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.

> (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

(map-double '(7 2 4))

We’ll call this the mapping pattern
Generalizing map\-double

(map\text{F} (\text{list } V_1 \ V_2 \ldots \ V_n))

Expressing \text{map}\text{F} as an accumulation

\begin{verbatim}
(define (map\text{F} xs)
  (if (null? xs)
      null
      (cons (\text{F} (first xs))
            (map\text{F} (rest xs))))
\end{verbatim}

Some Recursive Listfuns Need Extra Args

(define (map-scale \text{factor} ns)
  (if (null? ns)
      null
      (cons (* \text{factor} (first ns))
            (map-scale \text{factor} (rest ns)))))

Filtering Example: \text{filter-positive}

(filter-positive ns) returns a new list that contains only the positive elements in the list of numbers ns, in the same relative order as in ns.

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

\begin{verbatim}
(define (filter-positive ns)
  (if (null? ns)
      ; Flesh out base case
      ; Flesh out recursive case
      ))
\end{verbatim}
**Understanding filter-positive**

\[(\text{filter-positive} (\text{list} 7 -2 -4 8 5))\]

We’ll call this the filtering pattern

**Generalizing filter-positive**

\[(\text{filterP} (\text{list} V_1 V_2 \ldots V_n))\]

**(define (filterP xs))
(if (null? xs)
  null
  (if (P (first xs))
    (cons (first xs) (filterP (rest xs)))
    (filterP (rest xs))))**

**Expressing filterP as an accumulation**

**(define (filterP xs))
(if (null? xs)
  null
  ((lambda (fst subres)

  ) ; Flesh this out
  (first xs)
  (filterP (rest xs)))))**