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]

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

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))`
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!

\[
\text{(define (sum ns)}
\begin{aligned}
\text{(if (null? ns) )} & \text{0} \\
\text{(+) } & \text{(first ns) (sum (rest ns))})
\end{aligned}
\]

### Putting it all together: base & general cases

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

\[
\text{(define (sum ns)}
\begin{aligned}
\text{(if (null? ns) )} & \text{0} \\
\text{(+) } & \text{(first ns) (sum (rest ns))})
\end{aligned}
\]

### Understanding \text{sum}: Approach #1

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

\[
\text{We’ll call this the \text{recursive accumulation} pattern}
\]

### Understanding \text{sum}: Approach #2

\[
\text{In (sum (list 7 2 4)), the list argument to sum is}
\]

\[
\text{(cons 7 (cons 2 (cons 4 null)))}
\]

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

\[
\text{⇒ (+ 7 (+ 2 (+ 4 0)))}
\]

\[
\text{⇒ (+ 7 (+ 2 4))}
\]

\[
\text{⇒ (+ 7 6)}
\]

\[
\text{⇒ 13}
\]
Generalizing sum: Approach #1

\[
\text{(recf (list 7 2 4))}
\]

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:

\[
\text{(combine 7 (combine 2 (combine 4 nullval)))}
\]

Generalizing the sum definition

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

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

\[
\text{nullval}
\]

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

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

Your turn

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

- \text{(product ns)} returns the product of the numbers in \text{ns}
  \- \text{Hint: use min and +inf.0 (positive infinity)}

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

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

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

- \text{(my-length xs)} returns the length of the list \text{xs}
Recursive Accumulation Pattern Summary  Solutions

<table>
<thead>
<tr>
<th></th>
<th>combine</th>
<th>nullval</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>(\lambda (\text{fst subres}) (+ \text{fst subres}))</td>
<td>0</td>
</tr>
<tr>
<td>Note: below we show only simpler form, if it exists</td>
<td></td>
<td></td>
</tr>
<tr>
<td>product</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>min-list</td>
<td>min</td>
<td>+\text{inf}.0</td>
</tr>
<tr>
<td>max-list</td>
<td>max</td>
<td>-\text{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  Solutions

\[
\text{snoc \ 11 \ '(7 2 4)} \quad \text{snoc \ y \ xs}
\]
\[
\text{'(7 2 4 11)} \quad \text{(list y)} \quad \text{(cons \ (first \ xs) \ (snoc \ y \ (rest \ xs))}))
\]

\[
\text{(my-append \ (7 2 4) \ '(5 8))} \quad \text{(define \ (my-append \ xs \ ys)}
\]
\[
\text{'(7 2 4 5 8)} \quad \text{y) \quad (if \ (null? \ xs)}
\]
\[
\text{ys \quad (cons \ (first \ xs)}
\]
\[
\text{\quad (my-append \ (rest \ xs) \ ys)))})}
\]

\[
\text{(append-all \ xss \ xss \ means \ list} \quad \text{define \ (append-all \ xss)}
\]
\[
\text{if \ (null? \ xss) \ \text{; of \ list \ of \ elts}}
\]
\[
\text{'}() \quad \text{(my-append \ (first \ xss)}
\]
\[
\text{\quad (append-all \ (rest \ xss))})})}
\]

\[
\text{(my-reverse \ (5 7 2 4)} \quad \text{(define \ (my-reverse \ xs)}
\]
\[
\text{'(4 2 7 5)} \quad \text{if \ (null? \ xs)}
\]
\[
\text{') \quad (snoc \ (first \ xs)}
\]
\[
\text{\quad (my-reverse \ (rest \ xs)))}
\]

Mapping Example: map-double  Solutions

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

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

\[
\text{(define \ (map-double \ ns)}
\]
\[
\text{\text{if \ (null? \ ns)} \ (if \ (null? \ ns)}
\]
\[
\text{\quad '()} \quad \text{Can also write \text{null} \ or \text{ns}}
\]
\[
\text{\quad ; \text{Flesh out base case}}
\]
\[
\text{\quad (cons \ (* \ 2 \ \text{(first} \ \text{ns)}))}
\]
\[
\text{\quad (map-double \ (rest \ \text{ns}))} \quad \text{; \text{Flesh out general case}}
\]

\[
\text{))}
\]

Understanding map-double

\text{(map-double \ '(7 \ 2 \ 4))}

We’ll call this the mapping pattern
Generalizing \( \text{map-double} \)

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

\[
\begin{array}{c}
V_1 \\
\downarrow F \\
(F \ v_1) \\
\downarrow \\
\end{array} \quad \begin{array}{c}
V_2 \\
\downarrow F \\
(F \ v_2) \\
\downarrow \\
\end{array} \quad \ldots \quad \begin{array}{c}
V_n \\
\downarrow F \\
(F \ v_n) \\
\downarrow \\
\end{array} \\
\]

```
(define (map F xs)
  (if (null? xs)
      null
      (cons (F (first xs))
            (mapF (rest xs)))))
```

List Recursion 17

Expressing \( \text{mapF} \) as an accumulation

Solutions

```
(define (mapF xs)
  (if (null? xs)
      null
      ((λ (fst subres)
         (cons (F fst) subres) ) ; Flesh this out
       (first xs)
       (mapF (rest xs)))))
```

List Recursion 18

Some Recursive Listfuns Need Extra Args

```
(define (map-scale factor ns)
  (if (null? ns)
      null
      (cons (* factor (first ns))
            (map-scale factor (rest ns)))))
```

List Recursion 19

Filtering Example: \( \text{filter-positive} \)

Solutions

```
(define (filter-positive ns)
  (if (null? ns)
      ; Flesh out base case
      '()
    ; Can also write null or ns
    ; Flesh out recursive case
    (if (> (first ns) 0)
        (cons (first ns)
              (filter-positive (rest ns)))
        (filter-positive (rest ns))))
```

List Recursion 20
Understanding \texttt{filter-positive}

\begin{align*}
\texttt{(filter-positive (list 7 -2 -4 8 5))}
\end{align*}

We’ll call this the \textit{filtering} pattern

\begin{align*}
7 & \rightarrow -2 & \rightarrow -4 & \rightarrow 8 & \rightarrow 5 & \rightarrow \bullet \\
\text{\textgreater} 0 & \downarrow \texttt{#t} & \text{\textgreater} 0 & \downarrow \texttt{#f} & \text{\textgreater} 0 & \downarrow \texttt{#f} & \text{\textgreater} 0 & \downarrow \texttt{#t} & \text{\textgreater} 0 & \downarrow \texttt{#t} & \bullet
\end{align*}

\text{List Recursion 21}

\begin{align*}
\text{Generalizing \texttt{filter-positive}}
\end{align*}

\begin{align*}
\texttt{(filterP (list V1 V2 ... Vn))}
\end{align*}

\begin{align*}
\text{List Recursion 22}
\end{align*}

Expressing \texttt{filterP} as an accumulation \textbf{Solutions}

\begin{align*}
\text{(define (filterP xs)}
\text{ (if (null? xs)}
\text{ null)
\text{ (if (P (first xs)}
\text{ (cons (first xs) (filterP (rest xs)))
\text{ (filterP (rest xs))))))}
\end{align*}

\text{List Recursion 23}