05.4 List Idioms

Bookmark this page

There are an unlimited number of ways that we might wish to process the information in a list. However, after working with lists in a lot of practical contexts, we start to see some common themes emerge: standard styles of processing that come up again and again. We'll refer to these as list processing **idioms**.

In this module we'll continue to practice writing list functions, with an emphasis on exploring the three most important list idioms: **folding** a list, **mapping** a list, and **filtering** a list.

Folding a list

Folding a list (sometimes also called "reducing a list") is any process where we consume a list and boil all of its contents down to a single value (which is usually not a list, though it can be). That final value often summarizes, combines, or otherwise accumulates the information of all of the individual list elements. We use folding as a metaphor because we imagine the list written out on a strip of paper, which we then fold up in a zig-zag and flatten into a single value.

When we fold up a list, we start with a "base" value, a foundation upon which the folding occurs, and which will serve as the result of folding the empty list. We also choose a process for "combining" each element of the list with the base, one at a time. Those successive combinations produce ever more complete answers to sub-problems on *parts* of the list, so that by the time we're done we have the answer for the *whole* list.

If that seems mysterious, note that it's a perfect description of the two list functions we wrote in the previous lesson:

- In the case of `sum-list`, the base value was `0` and the combining step was addition (+). We start with `0` and add successive values from the list onto it. At the end of the process, we're left with the sum of all the elements in the list.
- In the case of `my-length`, the base value was also `0`. The combining step ignores the current list value and simply adds one to the result to the sub-problem. At the end of the process, we're left with the number of elements in the list.

We can see this pattern being played out in the condensed traces of these functions. In the two examples below, step forward, but not all the way to the end. Stop right after the function is applied to the empty list, when the base appears.

```
(sum-list (cons 7 (cons 8 (cons 9 (cons 10 empty)))))
```

<< First Step | < Step Back | Step 1 of 19 | Step Forward > | Last Step >>

```
(my-length (cons 7 (cons 8 (cons 9 (cons 10 empty)))))
```

<< First Step | < Step Back | Step 1 of 19 | Step Forward > | Last Step >>

As demonstrated by these examples, the process of folding eats through a list and leaves behind a nested sequence of computations that boil the list elements one-by-one down to a final value. Visualizing this pattern might help you solve problems using the folding idiom.

Concept Check 05.4.1

The code editor below shows our implementation of `sum-list`. Modify the function into a new function called `mult-list`, which consumes a list of numbers and produces the product of all the numbers in the list. For example, `(mult-list (cons 3 (cons 4 (cons 5 empty))))` ⇒ `60`.

The conversion should require very few changes:

- Rename the function (and the recursive call!)
- Choose a new base value
- Choose a new function for combining list values with recursive results.

1/1 points
Attempts: 2 / Unlimited

```
(define (mult-list lst)
  (cond
    [(empty? lst) 1]
    [else (* (first lst) (mult-list (rest lst)))]))
```

Submit Code | Reset Code

✔ Correct! 1/1 points

Concept Check 05.4.2

As another minor variation on the examples above, write a function `glue-strings` that consumes a list of strings `los` and produces a single string obtained by using `string-append` to attach them all together. For example, `(glue-strings (cons "CS" (cons " " (cons "115" empty))))` ⇒ `"CS 115"`. A modified list template is provided as a starting point. Remember to complete the function by relying on the template—don't guess at the structure of the recursive case!

1/1 points
Attempts: 1 / Unlimited

```
(define (glue-strings los)
  (cond
    [(empty? los) ""]
    [else (string-append (first los) (glue-strings (rest los)))]))
```

Submit Code | Reset Code

✔ Correct! 1/1 points

Mapping a list

Next, let's have a look at the mapping idiom. The intended behaviour of this idiom is easier to explain than that of folding. We have a list and some sort of transformation operation, and we want to produce a new list of the same length, where each value from the original list has had the transformation applied to it.

For example, if we want to map the built-in `sqr` function over the list `(cons 2 (cons 4 (cons 5 empty)))`, we would expect to get back `(cons 4 (cons 16 (cons 25 empty)))`. Or, if we map `string-length` over `(cons "a" (cons "abc" (cons "abcde" empty)))`, we'd expect `(cons 1 (cons 3 (cons 5 empty)))`.

Let's work through the example of negating a list. We can negate any individual number using the unary minus function: `(- 6)` ⇒ `-6`. How do we apply negation to every element of a list of numbers? We start with standard bookkeeping: we grab a copy of the list template, fix up the contract, and rename the function (here to `negate-list`):

```
;; negate-list: (listof Num) -> (listof Num)
(define (negate-list lon)
  (cond
    [(empty? lon) ...]
    [else (... (first lon) ...
           ... (negate-list (rest lon)) ...)]))
```

What should we fill in for the base case of an empty list? Well, the map idiom tells us that the result should be a list of the same length. When the length is zero, there's only one option: the empty list.

Now let's put on our recursive thinking caps. Consider an example like `(negate-list (cons 2 (cons -4 (cons 5 empty))))`. The template tells us that we'll have two values at our disposal:

- `(first lon)`, which for this list will be 2.
- `(negate-list (rest lon))`, which we'll assume (by leap of faith) is correctly evaluated as `(cons 4 (cons -5 empty))`.

Our job is to write an expression that turns 2 and `(cons 4 (cons -5 empty))` into the negation of the whole list: `(cons -2 (cons 4 (cons -5 empty)))`. It looks like a combination of `–` and `cons` will do the trick. Here's the final function:

```
;; negate-list: (listof Num) -> (listof Num)
(define (negate-list lon)
  (cond
    [(empty? lon) empty]
    [else (cons (- (first lon))
                (negate-list (rest lon)))]))
```

The use of `cons` in the body of the function is a strong hint that the function will eventually produce a list.

Concept Check 05.4.3

Write a function `sqr-list` that consumes a list of numbers and produces a list of the squares of those numbers, computed using the built-in `sqr` function. Write the function by refining the template provided below. You may find it useful to refer back to the implementation of `negate-list` above.

1/1 points
Attempts: 1 / Unlimited

```
(define (sqr-list lon)
  (cond
    [(empty? lon) '()]
    [else (cons (sqr (first lon))
              (sqr-list (rest lon)))]))
```

Submit Code | Reset Code

✔ Correct! 1/1 points

Concept Check 05.4.4

Write a function `shout-list` that consumes a list of strings and produces a list in which all lower-case characters in those strings have been converted to upper-case, using the built-in function `string-upcase`.

1/1 points
Attempts: 2 / Unlimited

```
(define (shout-list lst)
  (cond
    [(empty? lst) empty]
    [else (cons (string-upcase (first lst))
                (shout-list (rest lst)))]))
```


Submit Code | Reset Code

✔ Correct! 1/1 points

Let's finish up with mapping by having a look at the pattern that mapping produces when performing a condensed trace.

```
(negate-list (cons 2 (cons -4 (cons 5 (cons 9 empty)))))
```

<< First Step | < Step Back | Step 1 of 11 | Step Forward > | Last Step >>

We can see that mapping eats through the list like folding, and spits out a sequence of `cons`es together with the transformed list elements. When the recursive function reaches the end of the list and encounters `empty`, it stops and leaves behind `empty`, allowing all the `cons`es to assemble back into the result list.

Filtering a list

Like mapping, filtering a list is easy to describe. We start with a list, together with a property we'd like to test for every element of that list. Our goal is to produce a *sublist* of the original list, consisting of just those elements that have the property. For example, if we're filtering for positive numbers, then filtering the list `(cons 4 (cons -3 (cons -8 (cons 1 empty))))` would produce `(cons 4 (cons 1 empty))`. If we were instead asking for even numbers, we'd end up with `(cons 4 (cons -8 empty))`.

A passing grade in Waterloo CS courses is 50%. Given a list of grades (natural numbers between 0 and 100, inclusive), let's filter the list to produce a sub-list of passing grades. We'll start with a lightly modified template:

```
;; passing-grades: (listof Nat) -> (listof Nat)
;; Requires: every element of lon is between 0 and 100, inclusive
(define (passing-grades lon)
  (cond
    [(empty? lon) ...]
    [else (... (first lon) ...
           ... (passing-grades (rest lon)) ...)]))
```

Once again, the base case is straightforward. If you start with an empty list of grades, it doesn't contain any passing grades, so you've still got the empty list. we'll put in `empty` as the first `cond` answer.

For the recursive case, let's consider a general example like `(cons 87 (cons 45 (cons 61 empty)))`. We know that `(first lon)` is 87 for this list. If we take the leap of faith, we can assume that the recursive call `(passing-grades (rest lon))` will successfully skip over the 45 and give us the one-element list `(cons 61 empty)`. In this case we're left with a passing grade and a recursive list of passing grades, so all that's left to do is `cons` them together.

However, that's not quite enough. What if the first list element is below 50? In that case we want to *discard* that grade and keep going with just the remaining list items. We can do that by handing back the recursive result as the result of the whole function. We use a nested `cond` to decide what to do:

```
;; passing-grades: (listof Nat) -> (listof Nat)
;; Requires: every element of lon is between 0 and 100, inclusive
(define (passing-grades lon)
  (cond
    [(empty? lon) empty]
    [else (cond
            [(>= (first lon) 50)
             (cons (first lon) (passing-grades (rest lon)))]
            [else (passing-grades (rest lon))])]))
```

Notice how in the `else` case, where `(first lon)` is less than `50`, the `cond` answer doesn't have a `cons` in it: we just hand off whatever the recursive call tells us. If we have a passing grade at the start of the list, we use `cons` to keep it. In effect, the nested `cond` contains *two* copies of the recursive part of the original list template, each one customized in a slightly different way.

Let's make one last simplification to this code. We use a handy rule of thumb: if you see "`[else (cond`" in your code, you can almost certainly remove it, and collapse the nested `cond` into its parent:

```
;; passing-grades: (listof Nat) -> (listof Nat)
;; Requires: every element of lon is between 0 and 100, inclusive
(define (passing-grades lon)
  (cond
    [(empty? lon) empty]
    [(>= (first lon) 50)
     (cons (first lon) (passing-grades (rest lon)))]
    [else (passing-grades (rest lon))]))
```

A condensed trace of a function obeying the filter idiom doesn't give us much new insight. The function eats the list, spits out elements that have the property of interest (in this case, passing grades), and silently absorbs the elements that don't have the property:

```
(passing-grades (cons 77 (cons 46 (cons 98 (cons 32 empty)))))
```

<< First Step | < Step Back | Step 1 of 11 | Step Forward > | Last Step >>

Concept Check 05.4.5

By borrowing ideas from the `passing-grades` example above, write a function `eat-apples`. It consumes `los`, a list of symbols representing types of fruit, and produces a sublist consisting of all the fruits from the original list that aren't `'apple`. For example, `(eat-apples (cons 'pear (cons 'apple (cons 'orange empty))))` would produce `(cons 'pear (cons 'orange empty))`.

The list template is not included in this question, or most subsequent questions. But of course you should still use it! In particular, stick to the pattern of the recursive case (here `(... (first los) ... (eat-apples (rest los)) ...)`) to avoid getting lost.

1/1 points
Attempts: 1 / Unlimited

```
;; eat-apples: (listof Sym) -> (listof Sym)
(define (eat-apples los)
  (cond
    [(empty? los) empty]
    [(equal? 'apple (first los)) (eat-apples (rest los))]
    [else (cons (first los) (eat-apples (rest los)))]))
```

Submit Code | Reset Code

✔ Correct! 1/1 points

Concept Check 05.4.6

Write a function `renormalize` that consumes a list of extended numbers (the `ExtNum` type discussed earlier consisting of all `Num`s together with the symbol `'infinity`) and produces a new list consisting of the non-infinite members of the original list, in the order that they appear. For example, `(renormalize (cons 4.5 (cons 'infinity (cons -8 empty))))` would produce `(cons 4.5 (cons -8 empty))`.

1/1 points
Attempts: 1 / Unlimited

```
;; renormalize: (listof ExtNum) -> (listof ExtNum)
(define (renormalize lon)
  (cond
    [(empty? lon) empty]
    [(not (equal? (first lon) 'infinity))
     (cons (first lon) (renormalize (rest lon)))]
   [else (renormalize (rest lon))]))
```

Submit Code | Run Code | Reset Code

✔ Correct! 1/1 points

Summary

The three idioms introduced here aren't just a random sampling of recursive list computations: they're arguably the three most important ways that we work with lists. They're important in a wide range of real-world contexts, and methods for folding, mapping, and filtering can be found in almost all modern programming languages.

When you're given a list processing problem in this course, a natural starting point for solving it is to ponder whether it's a variation of one of these idioms. If one of them seems like a good fit, you can try to adapt the ideas in this lesson as you work.

Later in the course we'll revisit these three idioms with a great deal more sophistication and experience. We'll see that there are more advanced techniques available in Racket that allow us to embody folding, mapping, and filtering in built-in functions, saving us from the nitty-gritty work of writing templates and recursive code.

Discussion

Hide Discussion

Topic: Module 05 / 05.4 List Idioms

Add a Post

Show all posts | by recent activity

There are no posts in this topic yet.