05.5 List Variations

Bookmark this page

The previous lesson introduced three important idioms that arise frequently when processing lists: folding, mapping, and filtering. Before we leave this module, it will be useful to demonstrate a few other common forms of list processing. This lesson contains four scenarios that tend to come up a lot in practice.

Auxiliary data

Suppose we are given a number n and a list of numbers `lon`, and our goal is to produce a new list where n has been added to every member of the list. If our function is called `add-to-all`, we might expect `(add-to-all 4 (cons 2 (cons 3 (cons 7 empty))))` to produce `(cons 6 (cons 7 (cons 11 empty)))`. In the language of the previous lesson, this seems like a job for the map idiom: we're producing a new list by performing the same transformation to every element of the old list. Let's start by writing down a partial solution that's similar to `negate-list`, in which the value n is stashed in a constant instead of passed as a parameter. Don't worry, we'll fix that in a minute. This is a convenient starting point closely derived from the list template.

```
(define n 4)

;; add-to-all: (listof Num) -> (listof Num)
(define (add-to-all lon)
  (cond
    [(empty? lon) empty]
    [else (cons (+ n (first lon))
      (add-to-all (rest lon)))]))
```

The problem with this approach is that we get to define just one value of n when we write the program, and that becomes the only value we know how to add to numbers in lists. But in a single real-world program we might want to add 4 to list elements in one place, 7 in another place, and any other unknown number elsewhere. We should rearrange the function so that n is a parameter, not a constant. When we do this, we just have to keep one thing in mind: we've got to add the extra parameter to the recursive call too!

```
;; add-to-all: Num (listof Num) -> (listof Num)
(define (add-to-all n lon)
  (cond
    [(empty? lon) empty]
    [else (cons (+ n (first lon))
      (add-to-all n (rest lon)))]))
```

We call n an **auxiliary parameter**: it participates in the behaviour of the function, but doesn't affect the pattern of the recursion. We sometimes say that the auxiliary parameter "goes along for the ride" in the recursion: each invocation of the function hands a copy of the parameter to the next recursive call in line, so that they all have access to this piece of information. Please note that when we carry around these auxiliary values, *we don't change them*: whatever value we receive as a parameter is the same value we pass down recursively. This isn't a fundamental law of recursion, but it's a simplification we'll adhere to in this course. Keep that in mind, as it's one more way to avoid confusion when working with recursion.

Concept Check 05.5.1

Write a function `snoc` that behaves like the opposite of `cons`. It consumes a number `n` and a list of numbers `lon`, and produces a new list in which `n` has been inserted at the *end* of `lon`. For example, `(snoc 3 (cons 1 (cons 2 empty)))` would produce `(cons 1 (cons 2 (cons 3 empty)))`. Use the pattern above, treating `n` as auxiliary data.

1/1 points
Attempts: 1 / Unlimited

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

Submit Code | Reset Code

Correct! 1/1 points

Concept Check 05.5.2

Write a function `count` that consumes a list `lst` and a value `val`. The function computes the number of times that `val` appears in `lst`. Use the list folding idiom, modified to allow `val` to go along for the ride as auxiliary data.

1/1 points
Attempts: 1 / Unlimited

```
;; count: (listof Any) Any -> Nat
(define (count lst val)
  (cond
    [(empty? lst) 0]
    [(equal? val (first lst))
     (add1 (count (rest lst) val))]
   [else (count (rest lst) val)]))
```

Submit Code | Reset Code

Correct! 1/1 points

Stopping early

There's a built-in Racket function called `member?` that consumes a value and a list, and determines if the value occurs anywhere in the list. How might we write such a function? Let's start as usual with the list template, calling our function `my-member?` so that we don't conflict with the built-in version. As above, we'll also introduce an auxiliary parameter to represent the value we're seeking. That value will go along for the ride in the recursion: note again how it appears unchanged in the recursive call.

```
;; my-member?: Any (listof Any) -> Bool
(define (my-member? val lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst) ...
          ... (my-member? val (rest lst)) ...)]))
```

As usual, we'll deal with the base case first. If you're looking for `val` in the empty list, you're guaranteed not to find it. We should therefore put `false` in for the base case.

Let's now say that `lst` is non-empty. We have some list item `(first lst)`, and in this case what we care about is the possibility that it's equal to `val` (using `equal?`). If these two values are equal, what should we do? The recursive call is redundant in this case: we've found the value we're looking for, and no further searching in the list will change our answer. We can simply produce `true` immediately. It's only if we *don't* see `val` at the start of the list that it makes sense to keep looking deeper into the list. So, our next refinement is to add a question/answer pair to the `cond` that checks for equality and stops immediately:

```
;; my-member?: Any (listof Any) -> Bool
(define (my-member? val lst)
  (cond
    [(empty? lst) false]
    [(equal? val (first lst)) true]
    [else (my-member? val (rest lst))]))
```

This new question/answer pair is a *second base case*—another way that the function can stop immediately without performing a recursive call. Have a look at the full trace below. The function grinds through the first list value. When it gets to the second list value, it discovers the element it's looking for and stops abruptly.

```
(my-member? 5 (cons 2 (cons 5 (cons 8 (cons 9 empty)))))
```

<< First Step | < Step Back | Step 1 of 29 | Step Forward > | Last Step >>

There are many reasons why a recursive function might not need to process the entire list from start to finish. You can always modify the list template to include any many additional base cases as are needed to handle these situations.

Concept Check 05.5.3

Write a function `all-even?` that consumes a list of integers and produces `true` if and only if every number in the list is even. If you encounter an odd number as you traverse the list, stop immediately and produce `false`. (Note that every number in the empty list is even.)

1/1 points
Attempts: 1 / Unlimited

```
;; all-even?: (listof Int) -> Bool
(define (all-even? lon)
  (cond
    [(empty? lon) true]
    [(odd? (first lon)) false]
    [else (all-even? (rest lon))]))
```

Submit Code | Reset Code

Correct! 1/1 points

Concept Check 05.5.4

Write a function `has-long-string?` that consumes a list of strings and produces `true` if and only if the list contains at least one string of length 10 or greater. Again, if you encounter such a string, you should stop immediately and produce `true`.

1/1 points
Attempts: 1 / Unlimited

```
;; has-long-string?: (listof Str) -> Bool
(define (has-long-string? los)
  (cond
    [(empty? los) false]
    [(>= (string-length (first los)) 10) true]
    [else (has-long-string? (rest los))]))
```

Submit Code | Reset Code

Correct! 1/1 points

Recursive string processing

Module 03 hinted at the fact that a string is like a sequence of characters. Let's see how that relationship can be made precise and used in practice.

Characters are the atomic units of text. Every letter, number, symbol, and punctuation mark is a character, along with countless other glyphs and emoji. In Racket, a typical character literal can be written by preceding the character of interest with `#\`:

```
> #\a
#\a
> #\5
#\5
> #\!
#\!
> #\स ; Devanagari
#\स
```

A few unusual characters, like spaces, have special representations:

```
> #\space ; Still just a single space character
#\space
> #\newline
#\newline
```


In order to talk about characters in the Design Recipe, we will give them the type name `Char`, usually pronounced "car" with a hard C.

What does any of this have to do with lists? Racket includes built-in functions `string->list` and `list->string`, allowing us to convert back and forth between a `Str` and a `(listof Char)`:


```
> (string->list "Char")
(cons #\C (cons #\h (cons #\a (cons #\r empty))))
> (string->list "古池や")
(cons #\古 (cons #\池 (cons #\や empty)))
> (list->string (cons #\C (cons #\space (cons #\S empty))))
"C S"
```

Using these functions, we can now write new functions that transform strings by processing them character-by-character. The idea is to convert the string into a list, work through that list recursively, and reassemble the result into a string. For example, suppose we want to redact digits in a string, replacing them all with asterisks (*). For starters, we can use the built-in predicate `char-numeric?` to write a function that operates on a single character, producing an asterisk if given a digit and returning every other argument unchanged.

```
;; redact-char: Char -> Char
(define (redact-char ch)
  (cond
    [(char-numeric? ch) #\*]
    [else ch]))
```

From there, it's straightforward to apply this transformation to every character in a list of characters—that's just another use of the mapping idiom.

```
;; redact-list: (listof Char) -> (listof Char)
(define (redact-list loc)
  (cond
    [(empty? loc) empty]
    [else (cons (redact-char (first loc))
                (redact-list (rest loc)))]))
```

The last step is simply to use this function to process a string. That can be done in a single, elegant line of code, which unpacks, processes, and re-packs the string:

```
;; redact: Str -> Str
(define (redact str)
  (list->string (redact-list (string->list str))))
```

Remember that although we *say* "unpack, process, re-pack", the code is *written* in the order `list->string`, `redact-list`, `string->list`. The functions are evaluated from the inside out.

We've built a little tower of three functions here: `redact-list` is an engine for applying `redact-char` to every character in a list. Then, the main `redact` function has the narrow task of hiding the underlying list machinery from view, so that the rest of the program can operate in terms of strings. In a way, the real "meat" of this process is the recursive `redact-list` function.

In this context we refer to `redact` as a **wrapper function**: it's a thin layer of code that "sets the stage" for the real work. As with helper functions, there's no rigorous mathematical definition of a wrapper function—it's an informal term we use to talk about the structure of a program. A typical wrapper function will apply a simple transformation (like `string->list`) to its arguments to prepare the data for consumption by another function, and sometimes post-process the result of that other function (like `list->string`) to present a final result.

Concept Check 05.5.5

We would like to write a function `remove-whitespace` that consumes a string `str` and produces a new string identical to the first, but with all whitespace characters removed. For example, `(remove-whitespace "superb owl !")` should produce `"superbowl!"`.

We will write this function as a wrapper that uses a recursive helper function `remove-ws-list` that consumes and produces lists of characters (and that will need to use the filtering idiom). Below, we provide `remove-whitespace` and the header for `remove-ws-list`. Complete the function `remove-ws-list`. You will find the built-in function `char-whitespace?` helpful.

1/1 points
Attempts: 1 / Unlimited

```
;; remove-ws-list: (listof Char) -> (listof Char)
(define (remove-ws-list loc)
  (cond
    [(empty? loc) empty]
    [(char-whitespace? (first loc)) (remove-ws-list (rest loc))]
    [else (cons (first loc) (remove-ws-list (rest loc)))]))

;; remove-whitespace: Str -> Str
(define (remove-whitespace str)
  (list->string (remove-ws-list (string->list str))))
```

Submit Code | Reset Code

Correct! 1/1 points

Non-empty lists

The example we used to motivate recursive list processing was computing the sum of a list of numbers. In that case we were able to take advantage of the mathematical fact that zero is the additive identity (that is, $x + 0 = 0 + x = x$ for all numbers x). We used this fact implicitly in the base case for `sum-list`, by producing the "default" value `0` for an empty list.

What if we want to find the *maximum* of a list of numbers? That's another simple operation we frequently want to perform on a list. But if you start trying to write a function `list-max` based on the list template, you quickly run into trouble. What value should we produce in the base case? In other words, what's the maximum of an empty list? The simple answer is that there is no answer: if we require that the maximum of a list be one of the items in the list, then an empty list, which contains no items, can't have a maximum. We might be able to get around this limitation by making up a special value to mean "negative infinity", as with the `ExtNum` examples we've already seen in this course. A solution that's both better and more broadly applicable is to assume going in that our list must be non-empty.

Let's re-examine the data definition for a list of numbers:

```
;; A (listof Num) is one of:
;; * empty
;; * (cons Num (listof Num))
```

The empty list is unavoidable—it's right there in the base case of the data definition. We can't excise it altogether, because we need *some* base case. But we can invent a new type that's closely related to a list, but which has a non-empty base case. Let's temporarily use the type name `NonEmptyNumList`:

```
;; A NonEmptyNumList is one of:
;; * (cons Num empty)
;; * (cons Num NonEmptyNumList)
```

Our new base case is simply a one-element list (instead of a zero-element list!). Because the recursive case is built upon the base case, we know that `NonEmptyNumLists` will always have one or more elements in them. Following data-directed design, we can next use the data definition to produce a template. Our new base can't just check if the passed-in list is empty (which wouldn't even be a legitimate `NonEmptyNumList`). Instead we peek past the the first element of the list, and check whether the *rest* of the list is empty, indicating that we're in the base case of a one-element list. The recursive case is identical to that of a regular list (which makes sense, because the recursive case hasn't changed in the data definition). We end up with the following template:

```
;; nonemptynumlist-template: NonEmptyNumList -> Any
(define (nonemptynumlist-template nenl)
  (cond
    [(empty? (rest nenl)) ... (first nenl) ...]
    [else (... (first nenl) ...
          ... (nonemptynumlist-template (rest nenl))...)]))
```

Notice how the answer in the base case assumes you want to do something with `(first nenl)`. We're dealing with a non-empty list in this case, and we assume that we'll want to do something with the single element in a one-element list.

We can now refine the template into a function `max-list` that computes the maximum of a (non-empty) list of numbers. The base case is a one-element list, in which case we produce that single element as the maximum. Otherwise, we use the built-in `max` function to produce the larger of the first element of the list and the result of the recursive call (i.e., the maximum of everything else):

```
;; max-list: NonEmptyNumList -> Num
(define (max-list nenl)
  (cond
    [(empty? (rest nenl)) (first nenl)]
    [else (max (first nenl) (max-list (rest nenl)))]))
```

This example can easily be adapted to other scenarios with lists of other types, except for the fact that `NonEmptyNumList` is forever associated with the type `Num`. With regular lists we introduced the `listof` notation to talk about lists of different element types. We could do something similar here, but we'll adopt a simpler plan: when a list must be non-empty, we'll stick with our existing list types and mention the non-empty constraint in the Requires section:

```
;; max-list: (listof Num) -> Num
;; Requires: nenl is non-empty
(define (max-list nenl)
  ...)
```

We write this as a shorthand for the work we did above. We still *think* about problems with non-empty lists in terms of the data definition and template we produced for `NonEmptyNumList`, but take them for granted in the Design Recipe.

Concept Check 05.5.6

Write a predicate `all-same?` that consumes a non-empty list of numbers `lon`, and determines whether all the numbers in the list are identical. You may find the built-in function `second` useful.

1/1 points
Attempts: 1 / Unlimited

```
;; all-same?: (listof Num) -> Bool
;; Requires: lon is non-empty
(define (all-same? lon)
  (cond
    [(equal? (rest lon) empty) true]
    [(= (first lon) (first (rest lon))) (all-same? (rest lon))]
    [else false]))
```

Submit Code | Reset Code

Correct! 1/1 points

Discussion

Topic: Module 05 / 05.5 List Variations

Hide Discussion

Add a Post

Show all posts | by recent activity

- **Question regarding Concept Check 05.5.2** (2)
  By placing the following function into Drracket, I get the desired outcome. The only difference is the positioning of the parameters as I have "val" before "lst": ;; count: Any (listo...
- **Still not understanding 5.5.2** (3)
  This is what I have so far. I don't know how to make the code count all the numbers. ;; count: (listof Any) Any -> Nat (define (count lst val) (cond [(empty? lst)] [(equal? val (first...
- **5.5.5 why is this not correct?** (3)
- **Concept Check 05.5.6** (8)
  I've tried writing the code a lot of different ways now and testing them out in DrRacket. I keep getting the same errors when testing out my function with lists of odd numbers us...
- **Help with 05.5.1** (3)
  Hi there, I'm a little stumped on how I can move the snoc 3 term to the other end of the code. So far, I'm ending up with the first two items of the list, but I can't seem to get the ...
- **05.5.2 why this is not correct?** (2)
  ;; count: (listof Any) Any -> Nat (define (count lst val) (cond [(empty? lst) 0] [(= val (first lst)) (+ 1 (count (rest lst) val))] [else (+ 0 (count (rest lst) val))]))