Better Abstraction Guide

1. Identify the **similarities** in the signature. This will serve as the building block for your abstracted function.

```
; [List-of String] Number -> [List-of String]
; ... some function purpose ...
(define (function-A los n) ...)

; [List-of [List-of Number]] Symbol -> [List-of [List-of Number]]
; .... katsu goes here ...
(define (function-B lolon s) ...)

; Looking at their signatures
; function-A: [List-of String] Number -> [List-of String]
; function-B: [List-of [List-of Number]] Symbol -> [List-of [List-of
Number]]

; What do we notice? Let's Start with Input:
; Input can be broken into two parts:

; function-A: [List-of String] ...
; function-B: [List-of [List-of Number]] ...

; function-A: ... Number
; function-B: ... Symbol

; Okay, what about output?

; function-A: [List-of String]
; function-B: [List-of [List-of Number]]

; We're noticing patterns here:
; function-A: [List-of X] Y -> [List-of X]
;             where X = String, Y = Number
; function-B: [List-of X] Y -> [List-of X]
;             where X = [List-of Number], Y = Symbol

; We can generalize/abstract this then:
; [X Y] ... [List-of X] Y -> [List-of X]
; NOTE:  ^ means that we still need more information to generalize
completely
(define (my-abstr ... lx y))
```

2. Convert your functions in terms of your abstraction (use X and Y instead of the actual code.)

```
; ===== Actual Function ===== ;

; [List-of String] Number -> [List-of String]
; ... some function purpose ...
(define (function-A los n)
  (cond
   [(empty? los) '())]
   [else (cons (first los)
               (cons (number->string n)
                     (function-A (rest los) n) ))]))

; [List-of [List-of Number]] Symbol -> [List-of [List-of Number]]
; .... katsu goes here ...
(define (function-B lolon s)
  (cond
   [(empty? lolon) '()]
   [else (cons (first lolon)
               (cons (symbol->lon s)
                     (function-B (rest lolon) s)))]))

; Symbol -> [List-of Number]
; ... i don't know what this does ...
; ... some conversion magic happens here ...
(define (symbol->lon s) ...)

; ===== Abstracted Bastardizations ===== ;

; [List-of X] Y -> [List-of X]
; ... some function purpose ...
(define (function-A LoX Y)
  (cond
   [(empty? LoX) '())]
   [else (cons (first LoX)
               (cons (number->string Y)
                     (function-A (rest LoX) Y) ))]))

; [List-of X]] Y -> [List-of X]
; .... katsu goes here ...
(define (function-B LoX Y)
  (cond
   [(empty? lolon) '()]
   [else (cons (first LoX)
               (cons (symbol->lon Y)
                     (function-B (rest LoX) Y)))]))

; the helper doesn't need to be changed because it's not super important to
abstracting the main functions
```

3. Identify the **differences** in the function themselves. Think about what data-type they would represent in your abstraction.

```
; X = String
; Y = Number
; [List-of X] Y -> [List-of X]
; ... some function purpose ...
(define (function-A LoX Y)
  (cond
   [(empty? LoX) '())]
   [else (cons (first LoX)
               (cons (number->string Y)
                     (function-A (rest LoX) Y) ))]))

; X = [List-of Number]
; Y = Symbol
; [List-of X]] Y -> [List-of X]
; .... katsu goes here ...
(define (function-B LoX Y)
  (cond
   [(empty? LoX) '()]
   [else (cons (first LoX)
               (cons (symbol->lon Y)
                     (function-B (rest LoX) Y))))]))

;; What are the physical differences between the two functions? Aside
from function name, of course.

; number->string and symbol->lon, right?

; What are their data types, in terms of X and Y?

; Number -> String
; ... obv, you didn't write this, but this is what it would be, right?
(define (number->string ...))
; In terms of function-A:
; X = String
; Y = Number
; So, number->string's new signature can be [Y -> X]

; The same is the case for the helper
; Symbol -> [List-of Number]
(define (symbol->lon s) ...)
; In terms of function-B:
; X = [List-of Number]
; Y = Symbol
; So, symbol->lon can also be [Y -> X]
```

4. Put things into your abstraction

```
; Our abstraction from earlier:

; [X Y] ... [List-of X] Y -> [List-of X]
(define (my-abstr ... lx y) ...)

; We need to put a [Y -> X] somewhere, since function-A and function-B
have different ones

; So, we'll make it a new parameter for our function, call it f.

; [X Y] [Y -> X] [List-of X] Y -> [List-of X]
(define (my-abstr f lx y) ...)

; That's our complete signature for the abstraction.
```

5. Fill in the rest

```
; This is one of our functions (doesn't matter what we choose.)

; [List-of X]] Y -> [List-of X]
; .... katsu goes here ...
(define (function-B LoX Y)
  (cond
   [(empty? LoX) '()]
   [else (cons (first LoX)
               (cons (symbol->lon Y)
                     (function-B (rest LoX) Y)))]))

; This is our abstraction so far:

; [X Y] [Y -> X] [List-of X] Y -> [List-of X]
(define (my-abstr f lx y) ...)

; Now we just copy things over, accomodating for any specialties

; [X Y] [Y -> X] [List-of X] Y -> [List-of X]
(define (my-abstr f lx y)
  (cond
   [(empty? lolon) '()]
   [else (cons (first lx)
               (cons (f Y) ; <- This is one change we have to make,
because f
                           ;    differs depending on function.
                     (function-B (rest LoX) Y)))]))


; That's it, we're done!
```

6. Noice!