CS 135 — L19 - Functional Abstraction

Luke Lu • 2025-11-18

Info — `build-list`

`build-list` constructs a list

```
(build-list length (lambda (x) ... ))

;; i.e.
(build-list 4 (lambda (x) x)) ==> (list 0 1 2 3)
```

Info — `map`

`map` transforms a list

```
(map (function) list list)

;; i.e.
(map add1 (list 1 2 3 4)) => (list 2 3 4 5)
(map (lambda (x) (+ x 2)) (list 0 1 2 3)) ==> (list 2 3 4 5)
(map + (list 1 2 3) (list 1 2 3)) ==> (list 2 4 6)
(map list (list 1 2 3) (list 1 2 3)) ==> (list (list 1 1) (list 2 2) (list 3 3))
```

Info — `foldr`

`foldr` compresses list to single value direction: left → right

```
(foldr (function) base-case lst)

;; x is first of the list
;; y is the result of recurring on the rest of the list (or the base case)

;; i.e.

;; string-append is a build-in function that contentates strings
;; string-length produces the length of a given string

(foldr string-append "" (list "To" "Be" "Or" "Not" "2B"))
==> "ToBeOrNot2B"
(foldr (lambda (x y) (cons (* 2 x) y)) empty (list 0 1 2 3 4))
==> (list 0 2 4 6 8)
```

Implementing `map` and `filter` with `foldr`

```
(define (filtering ? lst)
 (foldr (lambda (x y) (cond [(? x) (cons x y)] [else y]))  empty lst))

(define (mapping f lst)
 (foldr (lambda (x y) (cons (f x) y)) empty lst))
```

🐳 **Info — `foldl`**
`foldl` compresses list to single value direction: right → left

```
(foldl (function) base-case lst)

;; x is first of the list
;; y is the result of recurring on the rest of the list (or the base case)
```

Implementation of `foldl` from scratch

```
(define (my-foldl f base lst)

 (local [(define (foldl/acc lst acc)
   (cond
     [(empty? lst) acc]
     [else (foldl/acc (rest lst) (f (first lst) acc))]))]
   (foldl/acc lst base)))
```