Functional abstraction

CS135 Lecture 19

L19.0 Abstract list functions

Abstraction

“Abstraction” is the process of finding similarities or common aspects, and forgetting unimportant differences.

Over the term, we have seen many similarities between functions and captured them in templates and rules.

```
(define (natural-template n)
  (cond
    [(zero? n) ...]
    [... ...]
    [else (... n (natural-template (sub1 n)))]))
```

filter

Recall **`eat-apples`** and **`keep-odds`**. Those two functions had a very similar structure. Each selected items from a list to keep (or discard, depending on your viewpoint).We abstracted the structure into a function called **`filter`** that consumed a predicate governing the items to keep.

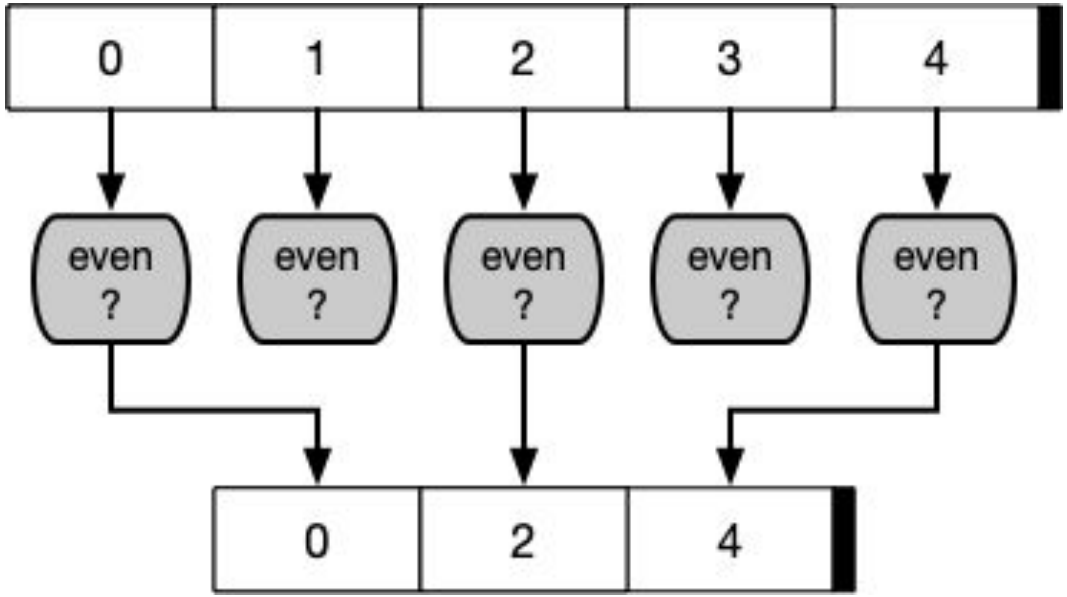
The function **`filter`** is an example of an “abstract list function”.

```
filter: (X -> Bool) (listof X) -> (listof X)
```

For example:

```
(define (eat-apples lst)
 (filter (lambda (sym) (not (symbol=? 'apple sym))) lst))

(define (keep-odds lst) (filter odd? lst))
```

Visualizing filter




```
(filter even? '(0 1 2 3 4))
```

Abstract list functions

We will now look for other patterns where we can perform similar abstractions

In CS135 we learn five of these “abstract list functions”, including **`filter`**:

`(filter pred? lst)`	Retain only those elements of a list for which **`pred?`** is **`true`**.
`(build-list n f)`	Construct a list by applying **`f`** to the numbers `0` to **`(- n 1)`**.
`(map f lst)`	Construct a new list by applying **`f`** to each element of a list.
`(foldr f base lst)`	Recursively combine (“fold”) elements of a list right to left.
`(foldl f base lst)`	Recursively combine (“fold”) elements of a list left to right.

What does it mean to “fold” a list? We will explain, but first let’s look at two simpler abstract list functions, **`build-list`** and **`map`**.

L19.1 `build-list`

build-list

A simple but useful built-in abstract list function is **build-list**.

```
build-list: Nat (Nat ->X) -> (listof X)
```

It consumes a natural number **n** and a function **f**, and produces the list:

```
(list (f 0) (f 1) ... (f (sub1 n)))
```

For example:

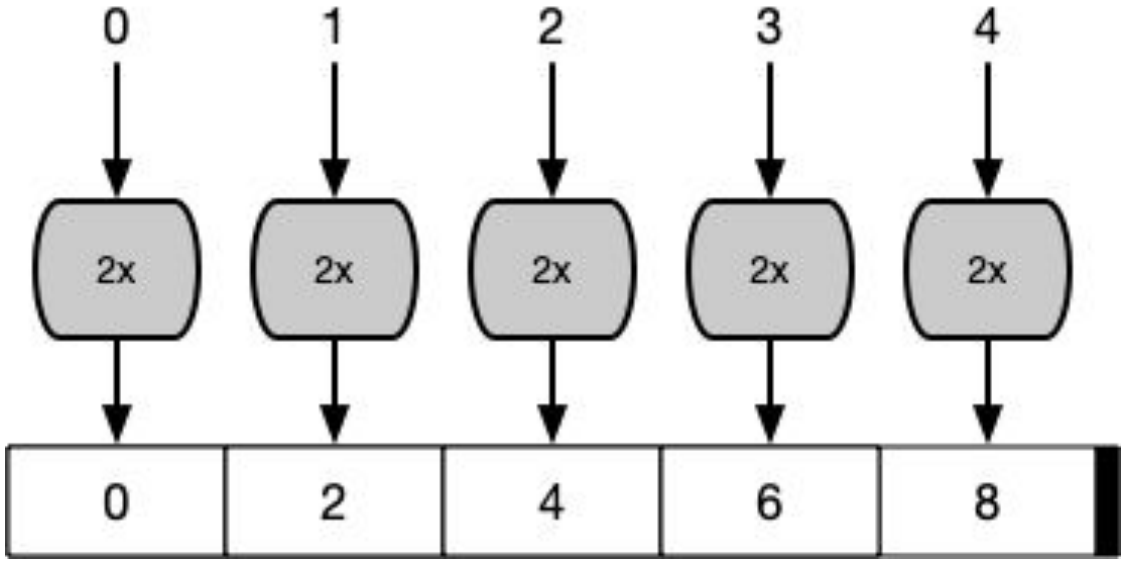
```
(build-list 4 (lambda (x) x)) ⇒ (list 0 1 2 3)
(build-list 4 (lambda (x) (* 2 x))) ⇒ (list 0 2 4 6)
```

The function **build-list** abstracts the “count up” pattern from lecture L14, and it is easy to write our own version.

We can implement an equivalent function ourselves

```
;; my-build-list: Nat (Nat -> X) -> (listof X)
(define (my-build-list n f)
  (local [(define (list-from i)
            (cond [(>= i n) empty]
                  [else (cons (f i) (list-from (add1 i)))]))]
    (list-from 0)))

(check-expect
 (my-build-list 4 (lambda (x) x)) (list 0 1 2 3))

(check-expect
 (my-build-list 4 (lambda (x) (* 2 x))) (list 0 2 4 6))
```

Visualizing `build-list`




```
(build-list 5 (lambda (x) (* 2 x)))
```

L19.2 `map`

Transforming a list

Another common pattern transforms each element of a list.

```
;; add1-list adds one to each element of a list
;; add1-list: (listof Num) -> (listof Num)
(define (add1-list lst)
  (cond [(empty? lst) empty]
        [else (cons (add1 (first lst))
                    (add1-list (rest lst)))]))

(check-expect (add1-list empty) empty)
(check-expect
 (add1-list (cons 10 (cons -6 (cons 999 empty))))
 (cons 11 (cons -5 (cons 1000 empty))))
```

Transforming a list

Transformations can be simple or complicated. Sometimes the exact transformation depends on the element being transformed.

```
(define (apples-to-oranges lst)
  (cond [(empty? lst) empty]
        [(symbol=? 'apple (first lst))
         (cons 'orange
               (apples-to-oranges (rest lst)))]
        [else
         (cons (first lst)
               (apples-to-oranges (rest lst)))]))
```

map

The built-in **map** abstract list function transforms a list, applying a function to each element and producing a new list comprising the result.

```
map: (X -> Y) (listof X) -> (listof Y)
```

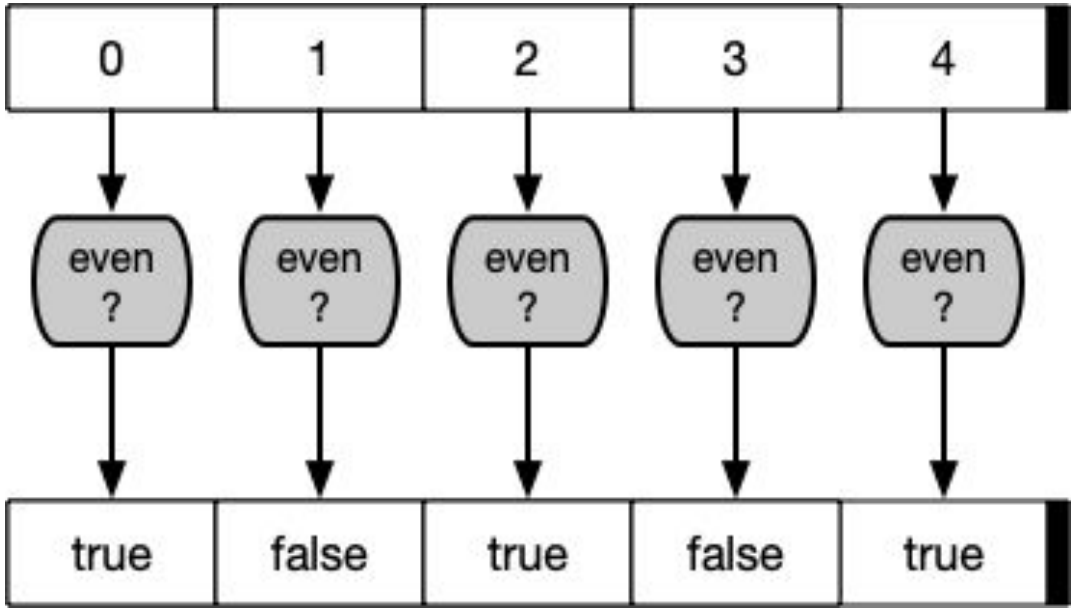
For example:

```
(map add1 '(0 1 2 3 4)) ⇒ '(1 2 3 4 5)

(map (lambda (x)
       (cond [(symbol=? x 'apple) 'orange]
             [else x]))
     '(apple eggs bread apple milk bread))
                  ⇒ '(orange eggs bread orange milk bread)
```

We can implement an equivalent function ourselves

```
;; my-map: (X -> Y) (listof X) -> (listof Y)
(define (my-map f lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst)) (my-map f (rest lst)))]))

(check-expect (my-map add1 '(0 1 2 3 4)) '(1 2 3 4 5))

(check-expect
 (my-map (lambda (x)
           (cond [(symbol=? x 'apple) 'orange] [else x]))
         '(apple eggs bread apple milk bread))
 '(orange eggs bread orange milk bread))
```

Visualizing `map`




```
(map even? '(0 1 2 3 4))
```

L19.3 `foldr`

Folding a list

A frequent pattern is to recurse on the **`rest`** of a list and then combine the result with the **`first`** of the list. The abstract list function **`foldr`** abstracts this pattern.

```
(define (sum-of-numbers lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst)
                 (sum-of-numbers (rest lst)))]))

(define (all-true? lst)
  (cond [(empty? lst) true]
        [else (and (first lst)
                   (all-true? (rest lst)))]))
```

Folding a list

To “fold” a list we supply a function that specifies how we are combining the **`first`** of a list with the result of recursing on the **`rest`** of the list. We must also supply a base case for the recursion.

In the case of **`sum-of-numbers`** this function is **`+`** and the base case is **`0`**.

```
(define (sum-of-numbers lst) (foldr + 0 lst))
(sum-of-numbers '(1 2 3 4 5)) ⇒ 15
```

In the case of **`all-true?`** this function is **`and`** and the base case is **`true`**.

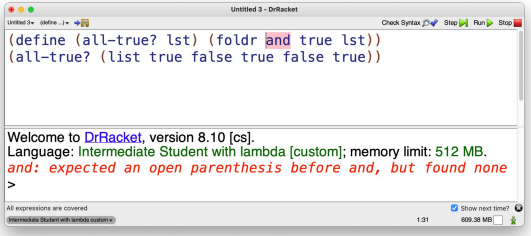
```
(define (all-true? lst)
  (foldr (lambda (x y) (and x y)) true lst))
(all-true? (list true false true false true)) ⇒ false
```

Folding a list

In the case of **`all-true?`** we would like to write:

```
(define (all-true? lst) and true lst)
```

However, this doesn't work for mysterious reasons.



Nonetheless, the example shows that the function used with **`foldr`** consumes two arguments:

```
(foldr (lambda (x y) (and x y)) true lst)
```

`first` of the list

result of recursing on the **`rest`** of the list (or the base case)

foldr

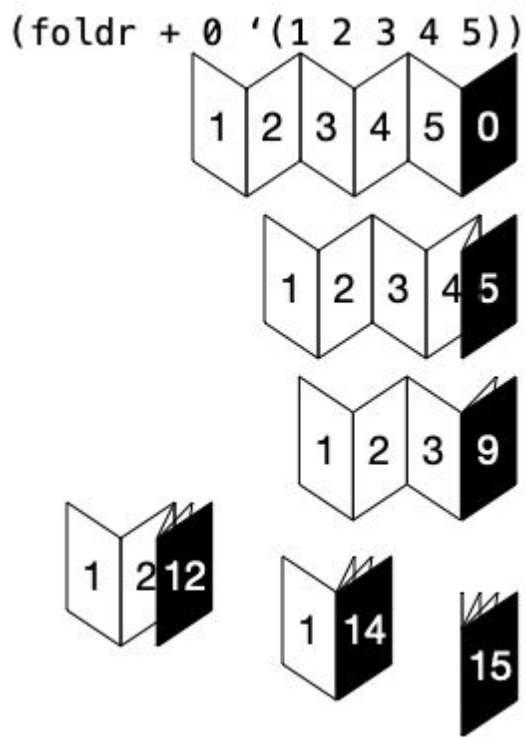
The **foldr** function recursively combines elements of a list **right to left**.

```
foldr: (X Y -> Y) X (listof X) -> Y
```

Starting with the base case and the last element, the function **f** is applied to the each element of the list along with the result of the previous application.

```
(foldr f base (list x1 x2 … xn))
    ⇒ (f x1 (f x2 … (f xn base)…))
```

Remember this pattern!!!



We can implement an equivalent function ourselves

```
;; my-foldr: (X Y -> Y) X (listof X) -> Y
(define (my-foldr f base lst)
  (cond [(empty? lst) base]
        [else (f (first lst)
                 (my-foldr f base (rest lst)))]))

(check-expect (my-foldr + 0 '(1 2 3 4 5)) 15)

(check-expect
 (my-foldr (lambda (x y) (and x y)) true
           (list true false true false true))
 false)
```

Producing lists with `foldr`

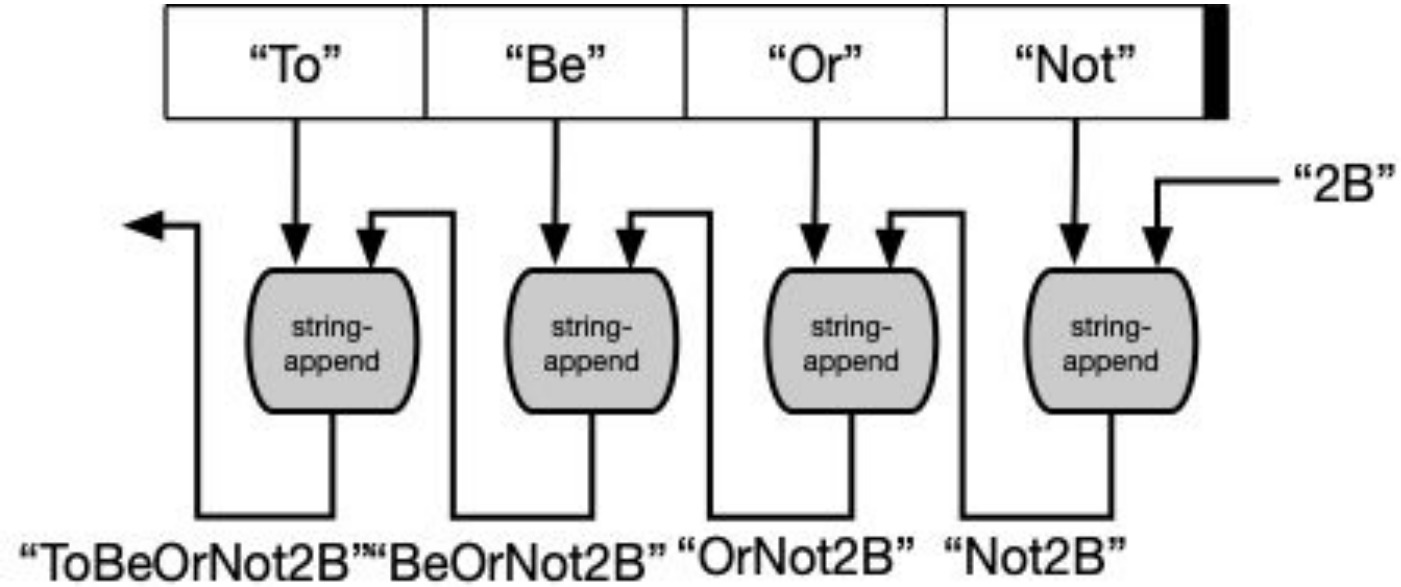
The `foldr` function can combine elements of a list to produce any values of any type, including lists.

```
foldr: (X Y -> Y) X (listof X) -> Y
```

`Y` can be any type, including `(listof Z)`.

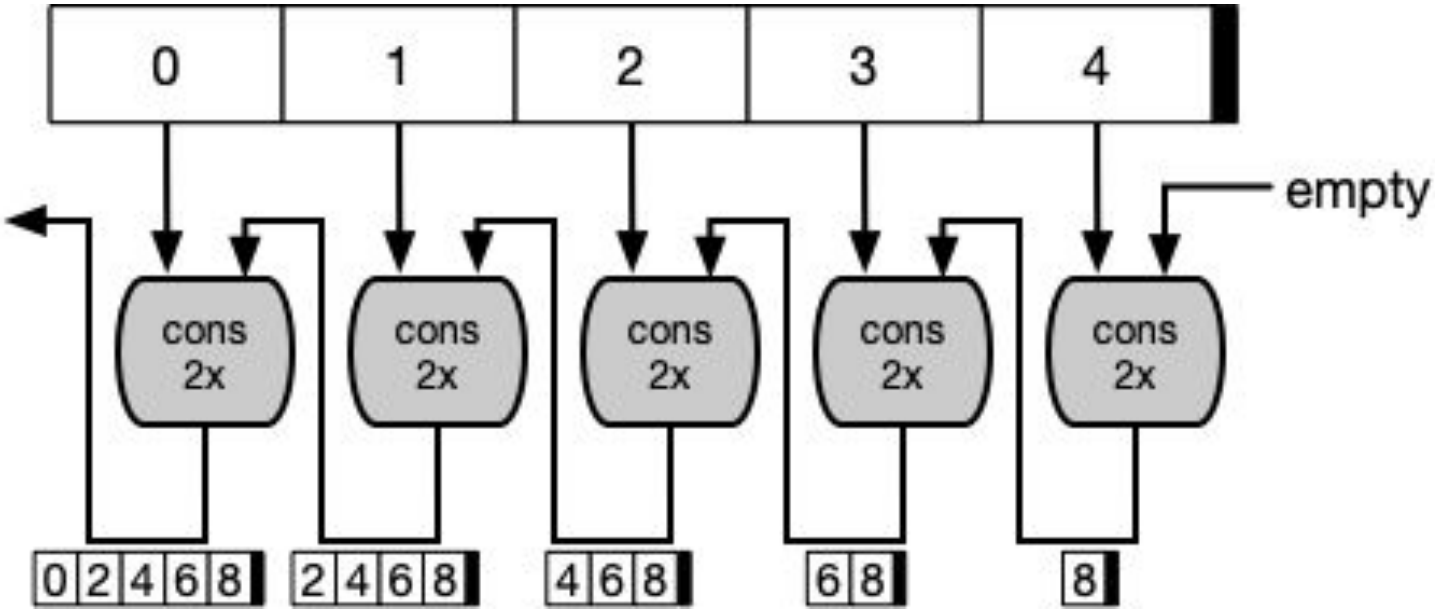
```
(foldr string-append "" '("To" "Be" "Or" "Not" "2B"))
    ⇒ "ToBeOrNot2B"

(foldr (lambda (x y) (cons (* 2 x) y)) empty '(0 1 2 3 4))
    ⇒ '(0 2 4 6 8)
```

Visualizing `foldr`




```
(foldr string-append "2B" '("To" "Be" "Or" "Not"))
```

Visualizing `foldr`




```
(foldr (lambda (x y) (cons (* 2 x) y)) empty '(0 1 2 3 4))
```

Combining higher order functions

Two or more abstract list functions can be used together to accomplish a task.

```
;; sum: Nat (Nat -> X) -> (listof X)
(define (sum n f)
  (foldr + 0 (build-list n f)))

(sum 4 sqr)
    ⇒ (foldr + 0 (build-list 4 sqr))
    ⇒ (foldr + 0 (list 0 1 4 9))
    ⇒ 14
```

Many assignment and exam questions will require you to combine two or more higher order functions.

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

The `foldr` function is powerful and general. For example, we can use it to implement our own versions of `filter` and `map`.

```
(define (my-filter ? lst)
  (foldr (lambda (x y) (cond [(? x) (cons x y)] [else y]))
         empty lst))
(my-filter even? '(1 2 3 4 5 6)) ⇒ '(2 4 6)

(define (my-map f lst)
  (foldr (lambda (x y) (cons (f x) y)) empty lst))
(my-map add1 '(0 1 2 3 4)) ⇒ '(1 2 3 4 5)
```

L19.3 `foldl`

foldl

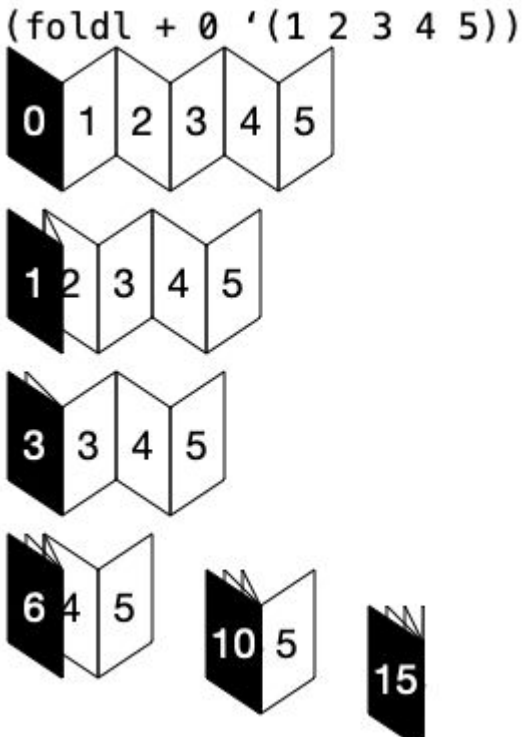
The `foldl` function recursively combines elements of a list **left to right**.

```
foldl: (X Y -> Y) X (listof X) -> Y
```

Starting with the base case and the last element, the function `f` is applied to the each element of the list along with the result of the previous application.

$$(\texttt{foldl f base (list } x_1\ x_2\ \ldots\ x_n\texttt{)}) \Rightarrow (\texttt{f } x_n\ \ldots\ (\texttt{f } x_2\ (\texttt{f } x_1\ \texttt{base})\ldots))$$

Remember this pattern!!!



`foldr` vs. `foldl`

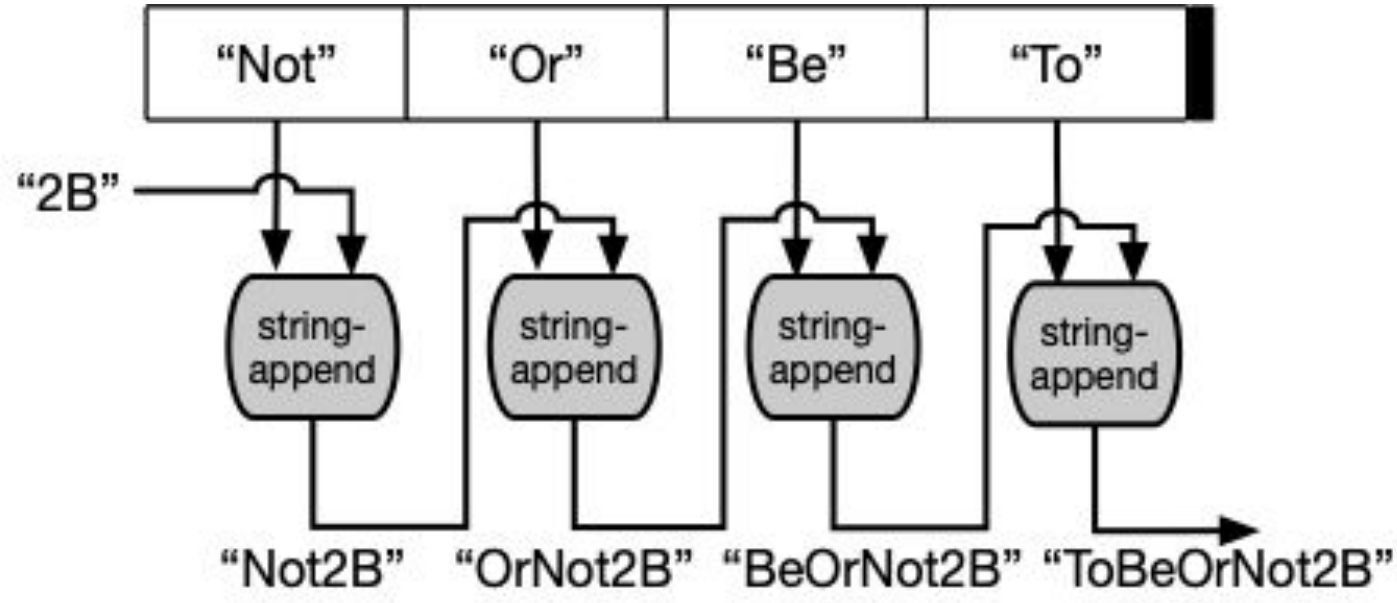
The `foldr` function recursively combines elements of a list **right to left**.

```
foldr: (X Y -> Y) X (listof X) -> Y

(foldr f base (list x1 x2 … xn))
          ⇒ (f x1 (f x2 … (f xn base)…))
```

The `foldl` function recursively combines elements of a list **left to right**.

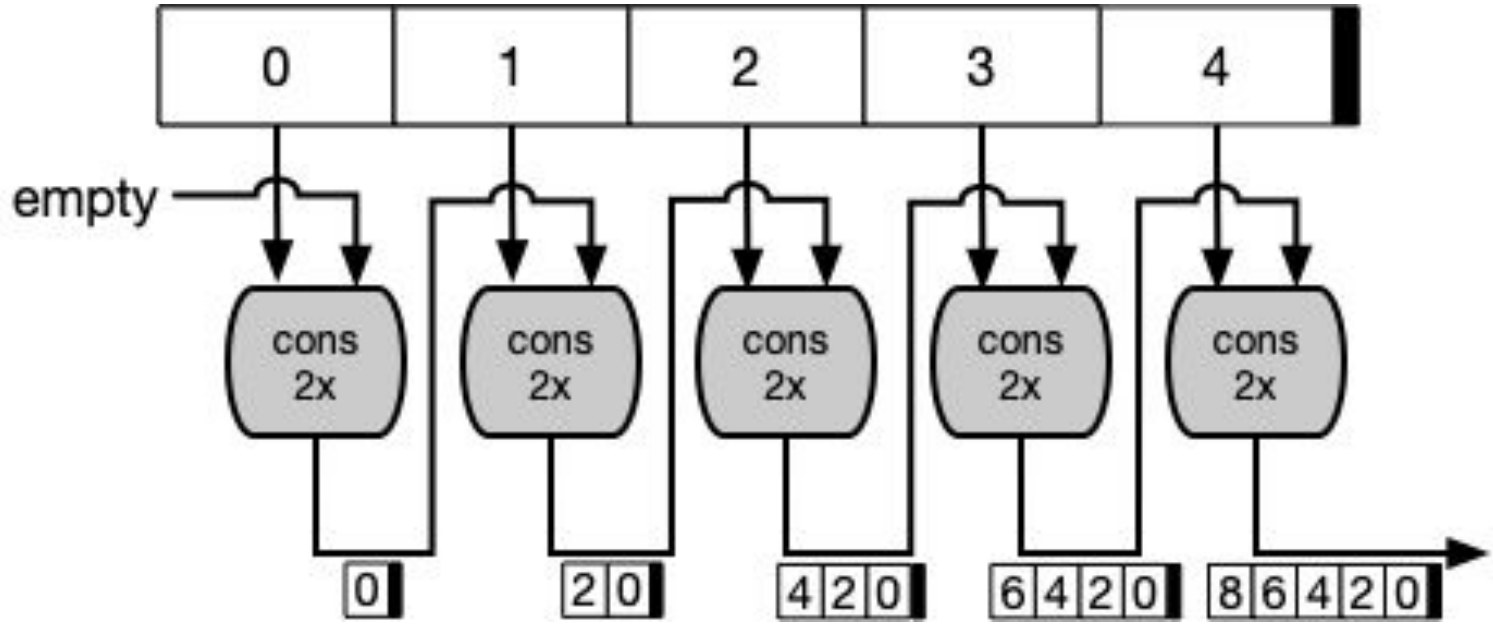
```
foldl: (X Y -> Y) X (listof X) -> Y

(foldl f base (list x1 x2 … xn))
          ⇒ (f xn … (f x2 (f x1 base)…))
```

Visualizing `foldl`




```
(foldl string-append "2B" '("Not" "Or" "Be" "To"))
```

Visualizing `foldl`




```
(foldl (lambda (x y) (cons (* 2 x) y)) empty '(0 1 2 3 4))
```

We can implement an equivalent function ourselves

```
(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)))

(check-expect
 (my-foldl string-append "2B" '("Not" "Or" "Be" "To"))
 "ToBeOrNot2B")
```

The `foldr` function generalizes accumulative recursion.

L19 Summary

L19: You should know

How to use and combine our five abstract list functions:

`(filter pred? lst)`	Retain only those elements of a list for which `pred?` is `true`.
`(build-list n f)`	Construct a list by applying `f` to the numbers `0` to `(- n 1)`.
`(map f lst)`	Construct a new list by applying `f` to each element of a list.
`(foldr f base lst)`	Recursively combine (“fold”) elements of a list right to left.
`(foldl f base lst)`	Recursively combine (“fold”) elements of a list left to right.

L19: Allowed constructs

Newly allowed constructs:

```
build-list foldl foldr map string-append
```

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ; '
abs acos add1 and append asin atan char? char=? char<?
check-expect check-within cond cons cons? cos define e else
empty empty? even? exp expt false filter first inexact?
integer? lambda length list list? local log max min not
number? odd? or pi quotient rational? remainder rest reverse
second sin sort sqr sqrt string? string=? string<?
string->list list->string string-append sub1 symbol?
symbol=? tan third true zero?
listof Any anyof Atom Bool Char Int Nat Num Rat Str Sym
```