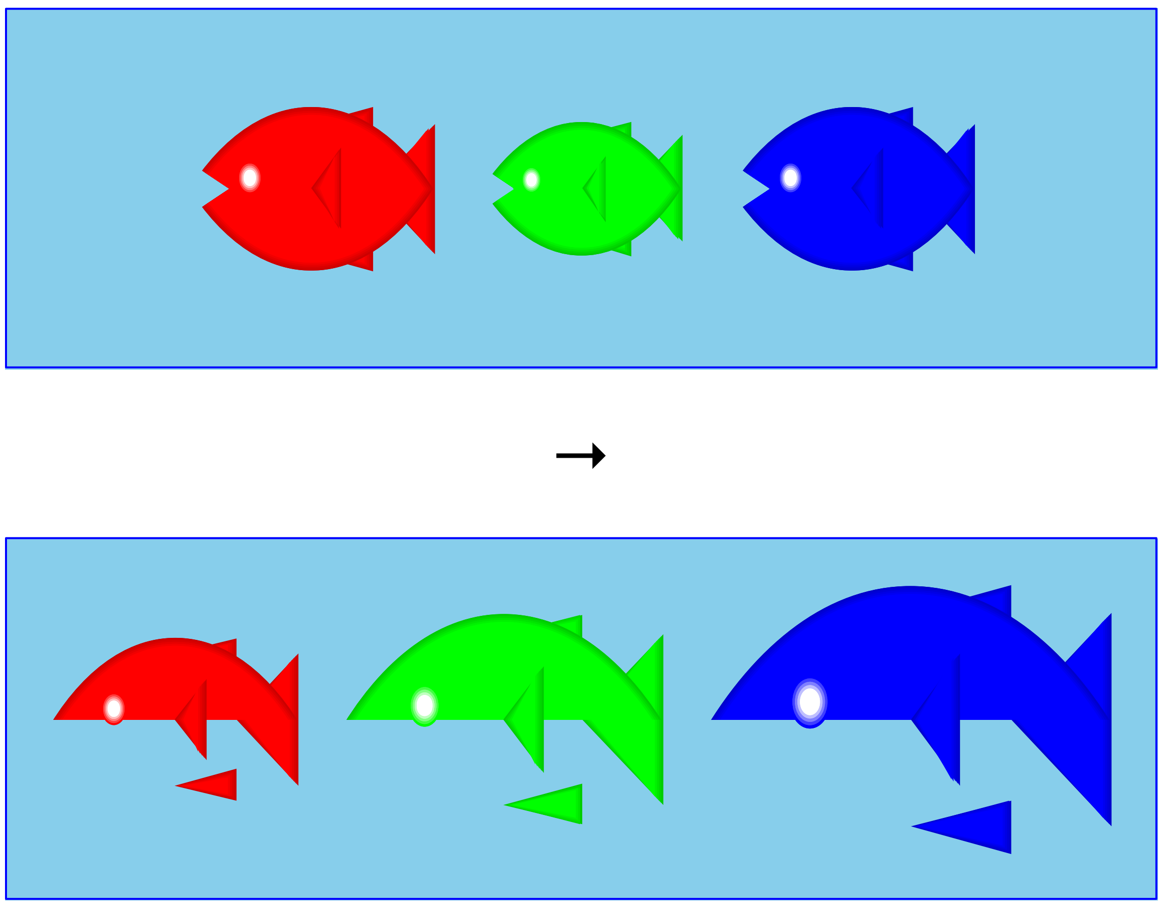
The Food Chain

Implement the function **`food-chain`** which takes a list of fish and returns a list of fish where each has eaten all of the fish to the left



The Food Chain

Implement the function **food-chain** which takes a list of fish and returns a list of fish where each has eaten all of the fish to the left

```
(food-chain '(3 2 3))
```

→

```
'(3 5 8)
```

Implementing the Food Chain

```
(define (food-chain l)
  (cond
   [(empty? l) ...]
   [else
    ... (first l)
    ... (food-chain (rest l)) ...]))
```

Is the result of `(food-chain '(2 3))` useful for getting the result of `(food-chain '(3 2 3))`?

```
(food-chain '(3 2 3))
→ ... 3 ... (food-chain '(2 3)) ...
→ ... 3 ... '(2 5) ...
→ → '(3 5 8)
```

Implementing the Food Chain

Feed the first fish to the rest, then **cons**:

```
(define (food-chain l)
  (cond
   [(empty? l) empty]
   [else
    (cons (first l)
          (feed-fish (food-chain (rest l))
                     (first l)))]))

(define (feed-fish l n)
  (cond
   [(empty? l) empty]
   [else (cons (+ n (first l))
               (feed-fish (rest l) n))]))
```

The Cost of the Food Chain

How long does `(feed-fish l)` take when `l` has n fish?

```
(define (food-chain l)
  (cond
   [(empty? l) empty]
   [else
    (cons (first l)
          (feed-fish (food-chain (rest l))
                     (first l)))]))
```

$$\boldsymbol{T}(0) = k_1$$

$$\boldsymbol{T}(n) = k_2 + \boldsymbol{T}(n\text{-}1) + \boldsymbol{S}(n\text{-}1)$$

where $\boldsymbol{S}(n)$ is the cost of `feed-fish`

The Cost of the Food Chain with feed-fish

$$\boldsymbol{T}(0) = k_1$$
$$\boldsymbol{T}(n) = k_2 + \boldsymbol{T}(n\text{-}1) + \boldsymbol{S}(n\text{-}1)$$

```
(define (feed-fish l n)
  (cond
   [(empty? l) empty]
   [else (cons (+ n (first l))
               (feed-fish (rest l) n))]))
```

$$\boldsymbol{S}(0) = k_3$$
$$\boldsymbol{S}(n) = k_4 + \boldsymbol{S}(n\text{-}1)$$

Overall, **S**(n) is proportional to n

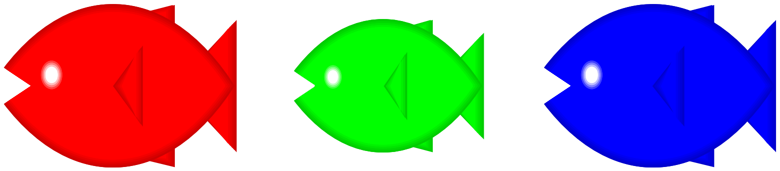
T(n) is proportional to n^2

How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Real fish:

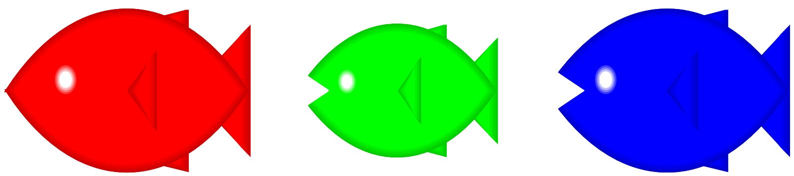


How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Real fish:

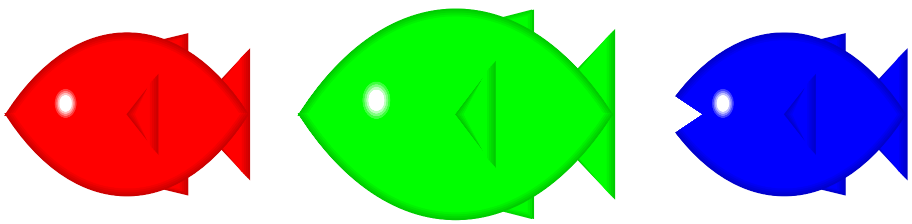


How Much a Food Chain should Cost

With 100 fish, our `food-chain` takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Real fish:

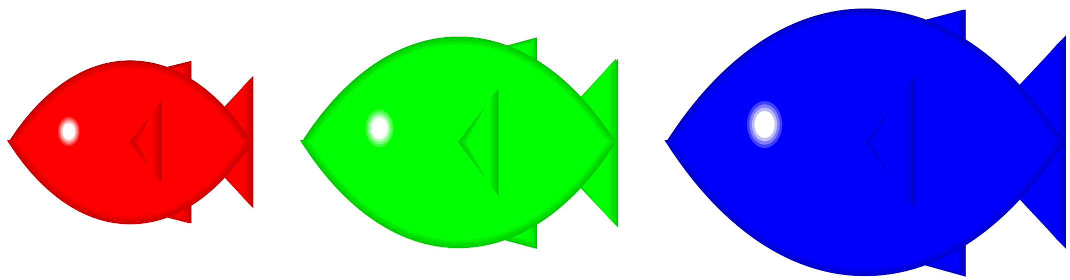


How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Real fish:

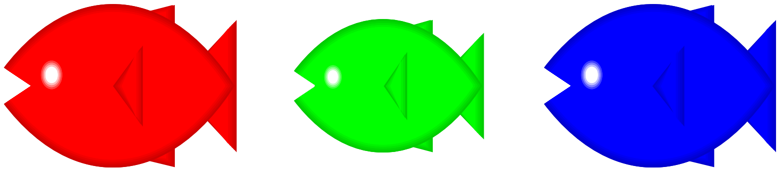


How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Our algorithm:

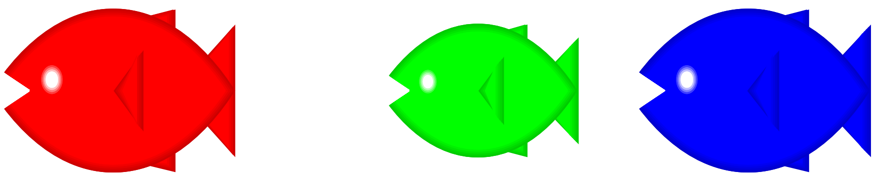


How Much a Food Chain should Cost

With 100 fish, our `food-chain` takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Our algorithm:

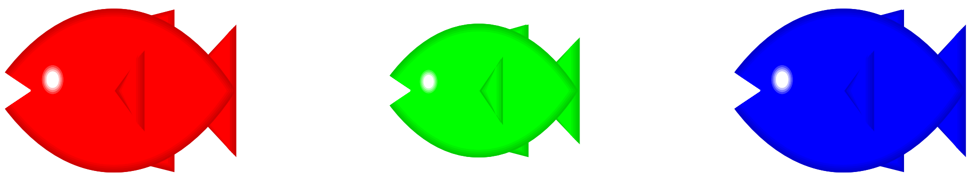


How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Our algorithm:

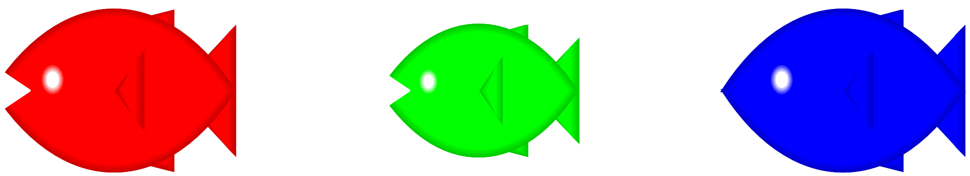


How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Our algorithm:

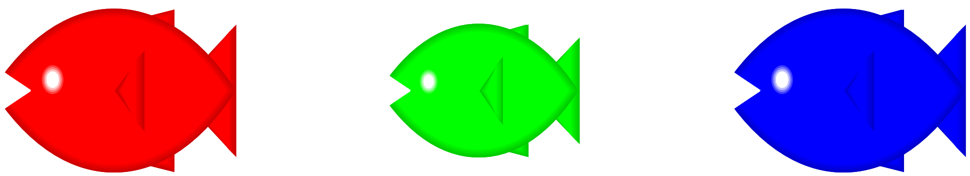


How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Our algorithm:

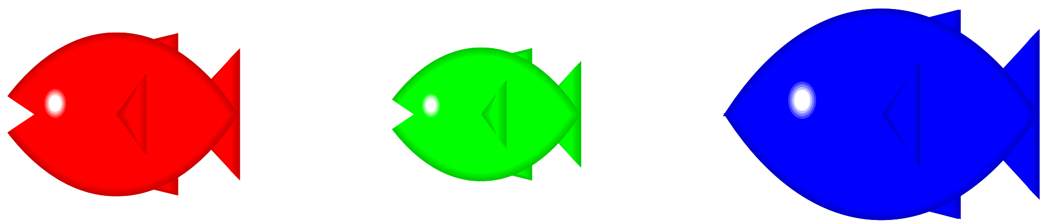


How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Our algorithm:

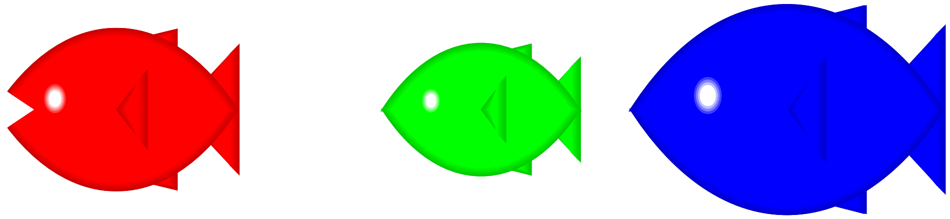


How Much a Food Chain should Cost

With 100 fish, our `food-chain` takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Our algorithm:

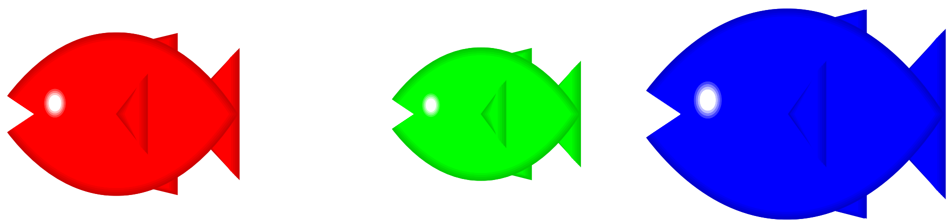


How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Our algorithm:

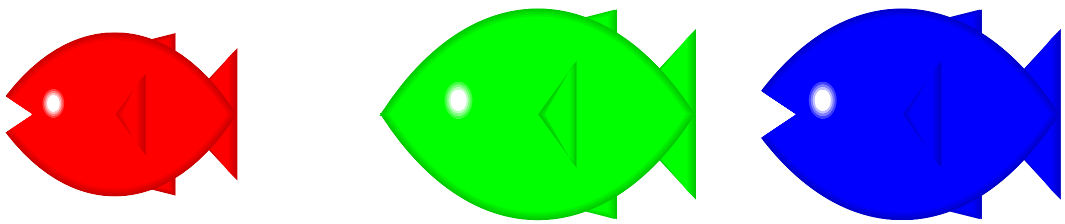


How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Our algorithm:

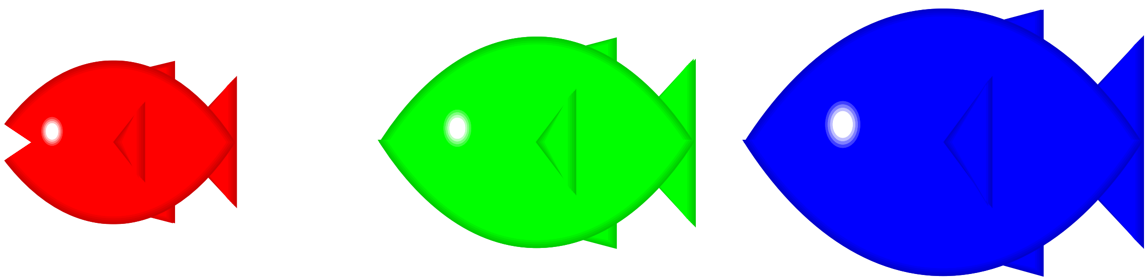


How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

Our algorithm:

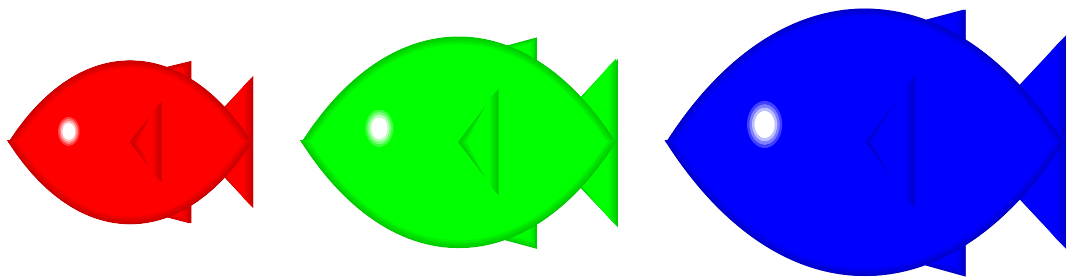


How Much a Food Chain should Cost

With 100 fish, our **`food-chain`** takes 10,000 steps to feed all the fish

Real fish are clearly more efficient!

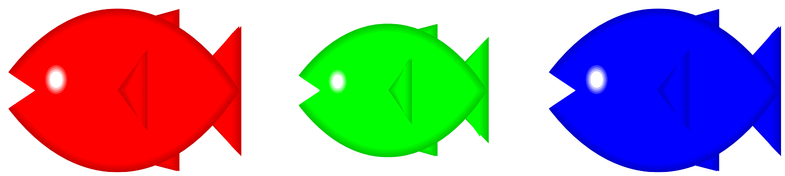
Our algorithm:



Practical Feeding

With real fish, eating ***accumulates*** a bigger fish while progressing up the chain:

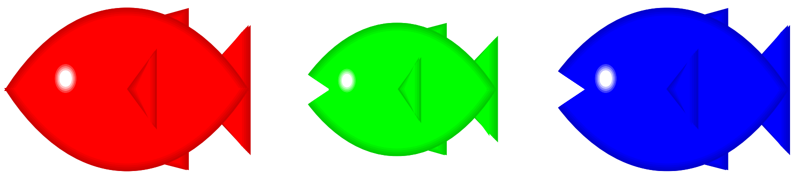
Real fish:



Practical Feeding

With real fish, eating ***accumulates*** a bigger fish while progressing up the chain:

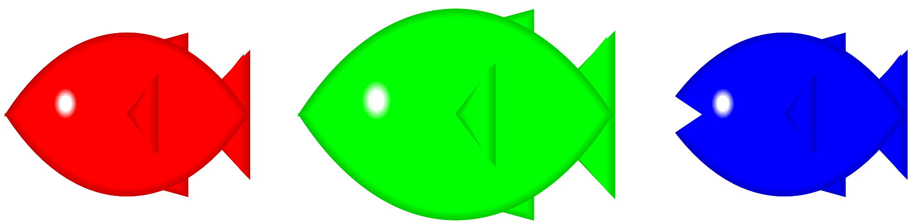
Real fish:



Practical Feeding

With real fish, eating ***accumulates*** a bigger fish while progressing up the chain:

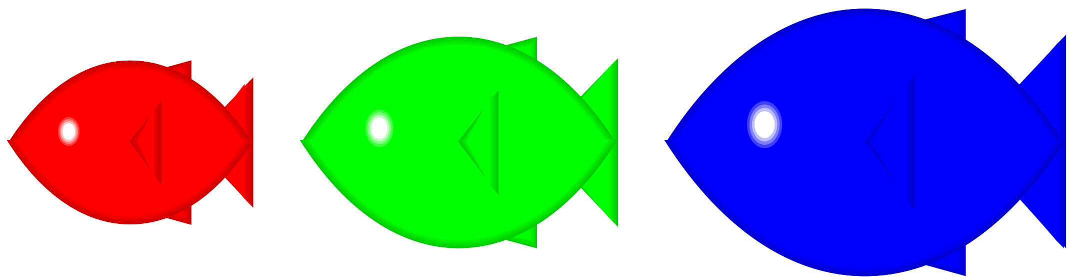
Real fish:



Practical Feeding

With real fish, eating ***accumulates*** a bigger fish while progressing up the chain:

Real fish:



Let's imitate this in our function

```
; food-chain-on
;    : list-of-num num -> list-of-num
; Feeds fish in l to each other,
; starting with the fish so-far
(define (food-chain-on l so-far) ...)
```

Accumulating Food

```
(define (food-chain-on l so-far)
  (cond
   [(empty? l) empty]
   [else
    (cons (+ so-far (first l))
          (food-chain-on
           (rest l)
           (+ so-far (first l))))]))

(define (food-chain l)
  (food-chain-on l 0))
```

```
(food-chain '(3 2 3))
→
(food-chain-on '(3 2 3) 0)
```

Accumulating Food

```
(define (food-chain-on l so-far)
  (cond
   [(empty? l) empty]
   [else
    (cons (+ so-far (first l))
          (food-chain-on
           (rest l)
           (+ so-far (first l))))]))

(define (food-chain l)
  (food-chain-on l 0))
```

```
(food-chain-on '(3 2 3) 0)
→ →
(cons 3 (food-chain-on '(2 3) 3))
```

Accumulating Food

```
(define (food-chain-on l so-far)
  (cond
   [(empty? l) empty]
   [else
    (cons (+ so-far (first l))
          (food-chain-on
           (rest l)
           (+ so-far (first l))))]))

(define (food-chain l)
  (food-chain-on l 0))
```

```
(cons 3 (food-chain-on '(2 3) 3))
→ →
(cons 3 (cons 5 (food-chain-on '(3) 5)))
```

Accumulating Food

```
(define (food-chain-on l so-far)
  (cond
   [(empty? l) empty]
   [else
    (cons (+ so-far (first l))
          (food-chain-on
           (rest l)
           (+ so-far (first l))))]))

(define (food-chain l)
  (food-chain-on l 0))
```

```
(cons 3 (cons 5 (cons 8 (food-chain-on empty 8))))
→ →
(cons 3 (cons 5 (cons 8 empty)))
```

Accumulators

```
(define (food-chain-on l so-far)
  (cond
   [(empty? l) empty]
   [else
    (cons (+ so-far (first l))
          (food-chain-on
           (rest l)
           (+ so-far (first l))))]))
```

The `so-far` argument of `food-chain-on` code is an ***accumulator***

The Direction of Information

With structural recusion, information from deeper in the structure is returned to computation shallower in the structure

```
(define (fun-for-loX l)
  (cond
   [(empty? l) ...]
   [else
    ... (first l)
    ... (fun-for-loX (rest l)) ...]))
```

The Direction of Information

An accumulator sends information the other way —
from shallower in the structure to deeper

```
(define (acc-for-loX l accum)
  (cond
   [(empty? l) ...]
   [else
    ... (first l) ... accum ...
    ... (acc-for-loX
         (rest l)
         ... accum ... (first l) ...)
    ...]))
```

Another Example: Reversing a List

Implement **`reverse-list`** which takes a list and returns a new list with the same items in reverse order

Pretend that **`reverse`** isn't built in

```
; reverse-list : list-of-X -> list-of-X

(check-expect (reverse-list empty) empty)
(check-expect (reverse-list '(a b c)) '(c b a))
```

Implementing Reverse

Using the template:

```
(define (reverse-list l)
  (cond
   [(empty? l) empty]
   [else
    ... (first l) ...
    ... (reverse-list (rest l)) ...]))
```

Is `(reverse-list '(b c))` useful for computing `(reverse-list '(a b c))`?

Yes: just add `'a` to the end

Implementing Reverse

```
(define (reverse-list l)
  (cond
   [(empty? l) empty]
   [else
    (snoc (first l)
          (reverse-list (rest l)))]))

(define (snoc a l)
  (cond
   [(empty? l) (list a)]
   [else
    (cons (first l)
          (snoc a (rest l)))]))

(check-expect (snoc 'a '(c b)) '(c b a))
```

The Cost of Reversing

How long does `(reverse l)` take when `l` has *n* items?

```
(define (reverse-list l)
  (cond
   [(empty? l) empty]
   [else
    (snoc (first l)
          (reverse-list (rest l)))]))
```

This is just like the old `food-chain` —
it takes time proportional to n^2

Reversing More Quickly

```
(reverse-list '(a b c))
→ →
(snoc 'a (reverse-list '(b c)))
→ →
(snoc 'a '(c b))
...
```

We could avoid the expensive `snoc` step if only we knew to start the result of `(reverse-list '(c b))` with `'(a)` instead of `empty`

Reversing More Quickly

```
(reverse-list '(a b c))
→ →
(reverse-onto '(b c) '(a))
...
```

It looks like we'll just run into the same problem with `'b` next time around...

Reversing More Quickly

```
(reverse-list '(a b c))
→ →
(reverse-onto '(b c) '(a))
→ →
(snoc 'b (reverse-onto '(c) '(a)))
???
```

But this isn't right anyway: `'b` is supposed to go before `'a`

Really we should reverse `'(c)` onto `'(b a)`

Reversing More Quickly

```
(reverse-list '(a b c))
→ →
(reverse-onto '(b c) '(a))
→ →
(reverse-onto '(c) '(b a))
...
```

And the starting point is that we reverse onto **`empty`**...

Reversing More Quickly

```
(reverse-list '(a b c))
→
(reverse-onto '(a b c) empty)
→ →
(reverse-onto '(b c) '(a))
→ →
(reverse-onto '(c) '(b a))
→ →
(reverse-onto empty '(c b a))
→ →
'(c b a)
```

The second argument to `reverse-onto` ***accumulates*** the answer

Accumulator-Style Reverse

```
; reverse-onto :
;   list-of-X list-of-X -> list-of-X
(define (reverse-onto l base)
  (cond
   [(empty? l) base]
   [else (reverse-onto (rest l)
                       (cons (first l)
                             base))]))

(define (reverse-list l)
  (reverse-onto l empty))
```

Foldl

Remember **foldr**, which is an abstraction of the template?

The pure accumulator version is **foldl**:

```
; foldl : (X Y -> Y) Y list-of-X -> Y
(define (foldl ACC accum l)
  (cond
   [(empty? l) accum]
   [else (foldl ACC
                (ACC (first l) accum)
                (rest l))]))

(define (reverse-list l)
  (foldl cons empty l))
```