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

\[
\begin{align*}
\text{(define } & (\text{food-chain } l) \\
& (\text{cond} \\
& \quad [(\text{empty? } l) \ldots] \\
& \quad \text{[else} \\
& \quad \quad \ldots (\text{first } l) \\
& \quad \quad \ldots (\text{food-chain } (\text{rest } l)) \ldots])]\end{align*}
\]

Is the result of \((\text{food-chain } '(2 3))\) useful for getting the result of \((\text{food-chain } '(3 2 3))\)?

\[
\begin{align*}
(\text{food-chain } '(3 2 3)) \\
\rightarrow \ldots 3 \ldots (\text{food-chain } '(2 3)) \ldots \\
\rightarrow \ldots 3 \ldots '(2 5) \ldots \\
\rightarrow \rightarrow '(3 5 8)
\end{align*}
\]
Implementing the Food Chain

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

```scheme
(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 \((\text{feed-fish } l)\) take when \(l\) has \(n\) fish?

\[
(\text{define } (\text{food-chain } l) \\
(\text{cond} \\
\quad [(\text{empty? } l) \ \text{empty}] \\
\quad [\text{else} \\
\quad \quad (\text{cons} \ (\text{first } l) \\
\quad \quad \quad (\text{feed-fish} \ (\text{food-chain} \ (\text{rest } l)) \\
\quad \quad \quad \quad (\text{first } l)))])))
\]

\[
\begin{align*}
T(0) &= k_1 \\
T(n) &= k_2 + T(n-1) + S(n-1)
\end{align*}
\]

where \(S(n)\) is the cost of \(\text{feed-fish}\)
The Cost of the Food Chain with feed-fish

\[ T(0) = k_1 \]
\[ T(n) = k_2 + T(n-1) + S(n-1) \]

\[(\text{define (feed-fish l n)}\]
\[\text{(cond}\]
\[\ [\text{(empty? l) empty]}\]
\[\ [\text{else (cons (+ n (first l))}\]
\[\ \text{(feed-fish (rest l) n)))]])\]

\[ S(0) = k_3 \]
\[ S(n) = k_4 + S(n-1) \]

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 \textit{food-chain} takes 10,000 steps to feed all the fish

\textbf{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:**

![Fish Diagram]
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

(defun (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))))]))

(defun (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))
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 recursion, information from deeper in the structure is returned to computation shallower in the structure

\[
\begin{align*}
\text{(define } & (\text{fun-for-loX } l) \\
\text{(cond } & \\
\text{ [(empty? } l) & \ldots] \\
\text{ [else } & \\
\text{ \ldots (first } l & \\
\text{ \ldots (fun-for-loX } (\text{rest } l)) & \ldots])\text{)}
\end{align*}
\]
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 \texttt{reverse-list} which takes a list and returns a new list with the same items in reverse order

Pretend that \texttt{reverse} isn’t built in

\begin{verbatim}
; reverse-list : list-of-X \rightarrow list-of-X

(check-expect (reverse-list empty) empty)
(check-expect (reverse-list '(a b c)) '(c b a))
\end{verbatim}
Implementing Reverse

Using the template:

```scheme
(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 \texttt{(reverse 1)} take when \texttt{1} has \texttt{n} items?

\begin{verbatim}
(define (reverse-list l)
  (cond
    [(empty? l) empty]
    [else
      (snoc (first l)
        (reverse-list (rest l)))]))
\end{verbatim}

This is just like the old \texttt{food-chain} —
it takes time proportional to \texttt{n²}
Reversing More Quickly

\[(\text{reverse-list } '(a \ b \ c))\]
\[\rightarrow \rightarrow\]
\[(\text{snoc } 'a \ (\text{reverse-list } '(b \ c)))\]
\[\rightarrow \rightarrow\]
\[(\text{snoc } 'a \ '(c \ b))\]
\[
\ldots
\]

We could avoid the expensive \textbf{snoc} step if only we knew to start the result of \[(\text{reverse-list } '(c \ b))\] 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

\[(\text{reverse-list} \ (a \ b \ c))\]
\[\rightarrow \rightarrow\]
\[(\text{reverse-onto} \ (b \ c) \ (a))\]
\[\rightarrow \rightarrow\]
\[(\text{snoc} \ b \ (\text{reverse-onto} \ (c) \ (a)))\]

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

Really we should reverse ' (c) onto ' (b a)
Reversing More Quickly

\[(\text{reverse-list} \ ' (a \ b \ c))\]
\[\rightarrow \rightarrow\]
\[(\text{reverse-onto} \ ' (b \ c) \ ' (a))\]
\[\rightarrow \rightarrow\]
\[(\text{reverse-onto} \ ' (c) \ ' (b \ a))\]
\[\ldots\]

And the starting point is that we reverse onto \textit{empty}...
Reversing More Quickly

\[(\text{reverse-list } '(a\ b\ c))\]
\[→\]
\[(\text{reverse-onto } '(a\ b\ c)\ \text{empty})\]
\[→ \ →\]
\[(\text{reverse-onto } '(b\ c)\ '(a))\]
\[→ \ →\]
\[(\text{reverse-onto } '(c)\ '(b\ a))\]
\[→ \ →\]
\[(\text{reverse-onto empty } '(c\ b\ a))\]
\[→ \ →\]
\['(c\ b\ a)\]

The second argument to \text{reverse-onto} \textit{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 \texttt{foldr}, which is an abstraction of the template?

The pure accumulator version is \texttt{foldl}:

\begin{verbatim}
; 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))
\end{verbatim}