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 \texttt{cons}:

\begin{verbatim}
(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)]))
\end{verbatim}
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))))]))

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

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

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

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

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

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

\textbf{Real fish are clearly more efficient!}

\textbf{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!}

\textbf{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:**

![Fish diagram]
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 \textit{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 \texttt{food-chain} takes 10,000 steps to feed all the fish

\textbf{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

```scheme
; 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

```scheme
(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

\[
\text{(define (food-chain-on l so-far)}
\begin{cases}
\text{empty} & \text{if (empty? l)} \\
\text{else} & \text{(cons (+ so-far (first l)) (food-chain-on (rest l) (+ so-far (first l))))}
\end{cases}
\text{)}
\]

\[
\text{(define (food-chain l)}
\begin{cases}
\text{(food-chain-on l 0)} & \text{)}
\end{cases}
\]

\[
\text{(food-chain-on '(3 2 3) 0)}
\rightarrow \rightarrow
\begin{cases}
\text{(cons 3 (food-chain-on '(2 3) 3))} & \text{)}
\end{cases}
\]
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 recursion, 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 \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 -> 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:

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

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

This is just like the old \text{food-chain} — it takes time proportional to \(n^2\)
Reversing More Quickly

\[
\begin{align*}
(reverse\text{-}list &\ (a\ b\ c)) \\
\rightarrow &\ \rightarrow \\
(snoc &\ 'a\ (reverse\text{-}list\ (b\ c))) \\
\rightarrow &\ \rightarrow \\
(snoc &\ 'a\ '(c\ b)) \\
\ldots \\
\end{align*}
\]

We could avoid the expensive \texttt{snoc} step if only we knew to start the result of
\[
(reverse\text{-}list\ (c\ b))\text{ with }'(a)\text{ instead of empty}
\]
Reversing More Quickly

\[
\begin{align*}
(r\text{e\text-ac-lis\text-t} & \ (a \ b \ c)) \\
\rightarrow & \rightarrow \\
(r\text{e\text-ver\text-s\text-ont\text-to} & \ (b \ c) \ (a)) \\
\ldots
\end{align*}
\]

It looks like we’ll just run into the same problem with 'b next time around...
Reversing More Quickly

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

???

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

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

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

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

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

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}:

\[
; \text{foldl} : (X \ Y \rightarrow Y) \ Y \ \text{list-of-X} \rightarrow Y \\
(\text{define} \ (\text{foldl} \ \text{ACC} \ \text{accum} \ \text{l}) \\
(\text{cond} \\
\quad [(\text{empty?} \ \text{l}) \ \text{accum}] \\
\quad [\text{else} \ (\text{foldl} \ \text{ACC} \\
\quad \quad (\text{ACC} \ (\text{first} \ \text{l}) \ \text{accum}) \\
\quad \quad (\text{rest} \ \text{l}))]))) \\
(\text{define} \ (\text{reverse-list} \ \text{l}) \\
\quad (\text{foldl} \ \text{cons} \ \text{empty} \ \text{l}))
\]