The Food Chain

Implement the function \texttt{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)) ...)]))
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 \( \text{feed-fish} \ 1 \) take when \( l \) has \( n \) fish?
The Cost of the Food Chain

How long does \((\text{feed-fish } l)\) take when \(l\) has \(n\) fish?

\[
\begin{align*}
\text{(define (food-chain } l) \\
\quad \text{(cond} \\
\quad \quad \text{[(empty? } l) \text{ empty]} \\
\quad \quad \text{[else} \\
\quad \quad \quad \text{(cons (first } l) \\
\quad \quad \quad \quad \text{(feed-fish (food-chain (rest } l))} \\
\quad \quad \quad \quad \text{(first } l))))]))
\end{align*}
\]
The Cost of the Food Chain

How long does \((\text{feed-fish } 1)\) take when \(1\) has \(n\) fish?

\[
\text{(define (food-chain 1)}
\begin{align*}
&\text{(cond} \\
&\quad [(\text{empty? } 1) \text{ empty}] \\
&\quad [\text{else} \\
&\quad \quad (\text{cons} (\text{first } 1) \\
&\quad \quad (\text{feed-fish (food-chain (rest } 1)) \\
&\quad \quad (\text{first } 1)))]
\end{align*}
\)
\]

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

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

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

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

\[ S(0) = k_3 \]
\[ S(n) = k_4 + S(n-1) \]
The Cost of the Food Chain with feed-fish

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

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

\begin{center}
\includegraphics[scale=0.5]{fish.png}
\end{center}
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 \textit{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:**

![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:
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:

```
(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))
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

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

\[
\text{(define (food-chain \text{l})}
\text{(food-chain-on \text{l} 0))}
\]

\[
\text{(food-chain-on '}(3 2 3) 0)
\]
\[
\rightarrow\rightarrow
\]
\[
\text{(cons 3 (food-chain-on '}(2 3) 3))
\]
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

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

The \textbf{so-far} argument of \texttt{food-chain-on} code is an \textbf{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
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

```scheme
; 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)) ...)]))
```
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))`?
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 \((\text{reverse } l)\) take when \(l\) has \(n\) items?
The Cost of Reversing

How long does \((\text{reverse } l)\) take when \(l\) has \(n\) items?

\[
\text{(define (reverse-list l)} \nonumber
\begin{align*}
&\text{(cond} \nonumber\\
&\quad [(\text{empty? } l) \ \text{empty}] \nonumber\\
&\quad [\text{else} \nonumber\\
&\quad\quad (\text{snoc } (\text{first } l) \nonumber\\
&\quad\quad\quad (\text{reverse-list } (\text{rest } l)))] \nonumber
\end{align*}
\]

}
The Cost of Reversing

How long does \texttt{(reverse 1)} take when 1 has \(n\) 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 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

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

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

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

(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

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

The second argument to \textit{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 **foldr**, which is an abstraction of the template?

The pure accumulator version is **foldl**:

```scheme
; 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))
```