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

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

→

`'(3 5 8)"
Implementing the Food Chain

(\texttt{define (food-chain \textit{l})})
(\texttt{cond})
(\texttt{[(empty? \textit{l}) ...]})
(\texttt{[else}})
(\texttt{... (first \textit{l})})
(\texttt{... (food-chain (rest \textit{l})) ...]))

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

(\texttt{(food-chain '(3 2 3)})
\rightarrow ... 3 ...
(\texttt{food-chain '(2 3)}) ...
\rightarrow ... 3 ...
(\texttt{2 5}) ...
\rightarrow \rightarrow '(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 \((\text{feed-fish } l)\) take when \(l\) has \(n\) fish?

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

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

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

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

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

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

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 *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

```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

```
(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)
(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

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

The \text{so-far} argument of \text{food-chain-on} code is an \text{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) ...)
      ...])
```

39
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

(reverse-list empty) "should be" empty
(reverse-list '(a b c)) "should be" '(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))))])

(snoc 'a '(c b)) "should be" '(c b a)
The Cost of Reversing

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

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

This is just like the old \texttt{food-chain}—it takes time proportional to \textit{n}^2
\end{verbatim}
Reversing More Quickly

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

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

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

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: \text{'b} is supposed to go before \text{'a}

Really we should reverse \text{'(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 \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}