Today we will learn...

1. Revise general recursion patterns
2. Implement general recursion patterns
3. Refactor code to reduce code repetition
4. Refactor code to improve performance
Functional patterns:
Reduction
Concatenate two lists
Concatenate two lists

Implement function \((\text{append } 11 \ 12)\) that appends two lists together.

Spec

\[
(\text{check-equal?} \\
(\text{append} \ (\text{list} \ 1 \ 2) \ (\text{list} \ 3 \ 4)) \\
(\text{list} \ 1 \ 2 \ 3 \ 4))
\]
Concatenate two lists

Implement function \(\text{append } \text{list } 1 \text{ 12}\) that appends two lists together.

Spec

\[
(\text{check-equal?}
\quad (\text{append } (\text{list } 1 \text{ 2}) (\text{list } 3 \text{ 4}))
\quad (\text{list } 1 \text{ 2 } 3 \text{ 4}))
\]

Solution

\[
(\text{define } (\text{append } \text{list } 1 \text{ 12})
\quad (\text{cond } [[(\text{empty? } \text{list } 1) \text{ 12}]
\quad [(\text{else } (\text{cons } (\text{first } \text{list } 1) (\text{append } (\text{rest } \text{list } 1) \text{ 12}))])])
\]

Is it tail recursive?
Concatenate two lists

Implement function \((\text{append} \ 11 \ 12)\) that appends two lists together.

Spec

\[
(\text{check-equal?}
\begin{align*}
& (\text{append} \ (\text{list} \ 1 \ 2) \ (\text{list} \ 3 \ 4)) \\
& (\text{list} \ 1 \ 2 \ 3 \ 4))
\end{align*}
\]

Solution

\[
(\text{define} \ (\text{append} \ 11 \ 12)
\begin{align*}
& (\text{cond} \ [\ (\text{empty?} \ 11) \ 12] \\
& \hspace{1em} \text{[else} \ (\text{cons} \ (\text{first} \ 11) \ (\text{append} \ (\text{rest} \ 11) \ 12))])
\end{align*}
\]

Is it tail recursive? **No!**
Generalizing reduction
A pattern arises

; Example 1:
(define (map f l)
  (cond [(empty? l) empty] [else (cons (f (first l)) (map f (rest l)))]))

; Example 2:
(define (filter to-keep? l)
  (cond [(empty? l) empty] [else (cond [(to-keep? (first l)) (cons (first l) (filter to-keep? (rest l)))] [else (filter to-keep? (rest l))])]))

; Example 3:
(define (append l1 l2)
  (cond [(empty? l1) l2] [else (cons (first l1) (append (rest l1) l2))]))
A pattern arises

### Example 1:

```scheme
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l)) (map f (rest l)))]))
```

### Example 2:

```scheme
(define (filter to-keep? l)
  (cond [(empty? l) empty]
        [else (cond [(to-keep? (first l))
                     (cons (first l) (filter to-keep? (rest l)))]
                   [else (filter to-keep? (rest l))])]))
```

### Example 3:

```scheme
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else (cons (first l1) (append (rest l1) l2))]))
```

---

General recursion pattern for lists

```scheme
(define (rec l)
  (cond [(empty? l) base-case]
        [else (step (first l) (rec (rest l)))]))
```

For instance,

```scheme
(cons (f (first l)) (map f (rest l)))
```

maps to

```scheme
(step (first l) (rec (rest l)))
```
Implementing this recursion pattern

Recursive pattern for lists

(define (rec l)
  (cond
   [(empty? l) base-case]
   [else (step (first l)
               (rec (rest l)))]))

Fold right reduction

(define (foldr step base-case l)
  (cond
   [(empty? l) base-case]
   [else (step (first l)
               (foldr step base-case (rest l)))]))
Implementing map with foldr

\[
\text{define (map f l) = (cond \[\text{empty? l) empty] \\
\text{else (cons (f (first l)) (map f (rest l)))}}\]
Implementing map with foldr

\[
\text{(define } \text{(map } f \text{ l)} \\
\quad \text{(cond } \begin{cases} 
\text{empty? l) empty} \\
\text{else (cons } \begin{cases} 
\text{f (first l))} \\
\text{(map } f \text{ (rest l))} 
\end{cases} 
\end{cases}) \text{)])}
\]

Solution

\[
\text{(define } \text{(map } f \text{ l)} \\
\quad \text{(define } \text{(on-elem } \text{elem new-list)} \\
\quad \quad \text{(cons } \begin{cases} 
\text{f elem)} \\
\text{new-list) (foldr on-elem empty l))}
\end{cases} 
\end{cases} 
\text{)])}
\]
Implementing append with foldr

(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else (cons (first l1) (append (rest l1) l2))])))
Implementing append with foldr

(define (append l1 l2)
  (cond
   [[(empty? l1) l2]
    [else
     (cons (first l1)
       (append (rest l1) l2))]]))

Solution

(define (append l1 l2)
  (foldr cons l2 l1))
Implementing filter with foldr

```
(define (filter to-keep? l)
  (cond
    [(empty? l) empty]
    [else
     (cond
      [(to-keep? (first l))
       (cons (first l)
            (filter to-keep? (rest l)))]
      [else (filter to-keep? (rest l))])]))
```

Solution

```
(define (filter to-keep? l)
  (define (on-elem elem new-list)
    (cond
      [(to-keep? elem) (cons elem new-list)]
      [else new-list]))
  (foldr on-elem empty l))
```
Contrasting the effect of using foldr

; Example 1:
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                     (map f (rest l)))]))

; Example 2:
(define (filter to-keep? l)
  (cond [(empty? l) empty]
        [else (cond [(to-keep? (first l))
                     (cons (first l)
                           (filter to-keep? (rest l)))]
               [else (filter to-keep? (rest l))])]))

; Example 3:
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else (cons (first l1)
                    (append (rest l1) l2))]))
Contrasting the effect of using foldr

; Example 1:
(define (map f l)
  (cond [(empty? l) empty]
        [else (cons (f (first l))
                    (map f (rest l)))]))

; Example 2:
(define (filter to-keep? l)
  (cond [(empty? l) empty]
        [else (cond [(to-keep? (first l))
                    (cons (first l)
                          (filter to-keep? (rest l)))]
                  [else (filter to-keep? (rest l)))]))

; Example 3:
(define (append l2 l1)
  (cond [(empty? l1) l2]
        [else (cons (first l1)
                    (append (rest l1) l2)])])
What about the fold left reduction?
Reversing a list

Implement function \( (\text{reverse} \ 1) \) that reverses a list.

Spec

\[
(\text{check-equal?} \ (\text{list} \ 4 \ 3 \ 2 \ 1) \ (\text{reverse} \ (\text{list} \ 1 \ 2 \ 3 \ 4)))
\]
Reversing a list

Implement function $(\text{reverse } \text{l})$ that reverses a list.

Spec

$$(\text{check-equal?} \ (\text{list} \ 4 \ 3 \ 2 \ 1) \ (\text{reverse} \ (\text{list} \ 1 \ 2 \ 3 \ 4)))$$

Solution

$$(\text{define} \ (\text{reverse} \ \text{l})$$
$$\ (\text{define} \ (\text{rev} \ \text{l} \ \text{accum})$$
$$\ (\text{cond} \ [(\text{empty?} \ \text{l}) \ \text{accum}]$$
$$\ [\text{else} \ (\text{rev} \ (\text{rest} \ \text{l}) \ (\text{cons} \ (\text{first} \ \text{l}) \ \text{accum}))])$$
$$\ (\text{rev} \ \text{l} \ \text{empty}))$$
Another pattern arises

; Example 1
(define (concat-nums 1)
  (define (f n)
    (string-append " " (number->string n)))
  (define (concat-nums-aux accum l)
    (cond
      [(empty? l) accum]
      [else
        (concat-nums-aux
          (string-append accum (f (first l) (rest l))))]
    (concat-nums-aux ">" 1))
  )

; Example 2
(define (reverse 1)
  (define (rev accum l)
    (cond
      [(empty? l) accum]
      [else (rev (cons (first l) accum) (rest l))]
    (rev empty l)))

A generalized recursion pattern for lists

(define (rec base-case 1)
  (cond
    [(empty? 1) base-case]
    [else
     (rec (step (first 1) base-case))
    ])
  )

For instance,

(cons (first 1) accum)
maps to

(step (first 1) accum)
Implementing this recursion pattern

Recursive pattern for lists

```
(define (rec accum l)
  (cond
    [(empty? l) accum]
    [else
      (rec (step (first l) accum) (rest l))]))
```

Fold left reduction

```
(define (foldl step base-case l)
  (cond
    [(empty? l) base-case]
    [else (foldl step
      (step (first l) base-case) (rest l))]))
```
Implementing concat-nums with foldl

Before

```
(define (concat-nums l)
  (define (f n)
    (string-append " " (number->string n)))
  (define (concat-nums-aux accum l)
    (cond
     [(empty? l) accum]
     [else
      (concat-nums-aux
       (string-append accum (f (first l)))
       (rest l))]
    (concat-nums-aux "->" l))
```
Implementing concat-nums with foldl

Before

```scheme
(define (concat-nums l)
  (define (f n)
    (string-append " " (number->string n)))
  (define (concat-nums-aux accum l)
    (cond
      [(empty? l) accum]
      [else
        (concat-nums-aux
          (string-append accum (f (first l)))
          (rest l)))))
  (concat-nums-aux ">" l))
```

After

```scheme
(define (concat-nums l)
  (define (f n a)
    (string-append a " "
      (number->string n)))
  (foldl f ">" l))
```
Implementing reverse with foldl

Original

```scheme
(define (reverse l)
  (define (rev accum l)
    (cond [(empty? l) accum]
          [else (rev (cons (first l) accum)
                     (rest l))]))
  (rev empty l))
```

Implementing reverse with foldl

Original

```
(define (reverse l)
  (define (rev accum l)
    (cond [(empty? l) accum]
          [else (rev (cons (first l) accum) (rest l))])
  (rev empty l))
```

Solution

```
(define (reverse l)
  (foldl cons empty l))
```
Contrasting the effect of using foldl

Before

; Example 1
(define (concat-nums l)
 (define (f n)
   (string-append " " (number->string n)))
(define (concat-nums-aux accum l)
 (cond
   [(empty? l) accum]
   [else
    (concat-nums-aux
      (string-append accum (f (first l)))
      (rest l))]])
(concat-nums-aux "">" l))

; Example 2
(define (reverse l)
 (define (rev accum l)
   (cond [(empty? l) accum]
         [else (rev (cons (first l) accum) (rest l))]])
 (rev empty l))
Contrasting the effect of using foldl

Before

; Example 1
(define (concat-nums l)
  (define (f n)
    (string-append " " (number->string n)))
  (define (concat-nums-aux accum l)
    (cond
      [(empty? l) accum]
      [else
        (concat-nums-aux
          (string-append accum (f (first l)))
          (rest l))]))
  (concat-nums-aux "" l))

; Example 2
(define (reverse l)
  (define (rev accum l)
    (cond
      [(empty? l) accum]
      [else (rev (cons (first l) accum) (rest l))]))
  (rev empty l))

After

; Example 1
(define (concat-nums l)
  (define (f n a)
    (string-append a " "
      (number->string n)))
  (foldl f "" l))

; Example 2
(define (reverse l)
  (foldl cons empty l))
What about tail-recursive optimization?
What about tail-recursive optimization?

- We note that `fold1` is tail-recursive already
- However, our original implementation of `foldr` is not tail recursive

Can't we implement the tail-recursive optimization pattern?

**Unoptimized**

```scheme
(define (rec l)
  (cond
    [(empty? l) base-case]
    [else (step (first l) (rec (rest l)))]))
```

**Optimized**

```scheme
(define (rec l)
  (define (rec-aux accum l)
    (cond
      [(empty? l) (accum base-case)]
      [else
        (rec-aux
          (lambda (x) (accum (step (first l) x)))
          (rest l))])))
  (rec-aux (lambda (x) x) 1)
```
Optimized foldr

Generalized pattern

\[
\begin{align*}
\text{(define (rec l)} & \text{ (define (rec-aux aux accum l)} \\
& \text{(cond)} \\
& \text{[\text{(empty? l)} \text{ (accum base-case)}]} \\
& \text{[else]} \\
& \text{(rec-aux)} \\
& \text{\hspace{1em} (lambda (x))} \\
& \text{\hspace{1em} (accum \text{ (step (first l) x))}} \\
& \text{\hspace{1em} (rest l))])]} \\
\text{(rec-aux (lambda (x) x) l)}
\end{align*}
\]

Implementation

\[
\begin{align*}
\text{(define (foldr step base-case l)} & \text{ (define (foldr-iter aux accum l)} \\
& \text{(cond)} \\
& \text{[\text{(empty? l)} \text{ (accum base-case)}]} \\
& \text{[else]} \\
& \text{(foldr-iter)} \\
& \text{\hspace{1em} (lambda (x))} \\
& \text{\hspace{1em} (accum \text{ (step (first l) x))}} \\
& \text{\hspace{1em} (rest l))])]} \\
\text{(foldr-iter (lambda (x) x) l)}
\end{align*}
\]
Benchmark evaluation

- Unoptimized foldr
- Tail-recursive foldr

Processing a list of size: 1000000

Throughput (unopt): 7310 elems/ms
Mean (unopt): 136.8±7.56ms

Throughput (tailrec): 12349 elems/ms
Mean (tailrec): 80.98±1.49ms
Speed-up (tailrec): 1.7

A speed improvement of 1.7
What if we use foldl + reverse?
What if we use foldl + reverse?

- Instead of creating nested functions,
- We reverse the list and apply foldl

```
(define (foldr step base-case l)
  (foldl step base-case (reverse l)))
```
What if we use foldl + reverse?

- Instead of creating nested functions,
- We reverse the list and apply foldl

```
(define (foldr step base-case l)
  (foldl step base-case (reverse l)))
```

Simpler implementation!

But is it faster?
Rev+fold runs the slower (0.7)

Processing a list of size: 1000000

Throughput (unopt): 7310 elems/ms
Mean (unopt): 136.8±7.56ms

Throughput (tailrec): 12349 elems/ms
Mean (tailrec): 80.98±1.49ms
Speed-up (tailrec): 1.7

Throughput (rev+foldl): 4846 elems/ms
Mean (rev+foldl): 206.34±3.33ms
Speed-up (rev+foldl): 0.7
Conclusion

We learned to generalize two reduction patterns (foldl and foldr)

- **Pro:** generalizing code can lead to a central point to optimize code
- **Pro:** generalizing code can reduce our code base
  (less code means less code to maintain)
- **Con:** one level of indirection increases the cognitive code
  (more cognitive load, code harder to understand)

Easier to understand (self-contained)  Harder to understand (what is foldr?)

```
(define (map f 1)
  (cond [(empty? l) empty]
        [else (cons (f (first 1))
                    (map f (rest l))))])
```

```
(define (map f 1)
  (define (on-elem elem new-list)
    (cons (f elem) new-list))
  (foldr on-elem empty 1))
```