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
Appending two lists together

Implement function `(append l1 l2)` that appends two lists together.

Spec

```scheme
(check-equal?
 (append (list 1 2) (list 3 4))
 (list 1 2 3 4))
```
Appending two lists together

Implement function (append l1 l2) that appends two lists together.

Spec

(check-equal?
 (append (list 1 2) (list 3 4))
 (list 1 2 3 4))

Solution

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

Is it tail recursive?
Appending two lists together

Implement function (append 11 12) that appends two lists together.

Spec

```
(check-equal?
  (append (list 1 2) (list 3 4))
  (list 1 2 3 4))
```

Solution

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

Is it tail recursive? **No!**
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:
(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))]))

General recursion pattern for lists

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

For instance,

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

maps to

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

Recursive pattern for lists

\[
(\text{define } (rec \ l) \\
(\text{cond} \\
\quad [(\text{empty? } l) \text{ base-case}] \\
\quad [\text{else } \text{step} (\text{first } l) \\
\quad \quad (\text{rec } (\text{rest } l)))]))
\]

Fold right reduction

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

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

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

Solution

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

\[
\text{(define (append \textit{l1} \textit{l2})}
\begin{align*}
\text{(cond} & \begin{cases}
\text{(empty? \textit{l1})} & \text{\textit{l2}} \\
\text{else} & \begin{cases}
\text{(cons} & \begin{cases}
\text{(first \textit{l1})} & \text{(append \textit{rest l1} \textit{l2})]} \\
\end{cases} \\
\end{cases}
\end{cases}
\end{align*}
\)\]
Implementing append with foldr

\[
\begin{align*}
&\text{(define (append } l1 \ l2) \\
&\quad (\text{cond} [(\text{empty?} \ l1) \ l2] \\
&\quad \quad [\text{else} \ (\text{cons} (\text{first} \ l1) \ (\text{append} (\text{rest} \ l1) \ l2))]))
\end{align*}
\]

Solution

\[
\begin{align*}
&\text{(define (append} l1 \ l2) \\
&\quad (\text{foldr} \ \text{cons} \ l2 \ l1))
\end{align*}
\]
Implementing filter with foldr

\[
\begin{align*}
\text{(define (filter to-keep? l)} & \\
& \text{(cond)} & \\
& \text{[(empty? l) empty]} & \\
& \text{[else (cond [(to-keep? (first l))} & \\
& \text{ (cons (first l) (filter to-keep? (rest l)))] & \\
& \text{[else (filter to-keep? (rest l)))]}])))
\end{align*}
\]

Solution

\[
\begin{align*}
\text{(define (filter to-keep? l)} & \\
& \text{(define (on-elem elem new-list)} & \\
& \text{(cond [(to-keep? elem) (cons elem new-list)]} & \\
& \text{[else new-list]]}) & \\
& \text{foldr on-elem empty l))}
\end{align*}
\]
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 l1 l2)
  (cond [(empty? l1) l2]
        [else (cons (first l1)
                    (append (rest l1) l2))]))
What about the fold left reduction?
Reversing a list

Implement function `(reverse 1)` that reverses a list.

Spec

```scheme
(check-equal? (list 4 3 2 1) (reverse (list 1 2 3 4)))
```
Reversing a list

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

Spec

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

Solution

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

A generalized recursion pattern for lists

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

For instance,

```
(cons (first 1) accum)
```

maps to

```
(step (first 1) accum)
```
Recursive pattern for lists

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

Fold left reduction

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

```
(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

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

Solution

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

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 foldr

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 \texttt{foldl} is tail-recursive already
- However, our original implementation of \texttt{foldr} is not tail recursive

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

Unoptimized

\begin{verbatim}
(define (rec l)
  (cond
    [(empty? l) base-case]
    [else (step (first l) (rec (rest l)))]))
\end{verbatim}

Optimized

\begin{verbatim}
(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)
\end{verbatim}
Optimized foldr

Generalized pattern

(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) l))

Implementation

(define (foldr step base-case l)
  (define (foldr-iter accum l)
    (cond
      [(empty? l) (accum base-case)]
      [else
        (foldr-iter
          (lambda (x)
            (accum (step (first l) x)))
          (rest l))])
    (foldr-iter (lambda (x) x) l))
Benchmark evaluation

- Unoptimized foldr
- Tail-recursive foldr

Processing a list of size: 1000000

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

```scheme
(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

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

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

Throughoutput (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)

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

Harder to understand (what is foldr?)

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