Today we will learn...

- Reversing a list
- Joining strings with a separator
- Fold-left
- Implementing reverse/join with fold-left
- Benchmarking fold-right
Reversing a list
Reversing a list

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

Spec

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

Solution

\[
\text{(define (reverse l)} \\
\quad \text{(define (rev l accum)} \\
\quad \quad \text{(cond [(empty? l) accum]} \\
\quad \quad \quad \text{[else (rev (rest l) (cons (first l) accum))]}]) \\
\quad \text{(rev l empty))}
\]
Joining strings in Racket
Joining strings in Python

```python
>>> ",", join(["x", "y", "z"])
"x, y, z"
```

```python
>>> ",", join([])
"
```

```python
>>> ",", join(["x"])  # x
"x"
```
Joining strings in Racket

(require rackunit)
(check-equal? (join " " '("x" "y" "z")) "x, y, z")
(check-equal? (join " " '()) "")
(check-equal? (join " " '("x")) "x")
Joining strings in Racket (solution)

- Append prefix ", " after each element:
Joining strings in Racket (solution)

- Append prefix ", " after each element:
  "x, y, z, "

Joining strings in Racket (solution)

- Append prefix ", " after each element:
  "x, y, z, ",

- Append prefix after each element, but not on last:
Joining strings in Racket (solution)

- Append prefix ", " after each element:
  "x, y, z,"

- Append prefix after each element, but not on last:
  "x, y, " + "z"

How do you check the last?
Joining strings in Racket (solution)

- Append prefix ", " after each element:
  "x, y, z, "

- Append prefix after each element, but not on last:
  "x, y, " + "z"
  How do you check the last?

- Prepend prefix before each element:
Joining strings in Racket (solution)

- Append prefix ", " after each element:
  
  "x, y, z, "

- Append prefix after each element, but not on last:
  
  "x, y, " + "z"

  How do you check the last?

- Prepend prefix before each element:
  
  ", x, y, z"
Joining strings in Racket (solution)

- **Append prefix ", " after each element:**
  "x, y, z, "

- **Append prefix after each element, but not on last:**
  "x, y, " + "z"
  How do you check the last?

- **Prepend prefix before each element:**
  ", x, y, z"

- **Prepend prefix before each element, but do not prepend first:**
Joining strings in Racket (solution)

- **Append prefix ", " after each element:**
  "x, y, z, "

- **Append prefix after each element, but not on last:**
  "x, y, " + "z"
  How do you check the last?

- **Prepend prefix before each element:**
  ", x, y, z"

- **Prepend prefix before each element, but do not prepend first:**
  "x" + ", y, z"
  We're implementing this version, you'll see why.

```racket
(define (join sep 1)  
  (define (join-iter accum l)  
    (cond  
      [(empty? l) accum]  
      [else  
        (join-iter  
          (string-append accum sep (first l)  
          (rest l)))]))  
  (cond [(empty? l) ""]  
    [else  
      (join-iter (first l) (rest l))])))  

(import "rackunit")  
(join ", " "("x" "y" "z"))  
(join ", " "())  
(join ", " "("x"))
```
def join(sep, l):
    if l == []:
        return ""
    accum = l[0]
    l = l[1:]
    for x in l:
        accum = accum + sep + x
    return accum

l = ['x', 'y', 'z']
join(',', '', l)
'x, y, z'

(define (join sep l)
  (define (join-iter accum l)
    (cond
      [(empty? l) accum]
      [else
       (join-iter (string-append accum sep (first l)) (rest l))]))
  (cond [(empty? l) ""]
        [else
         (join-iter (first l) (rest l))])))
Another pattern arises

A generalized recursion pattern for lists

For instance,

\[
\text{(cons (first 1) accum)}
\]

maps to

\[
\text{(step (first 1) accum)}
\]
Implementing this recursion pattern

Recursive pattern for lists

\[
\text{(define (rec accum l)}
\text{ (cond}
\text{ [ (empty? l) accum]}
\text{ [else}
\text{ (rec (step (first l) accum)
\text{ (rest l)]])])}
\]

Fold left reduction

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

Before

```
(define (join sep l)
  (define (join-iter accum l)
    (cond
      [(empty? l) accum]
      [else
       (join-iter
        (string-append accum sep (first l)
         (rest l)))]))
  (cond [(empty? l) ""]
        [else
         (join-iter (first l) (rest l))]])
```
Implementing concat-nums with foldl

Before

(define (join sep l)
  (define (join-iter accum l)
    (cond [[(empty? l) accum]
          [else
            (join-iter
              (string-append accum sep (first l)
              (rest l)))]])
    (cond [[(empty? l) ""]
          [else
            (join-iter (first l) (rest l))]]))

After

(define (join sep l)
  (cond [[(empty? l) ""]
        [else
          (define (step elem accum)
            (string-append accum sep elem))
          (foldl step (first l) (rest l))]])

Python version suggested by Dakai Tzou:

from functools import reduce

def join(l, sep):
    if l == []:
        return l
    def step(elem, accum):
        return elem + sep + accum
    return reduce(step, l[1:], l[0])
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

(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)
  (define (on-elem elem accum)
    (cons elem accum))
  (foldl on-elem empty l))

or

(define (reverse l)
  (foldl cons empty l))
Optimizing fold-right
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 1)
  (cond
   [(empty? 1) base-case]
   [else (step (first 1) (rec (rest 1)))]))
\end{verbatim}

Optimized

\begin{verbatim}
(define (rec 1)
  (define (rec-aux accum 1)
    (cond
     [(empty? 1) (accum base-case)]
     [else (rec-aux (lambda (x)
                      (accum (step (first 1) x)))
                  (rest 1))]))
  (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) 1))
```

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) 1)))
```
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

\[
\text{define } (\text{foldr } \text{step base-case } l) \\
(\text{foldl } \text{step base-case (reverse } l))
\]
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

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

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