CS450

Structure of Higher Level Languages

Lecture 08: Reverse, join-strings, foldl, performance

Tiago Cogumbreiro

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 `(reverse l)` that reverses a list.

Spec

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

Reversing a list

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

Spec

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

Solution

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

Joining strings in Racket

Joining strings in Python

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

(import "rackunit")
(join ", " '("x" "y" "z"))
(join ", " '())
(join ", " '("x"))
```

Imperative versus functional

```python
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'
```

```scheme
(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

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

A generalized recursion pattern for lists

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

For instance,

```
(cons (first l) accum)
```

maps to

```
(step (first l) 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 (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

```racket
(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

```racket
(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:

```python
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 `foldl` is tail-recursive already
- However, our original implementation of `foldr` is not tail recursive

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

Unoptimized

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

Optimized

```
(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))
```

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

```
(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)

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

Harder to understand (what is foldr?)

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