CS450

Structure of Higher Level Languages

Lecture 07: Filter, append, fold

Tiago Cogumbreiro

Functional pattern:
Updating elements

Convert a list from floats to integers

Spec

```racket
(require rackunit)
; Supplied by the stdlib
(check-equal? 3 (exact-floor 3.14))
(check-equal?
  (list 1 2 3)
  (list-exact-floor (list 1.1 2.6 3.0)))
```

Convert a list from floats to integers

Spec

```racket
(require rackunit)
; Supplied by the stdlib
(check-equal? 3 (exact-floor 3.14))
(check-equal?
  (list 1 2 3)
  (list-exact-floor (list 1.1 2.6 3.0)))
```

Solution

```racket
(define (list-exact-floor l)
  (cond [(empty? l) l]
        [else
          (cons
            (exact-floor (first l))
            (list-exact-floor (rest l)))]))
```

> Can we generalize this for any operation on lists?

```racket
(check-equal?
  (list-exact-floor (list 1.1 2.6 3.0)))
  (list (exact-floor 1.1) (exact-floor 2.6) (exact-floor 3.0)))
```

UMass Boston

Function `map`

Generic solution

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

Using `map`

```
(define (list-exact-floor l)
  (map exact-floor l))
```

- Is `map` function tail-recursive?

Function `map`

Generic solution

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

Using `map`

```
(define (list-exact-floor l)
  (map exact-floor l))
```

> Is `map` function tail-recursive? **No.**

`map` passes the return value of the recursive call to `cons`. The order of applying `cons` is important, so we can't just apply it to an accumulator parameter (as that would reverse the order of application).

> **Idea: *delay adding to the right with a `lambda`*.** First, run all recursive calls at tail-call, while creating a function that processes the result and appends the element to the left (`cons`). Second, run the accumulator function.

The tail-recursive optimization pattern

Tail-recursive `map`, using the generalized tail-recursion optimization pattern

```
(define (map f l)
  (define (map-iter accum l)
    (cond [(empty? l) (accum l)]
          [else (map-iter (lambda (x) (accum (cons (f (first l)) x))) (rest l))]))
  (map-iter (lambda (x) x) l))
```

The accumulator delays the application of `(cons (f (first l)) ?)`.

1. The initial accumulator is `(lambda (x) x)`, which simply returns whatever list is passed to it.
2. The base case triggers the computation of the accumulator, by passing it an empty list.
3. In the inductive case, we just augment the accumulator to take a list `x`, and return `(cons (f (first l)) x)` to the next accumulator.

> The accumulator works like a pipeline: each inductive step adds a new stage to the pipeline, and the base case runs the pipeline: `(stage3 (stage2 (stage1 ((lambda (x) x) nil))))`

Tail-recursive map run

```
(map f (list 1 2 3)) =
; First, build the pipeline accumulator
(define (accum0 x) x) (map-iter accum0 (list 1 2 3)) =
(define (accum1 x) (accum0 (cons (f 1) x))) (map-iter accum1 (list 2 3)) =
(define (accum2 x) (accum1 (cons (f 2) x))) (map-iter accum2 (list 3)) =
(define (accum3 x) (accum2 (cons (f 3) x))) (map-iter accum3 (list)) =
; Second, run the pipeline accumulator
(accum3 (list)) =
(accum2 (list (f 3))) =
(accum1 (list (f 2) (f 3))) =
(accum0 (list (f 1) (f 2) (f 3))) =
(list (f 1) (f 2) (f 3)))
```

Tail-recursive optimization pattern

> To summarize, when a value has base case and an inductive case, we identified the following pattern for a tail-recursive optimization:

Unoptimized

```
(define (rec v)
  (cond
    [(base-case? v) (base v)]
    [else (step v (rec (dec v)))]))
```

Optimized

```
(define (rec v)
  (define (rec-aux accum v)
    (cond
      [(base-case? v) (accum (base v))]
      [else
        (rec-aux
          (lambda (x) (accum (step v x)))
          (dec v))]))
  (rec-aux (lambda (x) x) v))
```

Making map tail-recursive

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

Tail-recursive map run

```
(map f (list 1 2 3)) =
; First, build the pipeline accumulator
(define (accum0 x) x) (map-iter accum0 (list 1 2 3)) =
(define (accum1 x) (accum0 (cons (f 1) x))) (map-iter accum1 (list 2 3)) =
(define (accum2 x) (accum1 (cons (f 2) x))) (map-iter accum2 (list 3)) =
(define (accum3 x) (accum2 (cons (f 3) x))) (map-iter accum3 (list)) =
; Second, run the pipeline accumulator
(accum3 (list)) =
(accum2 (list (f 3))) =
(accum1 (list (f 2) (f 3))) =
(accum0 (list (f 1) (f 2) (f 3))) =
(list (f 1) (f 2) (f 3)))
```

Tail-recursive optimization pattern

> To summarize, when a value has base case and an inductive case, we identified the following pattern for a tail-recursive optimization:

Unoptimized

```
(define (rec v)
  (cond
    [(base-case? v) (base v)]
    [else (step v (rec (dec v)))]))
```

Optimized

```
(define (rec v)
  (define (rec-aux accum v)
    (cond
      [(base-case? v) (accum (base v))]
      [else
        (rec-aux
          (lambda (x) (accum (step v x)))
          (dec v))]))
  (rec-aux (lambda (x) x) v))
```

Tail-recursive `map`, using the generalized tail-recursion optimization pattern

```
(define (map f l)
  (define (map-iter accum l)
    (cond [(empty? l) (accum l)]
          [else (map-iter (lambda (x) (accum (cons (f (first l)) x))) (rest l))]))
  (map-iter (lambda (x) x) l))
```

Scanning

Remove zeros from a list

Spec

```
(require rackunit)
(check-equal? (list 1 3 4) (remove-0 (list 0 1 3 0 4)))
(check-equal? (list 1 2 3) (remove-0 (list 1 2 3)))
```

Remove zeros from a list

Spec

```
(require rackunit)
(check-equal? (list 1 3 4) (remove-0 (list 0 1 3 0 4)))
(check-equal? (list 1 2 3) (remove-0 (list 1 2 3)))
```

Solution

```
(define (remove-0 l)
  (cond
    [(empty? l) l]
    [(not (equal? (first l) 0)) (cons (first l) (remove-0 (rest l)))]
    [else (remove-0 (rest l))]))
```

Can we generalize this functional pattern?

Original

```racket
(define (remove-0 l)
  (cond
    [(empty? l) l]
    [(not (equal? (first l) 0))
     (cons (first l) (remove-0 (rest l)))]
    [else (remove-0 (rest l))]))
```

Generalized

```racket
(define (filter to-keep? l)
  (cond
    [(empty? l) l]
    [(to-keep? (first l))
     (cons (first l)
           (filter1 to-keep? (rest l)))]
    [else (filter to-keep? (rest l))]))

;; Usage example
(define (remove-0 l)
  (filter
    (lambda (x) (not (equal? x 0))) l))
```

▌ Is this function tail-recursive?

Can we generalize this functional pattern?

Original

```
(define (remove-0 l)
  (cond
    [(empty? l) l]
    [(not (equal? (first l) 0))
     (cons (first l) (remove-0 (rest l)))]
    [else (remove-0 (rest l))]))
```

Generalized

```
(define (filter to-keep? l)
  (cond
    [(empty? l) l]
    [(to-keep? (first l))
     (cons (first l)
           (filter1 to-keep? (rest l)))]
    [else (filter to-keep? (rest l))]))

;; Usage example
(define (remove-0 l)
  (filter
    (lambda (x) (not (equal? x 0))) l))
```

Is this function tail-recursive? **No.** Function `cons` is a tail-call; `filter` is not.

Tail-recursive filter

Revisiting the tail call optimization

> Function `filter` has very similar shape than function `map`, so we can apply the same optimization pattern.

```
(define (filter to-keep? l)
  (define (filter-aux accum l)
    (cond
      [(empty? l) (accum l)] ; same as before
      [else
        (define hd (first l)) ; cache the head of the list
        (define tl (rest l))  ; cache the tail of the list
        (cond
          [(to-keep? hd) (filter-aux (lambda (x) (accum (cons hd x))) tl)]
          [else (filter-aux accum tl)])]))
  (filter-aux (lambda (x) x) l))
```

Functional patterns:

Reduction

Concatenate two lists

Concatenate two lists

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))
```

Concatenate two lists

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?

Concatenate two lists

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? **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:
(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

```
(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

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

Implementing map with foldr

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

Solution

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

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

```racket
; 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))]))
```

```
; Example 1:
(define (map f l)
  (define (on-elem elem new-list)
    (cons (f elem) new-list))
  (foldr on-elem empty l))
; Example 2:
(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))
; Example 3:
(define (append l r)
  (foldr cons r l))
```