CS450

Structure of Higher Level Languages

Lecture 6: Functional patterns and tail call optimization

Tiago Cogumbreiro

The final grade is given by the instructor

(not by the autograder)

The autograder just approximates your grade. Why?

- **We are grading the correctness of each exercise; autograder is incomplete**
- Cheating
- Using disallowed functions
- Submissions tricking the autograder system

Grades for HW1 published Friday

Today we will...

- learn important functional patterns
- make functions more general by using functions as parameters
- transform functions into tail recursive functions (exercises 2 and 3 of HW2)

> Section 2.2.1 in SICP. Try out the interactive version of section 2.2 of the SICP book.

Functional pattern: Finding elements

Find a value in a list

Let us implement a function `member` that tests whether or not a list contains a value.

Specification

```
; Unit test that tests
(require rackunit)
(check-true (member 1 (list 3 6 1)))
(check-true (member #t (list 3 #t (list))))
(check-false (member 1 (list 3 #t (list 1))))
(check-false (member #f (list)))
```

Find a value in a list

- Let us implement a function `member` that tests whether or not a list contains a value.

Specification

```
; Unit test that tests
(require rackunit)
(check-true (member 1 (list 3 6 1)))
(check-true (member #t (list 3 #t (list))))
(check-false (member 1 (list 3 #t (list 1))))
(check-false (member #f (list)))
```

Solution

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

- Is the solution tail-recursive?

Find a value in a list

Let us implement a function `member` that tests whether or not a list contains a value.

Specification

```
; Unit test that tests
(require rackunit)
(check-true (member 1 (list 3 6 1)))
(check-true (member #t (list 3 #t (list))))
(check-false (member 1 (list 3 #t (list 1))))
(check-false (member #f (list)))
```

Solution

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

Is the solution tail-recursive? **Yes!**

Common mistakes

Example

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

- Forgetting the base case, results in calling `(first empty)`, which throws an error.
- Forgetting to make the list smaller, results in a non-terminating program.

Base case missing

```
(define (member x l)
  (cond
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

Value doesn't get smaller

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x l)]))
```

Functional pattern: Updating elements

Convert a list from floats to integers

Spec

```
(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)))
```

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.

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))
```

Functional patterns: Queries

Any string in a list matches a prefix

Spec

```racket
(require rackunit)
(check-true (string-prefix? "Racket" "R")) ; available in standard library
(check-true (match-prefix? "R" (list "foo" "Racket")))
(check-false (match-prefix? "R" (list "foo" "bar")))
```

Any string in a list matches a prefix

Spec

```
(require rackunit)
(check-true (string-prefix? "Racket" "R")) ; available in standard library
(check-true (match-prefix? "R" (list "foo" "Racket")))
(check-false (match-prefix? "R" (list "foo" "bar")))
```

Solution

```
(define (match-prefix? prefix l)
  (cond
    [(empty? l) #f]
    [(string-prefix? (first l) prefix) #t]
    [else (match-prefix? prefix (rest l))]))
```

Can we generalize the search algorithm?

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

```
(define (match-prefix? x l)
  (cond
    [(empty? l) #f]
    [(string-prefix? (first l) x) #t]
    [else (match-prefix? x (rest l))]))
```

Can we generalize the search algorithm?

```
(define (member x l)
  (cond
    [(empty? l) #f]
    [(equal? (first l) x) #t]
    [else (member x (rest l))]))
```

```
(define (match-prefix? x l)
  (cond
    [(empty? l) #f]
    [(string-prefix? (first l) x) #t]
    [else (match-prefix? x (rest l))]))
```

Solution

```
(define (exists predicate l)
  (cond
    [(empty? l) #f]
    [(predicate (first l)) #t]
    [else (exists predicate (rest l))]))
```

```
(define (member x l)
  (exists
    (lambda (y) (equal? x y)) l))
(define (match-prefix? x l)
  (exists
    (lambda (y) (string-prefix? y x))) l))
```

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

```racket
(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

```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))]))
```

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?

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 a list of lists

Spec

```
(require rackunit)
(check-equal?
  "foo bar"
  (string-append "foo " "bar"))
(check-equal?
  "> 1 2 3"
  (concat-nums (list 1 2 3)))
```

Concatenate a list of lists

Spec

```
(require rackunit)
(check-equal?
  "foo bar"
  (string-append "foo " "bar"))
(check-equal?
  "> 1 2 3"
  (concat-nums (list 1 2 3)))
```

Solution

```
(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))
```

> Is this tail recursive?

Concatenate a list of lists

Spec

```racket
(require rackunit)
(check-equal?
  "foo bar"
  (string-append "foo " "bar"))
(check-equal?
  "> 1 2 3"
  (concat-nums (list 1 2 3)))
```

Solution

```racket
(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))
```

Is this tail recursive? **Yes.**

Function foldl generalizes reduction

Concrete

```
(define (concat-nums l)
  (define (f n a)
    (string-append a " "
      (number->string n)))

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

Function foldl generalizes reduction

Concrete

```
(define (concat-nums l)
  (define (f n a)
    (string-append a " "
      (number->string n)))

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

General

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

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 emtpy))
```

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))
```

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 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!**

Tail recursive append

- Let us use the *tail-recursive optimization pattern*!

```
(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)
; Non-tail recursive version
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else
          (cons (first l1)
                (append (rest l1) l2))]))
```

Tail recursive append

- Let us use the *tail-recursive optimization pattern*!

```
(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)
; Non-tail recursive version
(define (append l1 l2)
  (cond [(empty? l1) l2]
        [else
          (cons (first l1)
                (append (rest l1) l2))]))
```

```
(define (append l1 l2)
  (define (append-aux accum l1)
    (cond [(empty? l1) (accum l2)]
          [else
            (define h (first l1))
            (append-aux
              (lambda (x) (accum (cons h x)))
              (rest l1))]))
  (append-aux (lambda (x) x) l1))
```