Programming Paradigms

Lecture 3. Higher-order functions and lists

Test N°2

See Moodle

Clarification: cons vs append

```
(cons 1 (list 2 3))                ; (list 1 2 3)
```

Clarification: cons vs append

```
(cons 1 (list 2 3))              ; (list 1 2 3)
(append (list 1) (list 2 3))     ; (list 1 2 3)
```

Clarification: cons vs append

```
(cons 1 (list 2 3))              ; (list 1 2 3)
(append (list 1) (list 2 3))     ; (list 1 2 3)

(cons (list 1) (list 2 3))
```

Clarification: cons vs append

```
(cons 1 (list 2 3))              ; (list 1 2 3)
(append (list 1) (list 2 3))     ; (list 1 2 3)

(cons (list 1) (list 2 3))       ; (list (list 1) 2 3)
```

Clarification: cons vs append

```
(cons 1 (list 2 3))              ; (list 1 2 3)
(append (list 1) (list 2 3))     ; (list 1 2 3)

(cons (list 1) (list 2 3))       ; (list (list 1) 2 3)

(append 1 (list 2 3))
```

Clarification: cons vs append

```
(cons 1 (list 2 3))              ; (list 1 2 3)
(append (list 1) (list 2 3))     ; (list 1 2 3)

(cons (list 1) (list 2 3))       ; (list (list 1) 2 3)

(append 1 (list 2 3))

append: contract violation
  expected: list?
  given: 1
```

Clarification: tail recursion

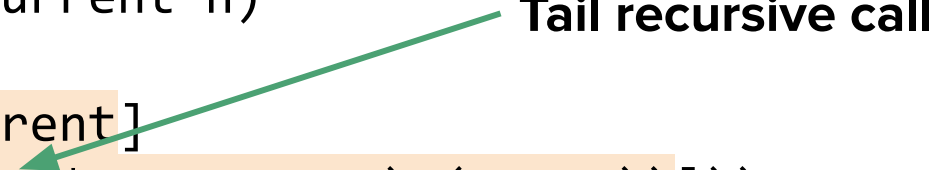
```
(define (factorial n)
  (cond
    [(<= n 1) 1]
    [else (* n (factorial (- n 1)))]))
```

```
(define (factorial n)
  (define (helper current n)
    (cond
      [(<= n 1) current]
      [else (helper (* n current) (- n 1))]))
  (helper 1 n))
```

Clarification: tail recursion

```
(define (factorial n)
  (cond
    [(<= n 1) 1]
    [else (* n (factorial (- n 1)))]))
```

```
(define (factorial n)
  (define (helper current n)
    (cond
      [(<= n 1) current]
      [else (helper (* n current) (- n 1))]))
  (helper 1 n))
```



Outline

- Pairs and Lists in Racket
- Quotation
- Higher-order functions
- Generators in Racket

Pairs in Racket

```
(cons 1 2)           ; '(1 . 2)
(cons "hi" "world") ; '("hi" . "world")
```

Pairs in Racket

```
(cons 1 2)           ; '(1 . 2)
(cons "hi" "world") ; '("hi" . "world")

; cons? = pair?
(pair? (cons 1 2))
```

Pairs in Racket

```
(cons 1 2)           ; '(1 . 2)
(cons "hi" "world") ; '("hi" . "world")

; cons? = pair?
(pair? (cons 1 2))

(cons 1 empty)       ; '(1)
(cons 1 (list 2 3)) ; '(1 2 3)
```

Pairs in Racket

```
(cons 1 2)           ; '(1 . 2)
(cons "hi" "world") ; '("hi" . "world")

; cons? = pair?
(pair? (cons 1 2))

(cons 1 empty)       ; '(1)
(cons 1 (list 2 3)) ; '(1 2 3)
(cons 1 (cons 2 3)) ; '(1 2 . 3)
```

Pairs in Racket: accessors

```
(first  (list 1 2 3)) ; 1
(second (list 1 2 3)) ; 2
(third  (list 1 2 3)) ; 3

(rest (list 1 2 3)) ; '(2 3)
```

Pairs in Racket: accessors

```
(first  (list 1 2 3)) ; 1
(second (list 1 2 3)) ; 2
(third  (list 1 2 3)) ; 3

(rest (list 1 2 3)) ; '(2 3)

(car (cons 3 4)) ; 3
(cdr (cons 3 4)) ; 4
```

Pairs in Racket: accessors

```
(first  (list 1 2 3)) ; 1
(second (list 1 2 3)) ; 2
(third  (list 1 2 3)) ; 3

(rest (list 1 2 3)) ; '(2 3)

(car (cons 3 4)) ; 3
(cdr (cons 3 4)) ; 4
```

```
(car   (list 1 2 3))

(cdr (list 1 2 3))
```

Pairs in Racket: accessors

```
(first  (list 1 2 3)) ; 1
(second (list 1 2 3)) ; 2
(third  (list 1 2 3)) ; 3

(rest (list 1 2 3)) ; '(2 3)

(car (cons 3 4)) ; 3
(cdr (cons 3 4)) ; 4
```

```
(car   (list 1 2 3))
(cadr  (list 1 2 3))
(caddr (list 1 2 3))

(cdr (list 1 2 3))
```

Quotation

```
(list (list 1) (list 2 3) (list 4)) ; '((1) (2 3) (4))
```

Quotation

```
(list (list 1) (list 2 3) (list 4)) ; '((1) (2 3) (4))

(quote ((1) (2 3) (4)))             ; '((1) (2 3) (4))
```

Quotation

```
(list (list 1) (list 2 3) (list 4)) ; '((1) (2 3) (4))

(quote ((1) (2 3) (4)))             ; '((1) (2 3) (4))

'((1) (2 3) (4))
```

Quotation

```
(list (list 1) (list 2 3) (list 4)) ; '((1) (2 3) (4))

(quote ((1) (2 3) (4)))             ; '((1) (2 3) (4))

'((1) (2 3) (4))

(first '((1) (2 3) (4)))  ; '(1)
(rest  '((1) (2 3) (4)))  ; '((2 3) (4))
```

Conditionals with lists

```
(define (fib n)
  (cond
    [(<= n 2) 1]
    [else (+ (fib (- n 1))
             (fib (- n 2)))]))
```

```
(or (> 3 2) (> (fib 100) 1000))    ; #t
```

```
(and (> 2 3) (> (fib 100) 1000))    ; #f
```

Higher-order functions: `twice`

```
(define (twice f x)
  (f (f x)))

(twice sqrt 16)  ; 2
```

Higher-order functions: map

```
(map (lambda (x) (* x x))
     '(1 2 3 4 5))
; '(1 4 9 16 25)
```

Higher-order functions: map

```
(map (lambda (x) (* x x))
     '(1 2 3 4 5))
; '(1 4 9 16 25)

(ormap odd? '(2 4 6 8 10))
; #f
```

Higher-order functions: map

```
(map (lambda (x) (* x x))
     '(1 2 3 4 5))
; '(1 4 9 16 25)

(ormap odd? '(2 4 6 8 10))
; #f

(andmap even? '(2 4 6 8 10))
; #t
```

Higher-order functions: ormap

```
(define (divides? n m)
  (= 0 (remainder m n)))

(define (prime? n)
  (not (ormap
        (lambda (k) (divides? k n))
        (range 2 n))))
```

Higher-order functions: ormap vs or+map

```
(define (my-ormap f lst)
  (apply or (map f lst)))
```

Higher-order functions: ormap vs or+map

```
(define (my-ormap f lst)
  (apply or (map f lst)))
```

```
or: bad syntax in: or
```

Higher-order functions: `filter`

```
(filter even? '(1 2 3 4 5))
; '(2 4)

(filter prime? (range 2 50))
; '(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47)
```

Higher-order functions: `filter`

```
(define students
  (list
   (cons "Anna"    4.3)
   (cons "Boris"   3.6)
   (cons "Charles" 4.1)
   (cons "Daria"   2.9)))
```

Higher-order functions: `filter`

```
(define students
  (list
   (cons "Anna"    4.3)
   (cons "Boris"   3.6)
   (cons "Charles" 4.1)
   (cons "Daria"   2.9)))

(map car
     (filter (lambda (student)
               (> (cdr student) 4.0))
             students))
; '("Anna" "Charles")
```

Higher-order functions: `filter`

```
(define students
  (list
   (cons "Anna"    4.3)
   (cons "Boris"   3.6)
   (cons "Charles" 4.1)
   (cons "Daria"   2.9)))

(map car
     (filter (lambda (student)
               (> (cdr student) 4.0))
             students))
; '("Anna" "Charles")
```

```
'("Anna"    . 4.3)
'("Boris"   . 3.6)
'("Charles" . 4.1)
'("Daria"   . 2.9)
```

Higher-order functions: `filter`

```
(define students
  (list
   (cons "Anna"    4.3)
   (cons "Boris"   3.6)
   (cons "Charles" 4.1)
   (cons "Daria"   2.9)))

(map car
     (filter (lambda (student)
               (> (cdr student) 4.0))
             students))
; '("Anna" "Charles")
```

```
'("Anna"    . 4.3)

'("Charles" . 4.1)
```

Higher-order functions: `filter`

```
(define students
  (list
   (cons "Anna"    4.3)
   (cons "Boris"   3.6)
   (cons "Charles" 4.1)
   (cons "Daria"   2.9)))
```

```
(map car
     (filter (lambda (student)
               (> (cdr student) 4.0))
             students))
; '("Anna" "Charles")
```

```
  "Anna"

  "Charles"
```

Higher-order functions: `foldl`

```
(define (f x current) ...)

(foldl f z '(a b c d))
; f (f (f (f z a) b) c) d

(foldl + 0 '(1 2 3 4))   ; 10
(foldl * 1 '(1 2 3 4))   ; 24
```

Higher-order functions: `foldl`

```
(foldl
 (lambda (student current)
   (cond
     [(> (cdr student) 4.0)
      (cons (car student) current)]
     [else current]))
 empty
 students)
```

students

```
'("Anna"    . 4.3)
'("Boris"   . 3.6)
'("Charles" . 4.1)
'("Daria"   . 2.9)
```

current

```
empty
```

Higher-order functions: `foldl`

```
(foldl
 (lambda (student current)
   (cond
     [(> (cdr student) 4.0)
      (cons (car student) current)]
     [else current]))
 empty
 students)
```

students

```
student = '("Anna"    . 4.3)
          '("Boris"   . 3.6)
          '("Charles" . 4.1)
          '("Daria"   . 2.9)
```

current

```
empty
```

Higher-order functions: `foldl`

```
(foldl
 (lambda (student current)
   (cond
     [(> (cdr student) 4.0)
      (cons (car student) current)]
     [else current]))
 empty
 students)
```

students

```
'("Anna"    . 4.3)
'("Boris"   . 3.6)
'("Charles" . 4.1)
'("Daria"   . 2.9)
```

student =

current

```
'("Anna")
```

Higher-order functions: `foldl`

```
(foldl
 (lambda (student current)
   (cond
     [(> (cdr student) 4.0)
      (cons (car student) current)]
     [else current]))
 empty
 students)
```

students

```
'("Anna"    . 4.3)
'("Boris"   . 3.6)
'("Charles" . 4.1)
'("Daria"   . 2.9)
```

student =

current

```
'("Anna")
```

Higher-order functions: `foldl`

```
(foldl
 (lambda (student current)
   (cond
     [(> (cdr student) 4.0)
      (cons (car student) current)]
     [else current]))
 empty
 students)
```

students

```
'("Anna"    . 4.3)
'("Boris"   . 3.6)
'("Charles" . 4.1)
'("Daria"   . 2.9)
```

student =

current

```
'("Charles" "Anna")
```

Implementing insertion sort with higher-order functions

Generators: `generator` and `yield`

```
(define (loop students)
  (cond
    [(empty? students)
     "no more 4.0+ students"]
    [(> (cdar students) 4.0)
     (begin
       (yield (car students))
       (loop (cdr students)))]
    [else (loop (cdr students))]))

(define students-with-4+
  (generator () (loop students)))
```

Generators: `generator` and `yield`

```
(define (loop students)
  (cond
    [(empty? students)
     "no more 4.0+ students"]
    [(> (cdr (first students)) 4.0)
     (begin
       (yield (car (first
students)))
       (loop (rest students)))]
    [else (loop (rest students))]))

(define students-with-4+
  (generator () (loop students)))
```

Generators: `generator` and `yield`

```
                                    (students-with-4+) ; "Anna"

(define (loop students)
  (cond
    [(empty? students)
     "no more 4.0+ students"]
    [(> (cdr (first students)) 4.0)
     (begin
       (yield (car (first
students)))
       (loop (rest students)))]
    [else (loop (rest students))]))

(define students-with-4+
  (generator () (loop students)))
```

Generators: `generator` and `yield`

```
(students-with-4+) ; "Anna"
(students-with-4+) ; "Charles"
```

```
(define (loop students)
  (cond
    [(empty? students)
     "no more 4.0+ students"]
    [(> (cdr (first students)) 4.0)
     (begin
       (yield (car (first
students)))
       (loop (rest students)))]
    [else (loop (rest students))]))

(define students-with-4+
  (generator () (loop students)))
```

Generators: `generator` and `yield`

```
(define (loop students)
  (cond
    [(empty? students)
     "no more 4.0+ students"]
    [(> (cdr (first students)) 4.0)
     (begin
       (yield (car (first
students)))
       (loop (rest students)))]
    [else (loop (rest students))]))

(define students-with-4+
  (generator () (loop students)))
```

```
(students-with-4+) ; "Anna"
(students-with-4+) ; "Charles"
(students-with-4+) ; "no more 4.0+ students"
```

What what the most unclear part of the lecture for you?

See Moodle

References

1. Racket Essentials 2.4 — Pairs, Lists, and Racket Syntax
2. Racket Essentials 2.2.5 — Conditionals with `if`, `and`, `or`, and `cond`
3. Racket Essentials 2.3.1 — Predefined List Loops