Scheme Notes 03

Geoffrey Matthews

Department of Computer Science
Western Washington University

January 23, 2018

Recursion *vs.* Tail-recursion

$$a^b = \begin{cases} 1 & \text{if } b = 0 \\ a(a^{b-1}) & \text{otherwise} \end{cases}$$

```
(define pow-rec
  (lambda (a b)
    (if (zero? b)
        1
        (* a (pow-rec a (- b 1))))))
```

Recursion *vs.* Tail-recursion

$$a^b = \begin{cases} 1 & \text{if } b = 0 \\ a(a^{b-1}) & \text{otherwise} \end{cases}$$

```
(define pow-rec
  (lambda (a b)
    (if (zero? b)
        1
        (* a (pow-rec a (- b 1))))))

(define pow-iter
  (lambda (a b)
    (define pow-iter-loop
      (lambda (b product)
        (if (zero? b)
            product
            (pow-iter-loop (- b 1) (* a product)))))
    (pow-iter-loop b 1)))
```

Named let

```
(define pow-iter
  (lambda (a b)
    (define pow-iter-loop
      (lambda (b product)
        (if (zero? b)
            product
            (pow-iter-loop (- b 1) (* a product)))))
    (pow-iter-loop b 1)))


(define pow-iter-2
  (lambda (a b)
    (let pow-iter-loop ((b b) (product 1))
      (if (zero? b)
          product
          (pow-iter-loop (- b 1) (* a product))))))
```

Fast recursion

```
(define pow-fast
  (lambda (a b)
    (cond ((zero? b) 1)
          ((even? b) (sqr (pow-fast a (/ b 2))))
          (else (* a (pow-fast a (- b 1)))))))
```

Lists

```
(define a (list 1 2 3 4 5))
(define b (list 6 7 8))
(define c '(1 2 3 4 5))
(define d (cons 6 (cons 7 (cons 8 '()))))
```

- Run `boxarrow.rkt` for pictures.

length

length

```
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))
```

nth

nth

```
(define (nth lst n)
  (cond ((null? lst) nil)
        ((= n 0) "Not defined")
        ((= n 1) (car lst))
        (else (nth (cdr lst) (- n 1)))))
```

last

last

```
(define (last lst)
  (cond ((null? lst) nil)
        ((null? (cdr lst)) (car lst))
        (else (last (cdr lst)))))
```

scale-list

scale-list

```
(define (scale-list lst n)
  (if (null? lst)
      nil
      (cons (* n (car lst))
            (scale-list (cdr lst) n))))
```

increment-list

increment-list

```
(define (increment-list lst)
  (if (null? lst)
      nil
      (cons (+ 1 (car lst))
            (increment-list (cdr lst)))))
```

map

map

```
(define (map lst op)
  (if (null? lst)
      nil
      (cons (op (car lst))
            (map (cdr lst) op))))
```

scale-list using map

scale-list using map

```
(define (scale-list lst n)
  (map lst (lambda (x) (* n x))))
```

increment-list using map

increment-list using map

```
(define (increment-list lst)
  (map lst (lambda (x) (+ x 1))))
```

append

append

```
(define (append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1)
            (append (cdr lst1) lst2))))
```

remove

remove

```
(define (remove n lst)
  (cond ((null? lst) nil)
        ((= n (car lst)) (remove n (cdr lst)))
        (else (cons (car lst)
                    (remove n (cdr lst))))))
```

Trees

Trees

```
(define tree1 (cons (list 1 2) (list 3 4)))

(define tree2 (list (list 1 2)
                    (list 3
                          (list 4 5 6))
                    (list 7 8)))
```

- Run `boxarrow.rkt` for pictures.

count-leaves

count-leaves

```
(define (count-leaves tree)
  (cond ((null? tree) 0)
        ((not (pair? tree)) 1)
        (else (+ (count-leaves (car tree))
                 (count-leaves (cdr tree))))))
```

fringe

fringe

```
(define (fringe tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (fringe (car tree))
                      (fringe (cdr tree))))))
```

sum-fringe

sum-fringe

```
(define (sum-fringe tree)
  (cond ((null? tree) 0)
        ((number? tree) tree)
        (else (+ (sum-fringe (car tree))
                 (sum-fringe (cdr tree))))))
```

map-tree

map-tree

```
(define (map-tree tree op)
  (cond ((null? tree) nil)
        ((number? tree) (op tree))
        (else (cons (map-tree (car tree) op)
                    (map-tree (cdr tree) op)))))
```

scale-tree using map-tree

scale-tree using map-tree

```
(define (scale-tree tree factor)
  (map-tree tree (lambda (x) (* x factor))))
```

increment-tree using map-tree

increment-tree using map-tree

```
(define (increment-tree tree)
  (map-tree tree inc))
```