Scheme Notes 03

Geoffrey Matthews

Department of Computer Science
Western Washington University

January 29, 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 loop
      (lambda (b product)
        (if (zero? b)
            product
            (loop (- b 1) (* a product)) )))
    (loop b 1)))
```

Named let

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


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

Fast recursion

$$a^b = \begin{cases} 1 & \text{if } b = 0 \\ (a^{b/2})^2 & \text{if } b \text{ is even} \\ a(a^{b-1}) & \text{otherwise} \end{cases}$$

```
(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

A **list** is either:

1. the **empty list**, or
2. **an item** and a **list**

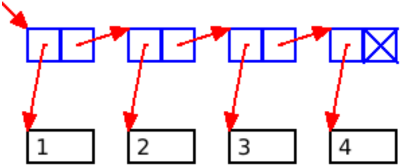
Lists

A **list** is either:

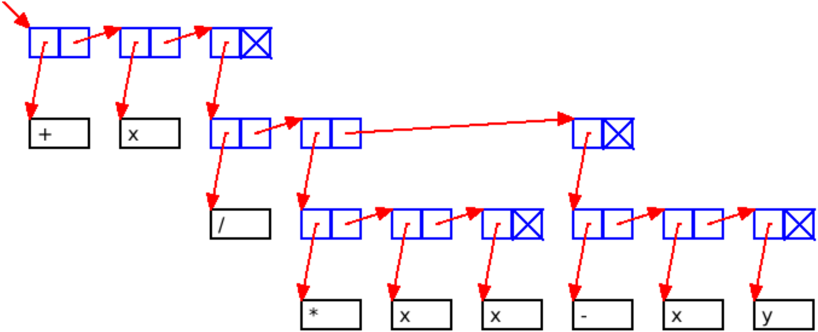
1. the **empty list**, or
2. **an item** and a **list**

Scheme uses:

1. the **null pointer** for the empty list, and
2. a **cons cell** of two pointers for a non-empty list.
3. The first pointer in a cons cell is called **car**.
4. The second pointer in a cons cell is called **cdr**.
5. The empty list has predicate **empty?**.



Scheme Programs are Lists




```
(+ x (/ (* x x) (- x y)))
```

Building Lists in Scheme:

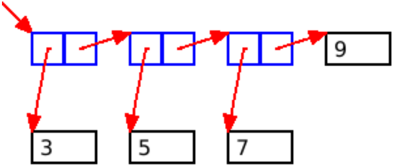
1. The empty list in Scheme: $\boxed{\texttt{'()}}$
2. Create a list from 3 and the empty list:
   $\boxed{\texttt{(cons 3 '())} \Rightarrow \texttt{(3)}}$
3. Create the list $\boxed{\texttt{(4 7 2)}}$:
   $\boxed{\texttt{(cons 4 (cons 7 (cons 2 '())))} \Rightarrow \texttt{(4 7 2)}}$
4. Shorthand for long lists: $\boxed{\texttt{(list 4 7 2)} \Rightarrow \texttt{(4 7 2)}}$

Building Lists in Scheme:

1. The empty list in Scheme: `'()`
2. Create a list from 3 and the empty list:
   `(cons 3 '()) ⇒ (3)`
3. Create the list `(4 7 2)`:
   `(cons 4 (cons 7 (cons 2 '())))⇒ (4 7 2)`
4. Shorthand for long lists: `(list 4 7 2) ⇒ (4 7 2)`
5. Using quote: `'(4 7 2) ⇒ (4 7 2)`
   `'(+ 4 7 2) ⇒ (+ 4 7 2)`
   `'(a b c) ⇒ (a b c)`
   `(a b c) ⇒` *error*
   `(+ 4 7 2) ⇒ 13`
   `'(list (+ 2 2) 7 2) ⇒ (list (+ 2 2) 7 2)`
   `(list (+ 2 2) 7 2) ⇒ (4 7 2)`

An improper list results in a dot:

- $\boxed{\texttt{(cons 4 8)} \Rightarrow \texttt{(4 . 8)}}$
- $\boxed{\texttt{(cons 3 (cons 5 (cons 7 9)))} \Rightarrow \texttt{(3 5 7 . 9)}}$
- Run `boxarrow.rkt` for pictures.



length

length

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

nth

nth

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

last

last

```
(define (last lst)
  (cond ((empty? lst) '())
        ((empty? (cdr lst)) (car lst))
        (else (last (cdr lst)))))
```

scale-list

scale-list

```
(define (scale-list lst n)
  (if (empty? lst)
      '()
      (cons (* n (car lst))
            (scale-list (cdr lst) n))))
```

increment-list

increment-list

```
(define (increment-list lst)
  (if (empty? lst)
      '()
      (cons (+ 1 (car lst))
            (increment-list (cdr lst)))))
```

map

map

```
(define (map lst op)
  (if (empty? lst)
      '()
      (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 (empty? lst1)
      lst2
      (cons (car lst1)
            (append (cdr lst1) lst2))))
```

remove

remove

```
(define (remove n lst)
  (cond ((empty? lst) '())
        ((= n (car lst)) (remove n (cdr lst)))
        (else (cons (car lst)
                    (remove n (cdr lst))))))
```