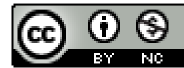
プログラミング入門

第7回「リストと高階関数」

二宮 崇（ ninomiya@cs.ehime-u.ac.jp ）

教科書

Structure and Interpretation of Computer Programs, 2nd Edition: Harold Abelson, Gerald Jay Sussman, Julie Sussman, The MIT Press, 1996



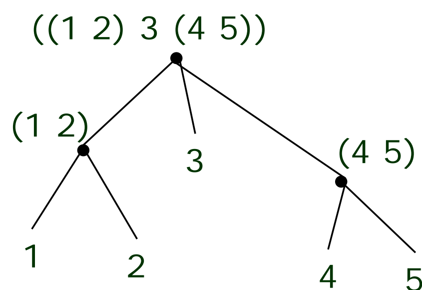
データによる抽象の構築（第2章の続き）

2.2.2 階層構造

リストの要素がリストであるようなデータ構造を作ることもできます。例えば、

```
(list (list 1 2) 3 (list 4 5))
```

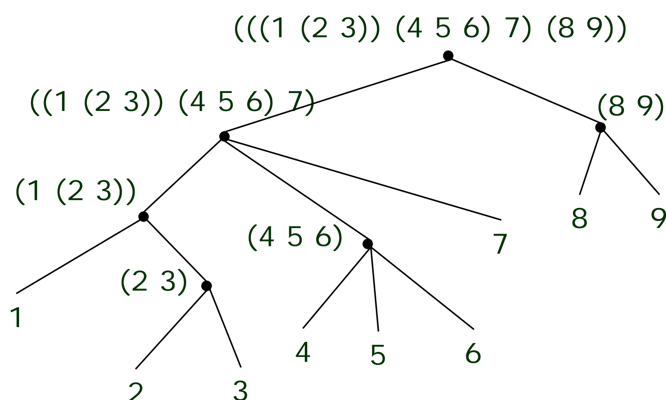
という風にリストの中にリストを埋め込むことができます。一つ目の要素が(list 1 2)で次の要素が 3 でそのさらに次の要素が(list 4 5)となっています。このようにリストの中にリストを埋め込めると、木構造を表現することができるようになります。この例を木構造で表現すると次のようになります。



一般にいくつかの頂点を辺で結んだ構造はグラフと呼ばれますが、辺に向きがない場合は無向グラフと呼ばれ、辺に向きがある場合（矢印で表現される）は有向グラフと呼ばれ

ます。無向グラフのうち、閉路(ある頂点から出発してまた元の頂点にもどってくるような辺の辿り道のこと)を含まないグラフを無向木または木と呼びます。無向木は上の図のようによく描かれますが、一番上の頂点のことを根(root)と呼びます。一番下に並ぶ末端は葉(leaf)と呼ばれます。この場合は、全体を表す((1 2) 3 (4 5))に対応する頂点が根であり、数値の 1, 2, 3, 4, 5 が葉となります。リストを使うとこのような根付き無向木を表現することができます。

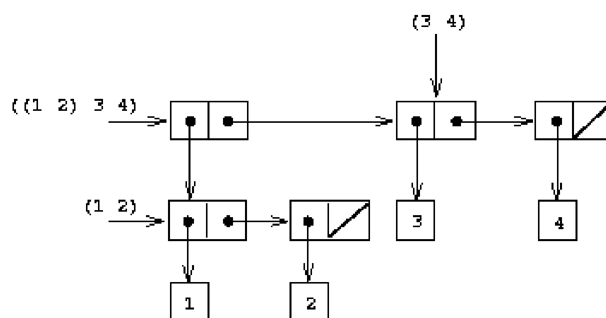
他にも例をだすと、(list (list (list 1 (list 2 3)) (list 4 5 6) 7) (list 8 9)) を評価すると、次のような木が得られます。



ちょっとわかりにくい例もやってみましょう。

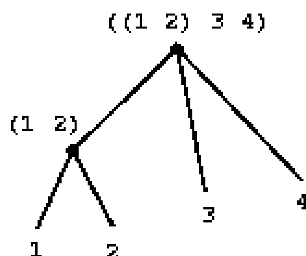
(cons (list 1 2) (list 3 4))

という構造はどのように解釈されるでしょうか。箱とポインタ記法で書くと次のようになります。



リストとして解釈すると、(cons x (list 3 4))という形になっていて x=(list 1 2)という形になっています。リストの先頭に cons でつなげたものもまたリストになる、ということに注意してください。つまり、これは(cons x (cons 3 (cons 4 '())))という

形とおなじなので、(list x 3 4)と同じことになります。つまりこれは、(list (list 1 2) 3 4)と同じことになります。(cons (list 1 2) (list 3 4))を木としてみると次のようになります。



　このような木に対して、リストの長さを測る length を実行すると、次のようになります。

(define x (cons (list 1 2) (list 3 4)))

(length x)　→　3

(length (list x x))　→ 2

木の根の最初の分岐数だけが返ってくることになります。木構造の全ての葉の数を数えたい場合は次のように再帰的に求めれば良いことになります。その関数を count-leaves と名付けましょう。

・空リストに対する count-leaves の値は 0 である。

・木 x に対する count-leaves は、(car x)の count-leaves 足す(cdr x)の count-leaves である。

・葉の count-leaves は 1 である。

葉である、ということは、空リストでも、cons でもないということです。cons であるかどうかは pair?という関数で判定できます。

　この再帰的定義にしたがって Scheme で実装すると次のようになります。

```
(define (count-leaves x)
   (cond ((null? x) 0)
```

```
((not (pair? x)) 1)
(else (+ (count-leaves (car x))
         (count-leaves (cdr x)))))
```

2.2.3 公認インターフェースとしてのリスト

ドット末尾記法

+、-、list といった関数は任意の数の引数をとることができます。そういう関数を定義する一つの方法にドット末尾記法と呼ばれる方法があります。

(define (f x y . z) ⟨*body*⟩)

と定義すると、(f 1 2 3 4 5 6)を評価すると、f の本体では、x は 1、y は 2、z はリスト(3 4 5 6)になります。

(define (g . w) ⟨*body*⟩)なら、関数 g は 0 個かそれ以上の引数で呼び出すことができます。(g 1 2 3 4 5 6)なら、g 本体で w はリスト(1 2 3 4 5 6)になります。

リストの写像

非常によく使われる手法として、リストの各要素になんらかの演算を加えたリストを返すことがあります。例えば、次の関数はリストの各要素を与えられた factor 倍にします。

```
(define (scale-list items factor)
  (if (null? items)
      '()
      (cons (* (car items) factor)
            (scale-list (cdr items) factor))))
```

例: (scale-list (list 1 2 3 4) 10)　→ (10 20 30 40)

この関数を抽象化して、リストの各要素にどのような処理を加えるか指定する関数を渡せるようにします。そのようにした抽象化された高階関数を map といいます。以下では、map や map 以外の便利な抽象化された関数について学びます。これらのリストを扱う抽象化された関数は一般によく使われ、リストは公認インターフェースとして用いられています。

公認インターフェースとしてのリスト

map, filter, reduce, range などのリストを扱う抽象化された関数があります。

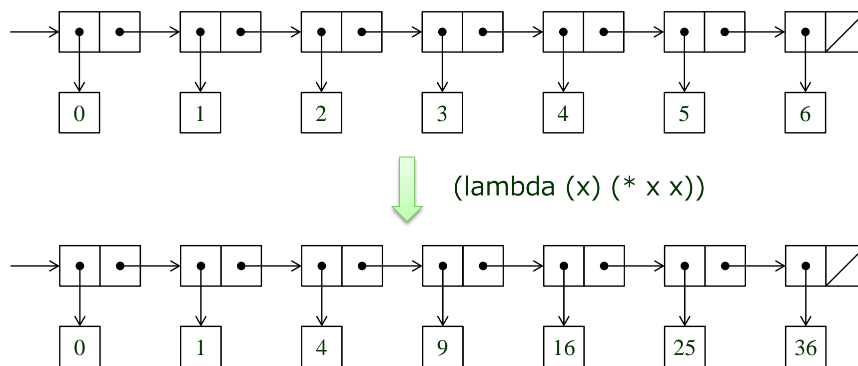
map（racket の R5RS に組み込みで定義）

```
(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items))
            (map proc (cdr items)))))
```

上で定義したように map は受け取ったリスト(items)に対し、受け取った関数(proc)を items のそれぞれの要素に適用した結果をリストとして返します。

例

```
> (map (lambda (x) (* x x)) (list 0 1 2 3 4 5 6))
(0 1 4 9 16 25 36)
```



例：(map abs (list -10 2.5 -11.6 17)) → (10 2.5 11.6 17) ※abs は絶対値を返す組み込み関数

例：(map (lambda (x) (* x x)) (list 1 2 3 4)) → (1 4 9 16)

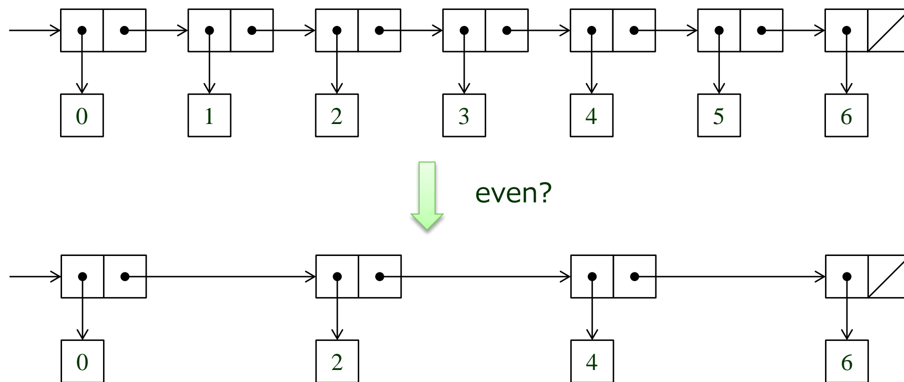
filter

```
(define (filter predicate sequence)
  (cond ((null? sequence) '())
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))
```

filter は sequence の要素のうち predicate の条件を満たす要素だけからなるリストを返します。

例

```
> (filter even? (list 0 1 2 3 4 5 6 7 8 9))
(0 2 4 6 8)
```



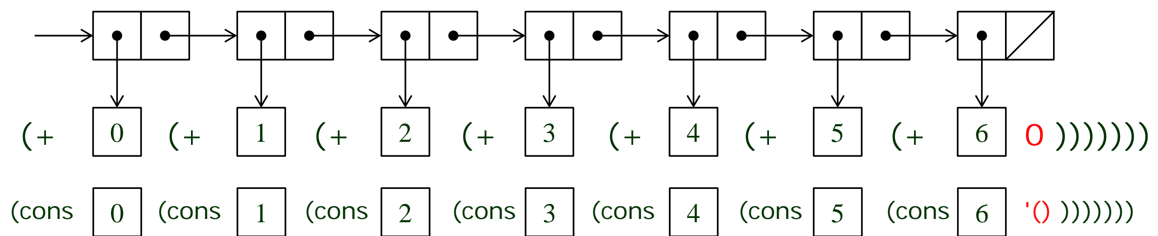
reduce（教科書では、accumulate）

```
(define (reduce op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (reduce op initial (cdr sequence)))))
```

(op ＜リストの第１要素＞ (op ＜リストの第２要素＞ (op … (op ＜リストの最後の要素＞ initial)…))) を実行します。つまり、initial を初期値として、リストの各要素に対し、op を順に適用することになります。ただし、op は二つ引数をとる関数で、リストの要素とリストの残りに対する計算結果を受け取り、新しい計算結果を返します。

例

```
> (reduce + 0 (list 0 1 2 3 4 5 6))
21
> (reduce cons '() (list 0 1 2 3 4 5 6))
(0 1 2 3 4 5 6)
```



range（教科書では enumerate-interval）

```
(define (range low high)
  (if (> low high)
      '()
      (cons low (range (+ low 1) high))))
```

low から high の整数のリストを生成する。

例

```
> (range 2 7)
(2 3 4 5 6 7)
```

これらの抽象化関数を使うと、例えば、フィボナッチ数列のうち偶数の列だけ取り出す、というプログラムをフィボナッチ数列のプログラムを書き換えることなく簡単に実現することができます。

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
(define (even-fibs n)
  (filter even? (map fib (range 0 n))))
```

```
> (even-fibs 20)

(0 2 8 34 144 610 2584)
```

効率はそんなによくないかもしれませんが、簡単に実装することができるため、急いで実装しなくてはいけないときや、簡単なテストを行いたいときなどに大変便利です。