

工学部専門科目

「プログラミング言語」 SICP
第3章 ~ Modularity, Objects, State ~
その3

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学 大学院情報学研究科
通信情報システム専攻

May 13, 2014

今日のメニュー

3.3 変更可能データを使ったモデリング

3.3.1 変更可能なリスト構造

3.3.2 キュー (queue) を表現する

3.3.3 表 (table) を表現する

3.3 変更可能データを使ったモデリング

- 2章のトピック: 複合的データの構成法とデータ抽象
 - ▶ コンストラクタ (cons) とセレクタ (car, cdr)
- 本章のトピック: 状態
 - ⇒ 複合的データの変更: ミューテータ (mutator)

3.3.1 変更可能なリスト構造

原始的なミューテータ `set-car!`, `set-cdr!`

`(set-car! <式1> <式2>)`

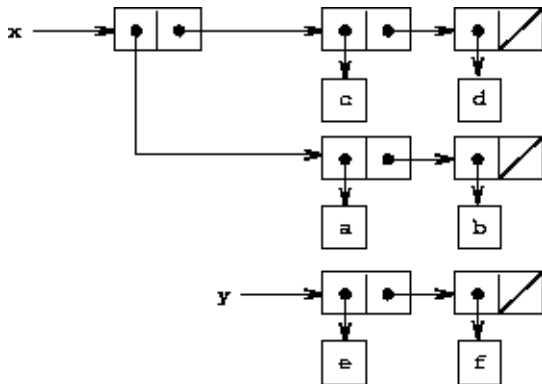
`(set-cdr! <式1> <式2>)`

- `<式1>` の値が指すペアの `car` 部 (`cdr` 部) (が指す先) を `<式2>` の値に変更する。
 - ▶ 第1引数は (`set!` と違い) 変数に限らず (ペアを示す) 式なら何でもよい
- 式の値は (`set!` 同様) 実装依存
 - ▶ なので値は使わないで捨てるのがふつう

例(1/4)

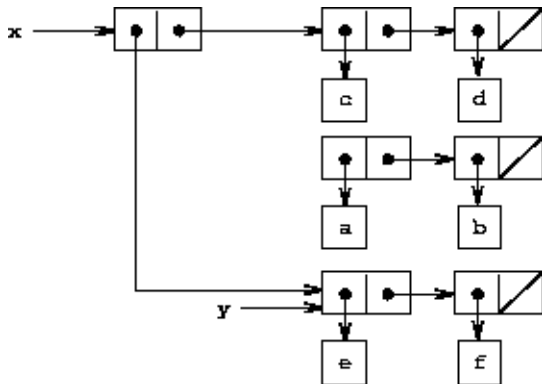
```
(define x '((a b) c d))
```

```
(define y '(e f))
```



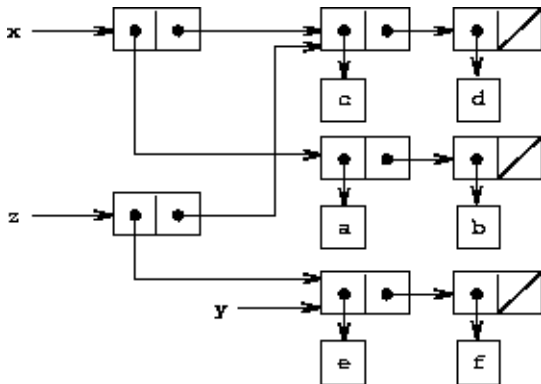
例(2/4)

(set-car! x y) をすると...



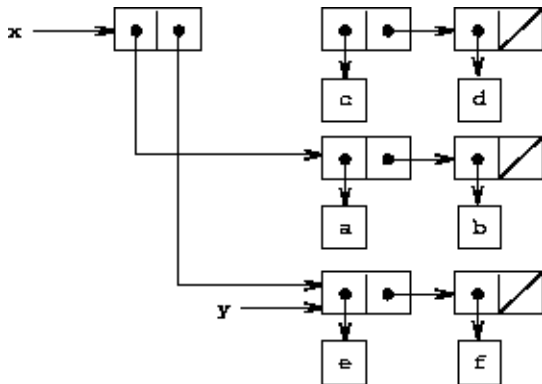
例 (3/4)

代わりに `(define z (cons y (cdr x)))` だと, 新しいペアが作られ...



例 (4/4)

(set-cdr! x y) をすると...



(cons x y) の動作を分解してみると...

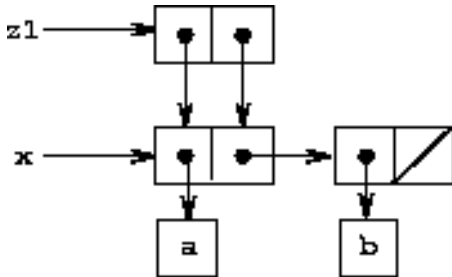
- (何も指していない) 新しいペアを作る
 - set-car! で新しいペアから x を指す
 - set-cdr! で新しいペアから y を指す
- と考えることができる．あえて書くなら

```
(define (cons x y)
  (let ((new (get-new-pair)))
    (set-car! new x)
    (set-cdr! new y)
    new))
```

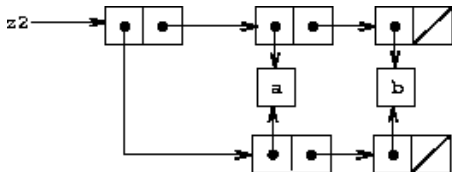
共有と同一性

- 3.1.3 では、値の「同じであること」「変化」について議論した
- データ構造におけるペアの共有でも同じことがいえる
 - ▶ 実際、データ構造の共有もエイリアシングという

```
(define x (list 'a 'b))  
(define z1 (cons x x))
```



```
(define z1 (cons (list 'a 'b) (list 'a 'b)))
```



クイズ

define 一発でひとつめの構造を作ることはできるでしょうか？

```
(define (set-to-wow! x)
  (set-car! (car x) 'wow)
  x)
```

z1

```
(set-to-wow! z1)
```

z2

```
(set-to-wow! z2)
```

eq? と同一性

eq? の機能

- シンボルの等しさ判定
- ポインタとしての等しさ (指す先が等しいかどうかの) 判定

```
(eq? (car z1) (cdr z1))
```

```
(eq? (car z2) (cdr z2))
```

更新 (mutation) と破壊的代入 (assignment)

set-car!, set-cdr! は set! があれば実現できる!

手続きを使った cons, car, cdr (2.1.3 より)

```
(define (cons x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (error ...))))
  dispatch)
(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
```

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ...
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          ...))
  dispatch)
(define (set-car! z new-value)
  ((z 'set-car!) new-value)
  z)
```


3.3.2 キュー (queue) を表現する

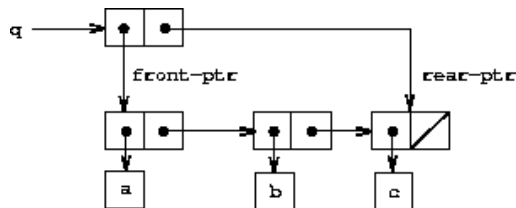


キューの操作

- コンストラクタ
 - ▶ (make-queue): 新しい空のキューを作る
- セレクタ
 - ▶ (empty-queue? q): q が空かどうかの検査
 - ▶ (front-queue q): q の先頭要素を返す (キューの状態はそのまま)
- ミューテータ
 - ▶ (insert-queue! q a): q の末尾に a を追加する
 - ▶ (delete-queue! q): q の先頭要素を削除する

実装のアイデア

- 要素の列はリストで表現する
 - ▶ が、リストは末尾の要素を取り出すのに (要素数に比例する) 時間がかかる...
- リストの先頭と末尾を指すふたつのポインタを保持する



- 要素の追加・削除ごとに先頭・末尾ポインタが指す先を「ずらす」

キュー操作の補助関数

```
(define (front-ptr queue) (car queue))  
(define (rear-ptr queue) (cdr queue))  
(define (set-front-ptr! queue item)  
  (set-car! queue item))  
(define (set-rear-ptr! queue item)  
  (set-cdr! queue item))
```

コンストラクタとセレクタ

```
(define (empty-queue? queue)
  (null? (front-ptr queue)))
(define (make-queue) (cons '() '()))

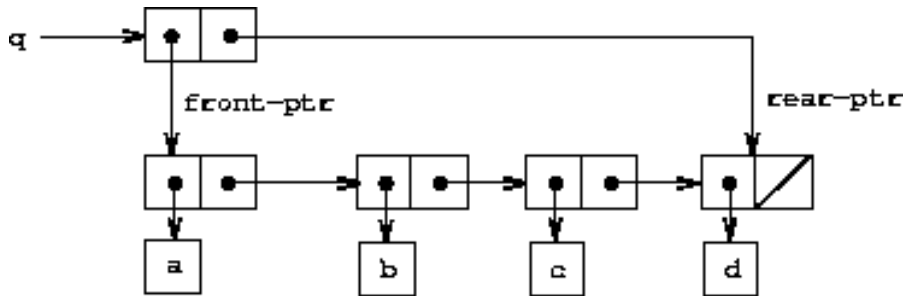
(define (front-queue queue)
  (if (empty-queue? queue)
      (error ...)
      (car (front-ptr queue))))
```

ミューテータ (1/2)

キューが空の場合は挿入処理が特殊

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond
      ((empty-queue? queue) ;; キューが空
       (set-front-ptr! queue new-pair)
       (set-rear-ptr! queue new-pair)
       queue)
      (else
       (set-cdr! (rear-ptr queue) new-pair)
       (set-rear-ptr! queue new-pair)
       queue))))
```

(insert-queue! q 'd) の結果:



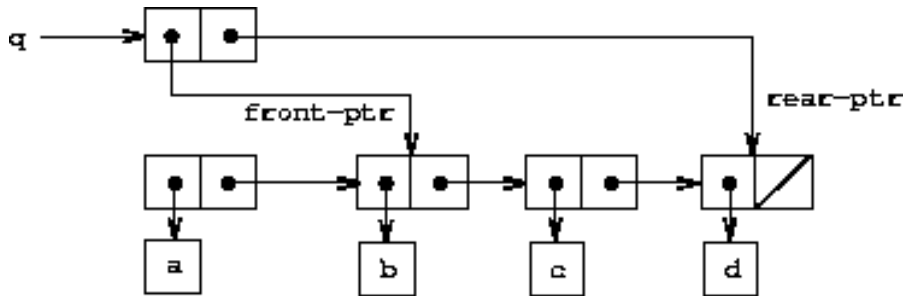
ミューテータ (2/2)

```
(define (delete-queue! queue)
  (cond
    ((empty-queue? queue)
     (error ...))
    (else
     (set-front-ptr! queue
                      (cdr (front-ptr queue))
                      queue))))
```

クイズ

なぜ `insert-queue!` や `delete-queue!` の定義では `if` を使わず `cond` を好んで使っているのでしょうか？

(delete-queue! q) の結果:



クイズ

あと, (delete-queue! q) を3回行くと, どんな状態になるでしょう?

3.3.3 table を表現する



3.3.3 表 (table) を表現する

¹ H																	² He
氢																	氦
³ Li	⁴ Be											⁵ B	⁶ C	⁷ N	⁸ O	⁹ F	¹⁰ Ne
锂	铍											硼	碳	氮	氧	氟	氖
¹¹ Na	¹² Mg											¹³ Al	¹⁴ Si	¹⁵ P	¹⁶ S	¹⁷ Cl	¹⁸ Ar
钠	镁											铝	硅	磷	硫	氯	氩
¹⁹ K	²⁰ Ca	²¹ Sc	²² Ti	²³ V	²⁴ Cr	²⁵ Mn	²⁶ Fe	²⁷ Co	²⁸ Ni	²⁹ Cu	³⁰ Zn	³¹ Ga	³² Ge	³³ As	³⁴ Se	³⁵ Br	³⁶ Kr
钾	钙	钪	钛	钒	铬	锰	铁	钴	镍	铜	锌	镓	锗	砷	硒	溴	氪
³⁷ Rb	³⁸ Sr	³⁹ Y	⁴⁰ Zr	⁴¹ Nb	⁴² Mo	⁴³ Tc	⁴⁴ Ru	⁴⁵ Rh	⁴⁶ Pd	⁴⁷ Ag	⁴⁸ Cd	⁴⁹ In	⁵⁰ Sn	⁵¹ Sb	⁵² Te	⁵³ I	⁵⁴ Xe
铷	锶	钇	锆	铌	钼	锝	钌	铑	钯	银	镉	铟	锡	锑	碲	碘	氙
⁵⁵ Cs	⁵⁶ Ba	lanthanoid	⁷² Hf	⁷³ Ta	⁷⁴ W	⁷⁵ Re	⁷⁶ Os	⁷⁷ Ir	⁷⁸ Pt	⁷⁹ Au	⁸⁰ Hg	⁸¹ Tl	⁸² Pb	⁸³ Bi	⁸⁴ Po	⁸⁵ At	⁸⁶ Rn
铯	钡		铪	钽	钨	铼	锇	铱	铂	金	汞	铊	铅	铋	钋	砹	氡
⁸⁷ Fr	⁸⁸ Ra	actinoid	¹⁰⁴ Rf	¹⁰⁵ Db	¹⁰⁶ Sg	¹⁰⁷ Bh	¹⁰⁸ Hs	¹⁰⁹ Mt	¹¹⁰ Uuq	¹¹¹ Uuu	¹¹² Uub		¹¹⁴ Uuq		¹¹⁶ Uuh		
钫	镭		钆	铈	镨	铀	镅	镆	-	-	-		-		-		

lanthanoid	⁵⁷ La	⁵⁸ Ce	⁵⁹ Pr	⁶⁰ Nd	⁶¹ Pm	⁶² Sm	⁶³ Eu	⁶⁴ Gd	⁶⁵ Tb	⁶⁶ Dy	⁶⁷ Ho	⁶⁸ Er	⁶⁹ Tm	⁷⁰ Yb	⁷¹ Lu
	镧	铈	镨	钆	铕	钐	铕	钆	铽	镱	铥	铒	铥	镱	镱
actinoid	⁸⁹ Ac	⁹⁰ Th	⁹¹ Pa	⁹² U	⁹³ Np	⁹⁴ Pu	⁹⁵ Am	⁹⁶ Cm	⁹⁷ Bk	⁹⁸ Cf	⁹⁹ Es	¹⁰⁰ Fm	¹⁰¹ Md	¹⁰² No	¹⁰³ Lr
	锕	钍	镤	铀	镎	钚	镅	镆	锫	锿	镄	镆	镈	镉	铹

表 (table) を表現する

表: (いくつかの) **検索キー** から, それに関連づけられた **情報** を与えるデータ構造

- 辞書 (キー: 見出し語, 情報: 語の説明)
- 情報学科時間割表 (キー: 曜日, 時限, 情報: 科目名, 担当者, 教室)

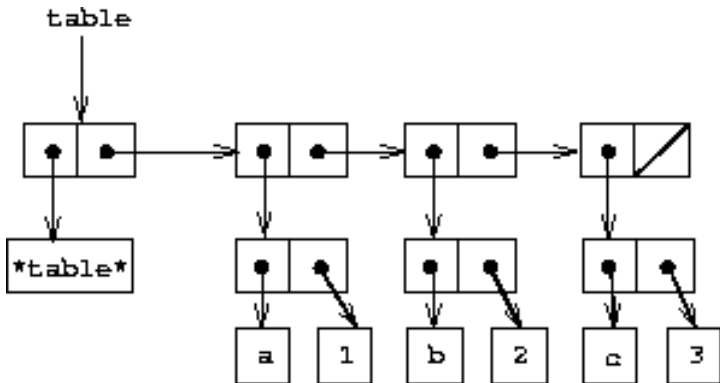
実装のアイデア

- 1次元の表 = キーと情報のペアのリスト (連想リスト)
- n 次元の表 = 情報を $n - 1$ 次元の表とする
- プラスひとひねり

表の操作

- コンストラクタ
 - ▶ (make-table): 新しい空の表を作る
- セレクタ
 - ▶ (lookup key table): table を key をキーとして検索
- ミューテータ
 - ▶ (insert! key value table): table に (key, value) のペアを追加 (すでにあれば更新)

(1次元の)表の構造



- backbone
- 「ひとひねり」のためのダミーエントリー

補助関数 assoc

たいていの Scheme 処理系で最初から用意されている

```
(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (caar records))
         (car records))
        (else (assoc key (cdr records)))))
```

クイズ

(define x '((a . 1) (b . 2) (c . 3))) をした .

- (assoc 'b x) の値は?
- (assoc 'd x) の値は?

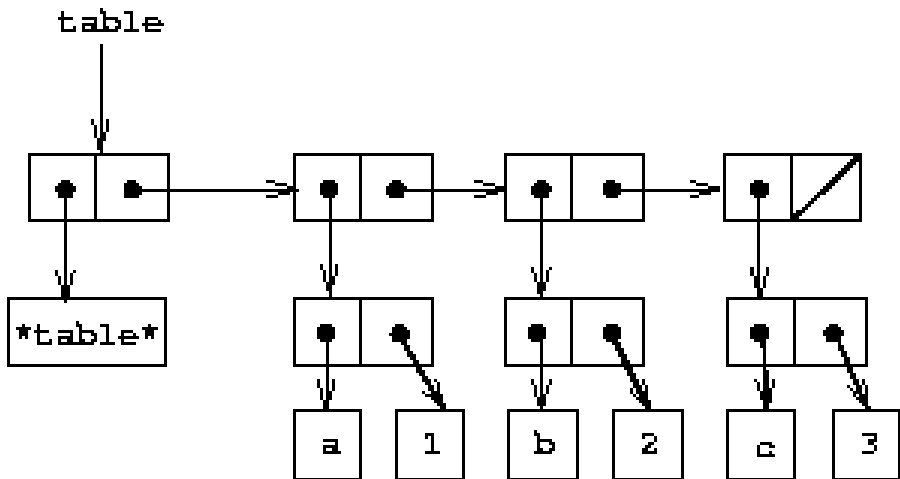
セクタ

```
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        false)))
```

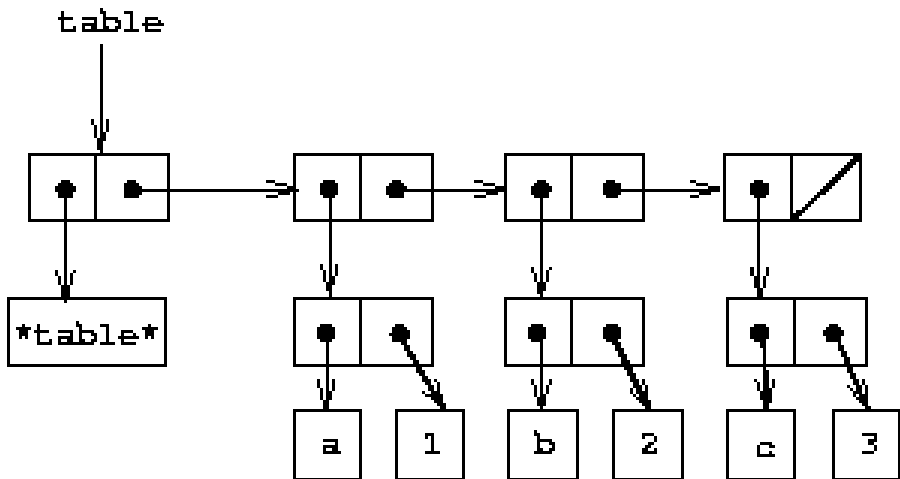

ミューテータ

```
(define (insert! key value table)
  (let ((record (assoc key (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons key value)
                        (cdr table)))))
  'ok)
```

(insert! 'b 4 table) の動作の説明



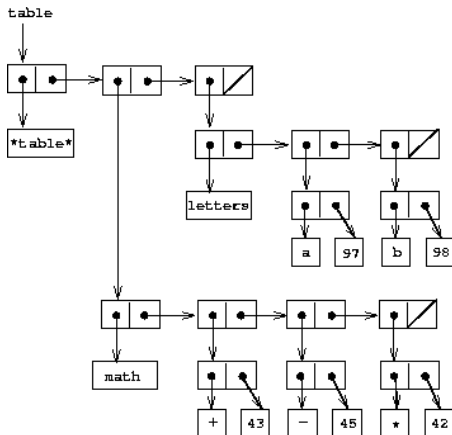
(insert! 'd 4 table) の動作の説明



コンストラクタ

```
(define (make-table)
  (list '*table*))
```

2次元の表



- ふたつめのキーの種類が，ひとつめのキーによって違う
 - ▶ ので「表」から連想するものちょっと違う？

セレクタ

```
(define (lookup key-1 key-2 table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record
                (assoc key-2 (cdr subtable))))
          (if record
              (cdr record)
              false))
        false)))
```

ミューテータ

```
(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        ;; ひとつめのキーを発見
        ;; 残りは insert! と同じ
        (let ((record
              (assoc key-2 (cdr subtable))))
          (if record
              (set-cdr! record value)
              (set-cdr! subtable
                        (cons (cons key-2 value)
                              (cdr subtable)))))
            (set-cdr! table ...))
```

ミューテータ

```
(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let (...)
          ;; ひとつめのキーが見つからない
          (set-cdr! table
                    (cons (list key-1
                                (cons key-2 value))
                          (cdr table))))))
  'ok)
```


表のオブジェクト化

- 表を操作関数の引数とせず操作関数群が共有する局所変数にする
- 銀行口座と同じアイデア
- 表を想定外の操作で「壊す」ことがなくなる

```
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key-1 key-2) ...)
    (define (insert! key-1 key-2 value) ...)
    (define (dispatch m) ...)
    dispatch))
```

予習: 3.3.5 制約の伝播

- 制約ネットワークとは何か？
- 講義ホームページから 3.3.5 のコードをダウンロードして，摂氏・華氏変換の制約ネットワークの例を実行してみよう
 - ▶ 最初は教科書にある制約ネットワークのための関数定義
 - ▶ 最後の方に教科書の実行例コード (コメントアウトされている)

(注) 3.3.4 はとばします

宿題： 5/20(火) 午前 10:30 締切

- Ex. 3.11, 3.12, 3.22, 3.25
- レポートには
 - ▶ 考え方の説明
 - ▶ プログラムリストと考え方の対応
 - ▶ 実行例を示すこと
- レポート (pdf) とプログラムファイルを提出システムを通じて提出
- 友達に教えてもらったなら、その人の名前を明記
- web は出典を明記 (「同じ」回答は減点)