プログラミング言語(SICP)
3. Modularity, Objects, and State
3.3 Modeling with Mutable Data

奥乃 博 (3章)
大学院情報学研究科知能情報学専攻
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/12/ProgLang/
{okuno, igarashi}@i.kyoto-u.ac.jp

TAの居室は10号館4階奥乃1研, 2研, ソ基分野
糸原 達彦(M2) 奥乃研・音楽ロボットG
柳楽 浩平(M2) 奥乃研・ロボット聴覚G

NoStudent Left Behind

4月25日・本日のメニュー

3.3 Modeling with Mutable Data
3-3-1 Mutable List Structure
3-3-2 Representing Queues
3-3-3 Representing Tables
3-3-4 A Simulator for Digital Circuits
3-3-5 Propagation of Constraints

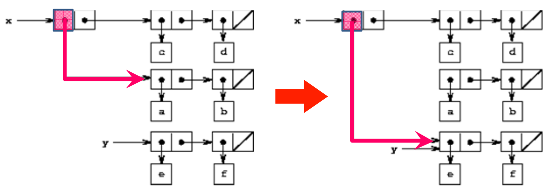
NoStudent Left Behind

3.3 書換え可能データのモデル

- Compound data: 複数のパーツから構成. 実世界モデル化のための計算オブジェクト構築手段
- Non-mutable compound data (書換えのない合成データ):
  - Constructor, selectors
- **Mutable compound data** (書換え可能データ):
  - constructor, selectors, **mutators**
  - (**set-balance!** 〈account〉 〈new-value〉)
- Chapter 2 では **pair** を使ってcompound data 作成
- Chapter 3 でも **pair** を使ってmutable compound data 作成

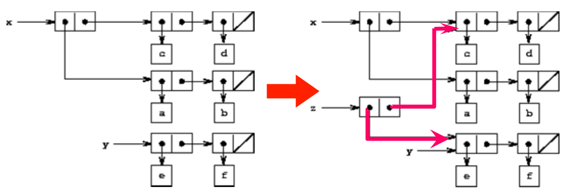
3.3.1 書換え可能なリスト構造

- Mutable compound data (書換え可能データ):
  - constructor, selectors: **cons**, **car**, **cdr**
  - mutators: **set-car!**, **set-cdr!**
- 例: リスト x: **((a b) c d)**, y: **(e f)**
  (set-car! x y) ⇒



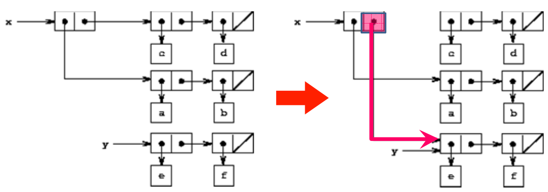
単調(非破壊的)合成データの構築

- 例: リスト x: **((a b) c d)**, y: **(e f)**
  (define z (cons y (cdr x)))
  ⇒
- 【問】**(set-car! x y)** を実行後
  (eq? z x) (equal z x) のそれぞれの値は?



破壊的合成データの構築

- 例: リスト x: **((a b) c d)**, y: **(e f)**
  (set-cdr! x y) ⇒



Cons の新しい実装

consは新しいペアを用意して，その内容を設定する.

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

【余談】 Lispでは

```
     set-car!  は   replaca
     set-cdr!  は   replacd
```

Exercise 3.12. append!

```
(define (append x y)
   (if (null? x)
      y
      (cons (car x) (append (cdr x) y)) ))
```

Append forms a new list by successively consing the elements of x onto y. The procedure append! is similar to append, but it is a mutator rather than a constructor. It appends the lists by splicing them together, modifying the final pair of x so that its cdr is now y. (It is an error to call append! with an empty x.)

```
(define (append! x y)
   (set-cdr! (last-pair x) y)
   x )
```

Here last-pair is a procedure that returns the last pair in its argument:

```
(define (last-pair x)
  (if (null? (cdr x)) x (last-pair (cdr x))) )
```

append と append! との違い

Consider the interaction

```
(define x (list 'a 'b))
(define y (list 'c 'd))
(define z (append x y))
z
(a b c d)
(cdr x)
<response>

(define w (append! x y))
w
(a b c d)
(cdr x)
<response>
```

What are the missing <response>s? Draw box-and-pointer diagrams to explain your answer.

Exercise 3.13. cyclic lsit

```
(define (make-cycle x)
   (set-cdr! (last-pair x) x)
   x)
```

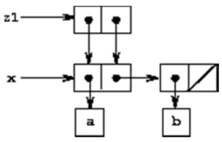
Draw a box-and-pointer diagram that shows the structure z created by

```
(define z (make-cycle (list 'a 'b 'c)))
```

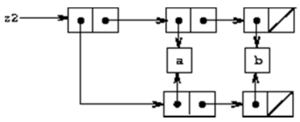
What happens if we try to compute (last-pair z)?

共有と同一性

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

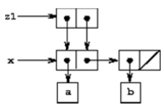



```
(define z2
   (cons (list 'a 'b) (list 'a 'b)))
z2
((a b) a b)
```

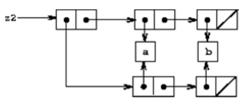


書換え可能であれば，共有かどうかが重要となる

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

z1
((a b) a b)

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

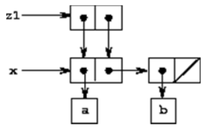



```
z2
((a b) a b)

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

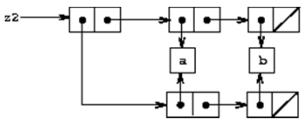


sameness: eq? は同一かどうかを調べる

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




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



書換えは代入と同じ

ペアは手続きとして定義できる(2.1.3節)

```
(define (cons x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (error "Undefined
              operation -- CONS" m ))))
  dispatch )
(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
```

書換えは代入として定義できる

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else (error "Undefined operation –
                            CONS" m ))))
  dispatch)
(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
(define (set-car! z new-value)
  ((z 'set-car!) new-value)
  z)
(define (set-cdr! z new-value)
  ((z 'set-cdr!) new-value)
  z)
```

Exercise 3.20.

Draw environment diagrams to illustrate the evaluation of the sequence of expressions

```
(define x (cons 1 2))
(define z (cons x x))
(set-car! (cdr z) 17)
(car x)
17
```

using the procedural implementation of pairs given above. (Compare exercise 3.11.)

3.3.2 待ち行列の表現

操作	待ち行列の内容
`(define q (make-queue))`	
`(insert-queue! q 'a)`	`a`
`(insert-queue! q 'b)`	`a b`
`(delete-queue! q)`	`b`
`(insert-queue! q 'c)`	`b c`
`(insert-queue! q 'd)`	`b c d`
`(delete-queue! q)`	`c d`

【余談】**enqueue, dequeue** を使うこともある.

3.3.2 queue というデータ構造

constructor:

`(make-queue)`

selectors:

`(empty-queue?` 〈queue〉`)`

`(front-queue` 〈queue〉`)`

mutators:

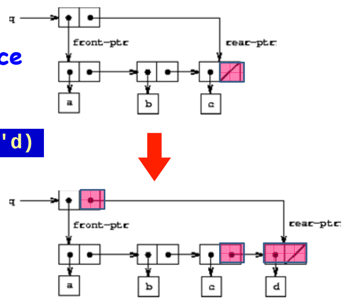
`(insert-queue!` 〈queue〉 〈item〉`)`

`(delete-queue!` 〈queue〉`)`

(insert-queue q 'd) の処理の概念

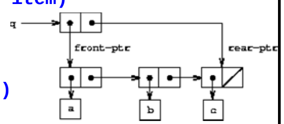
- 2種類のポインターで管理するsequenceで実装
  front-ptr
  rear-ptr
- Queue本体はsequence

(insert-queue! q 'd)

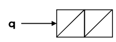


データ構造 queue の実装

```
(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))
```



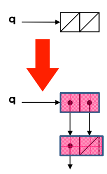

```
(define (empty-queue? queue)
   (null? (front-ptr queue)))

(define (make-queue) (cons '() '()))
```



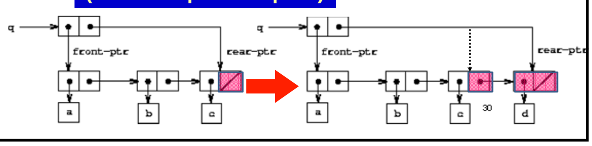

```
(define (front-queue queue)
   (if (empty-queue? queue)
      (error "FRONT called with
              an empty queue" queue )
      (car (front-ptr queue)) ))
```

(insert-queue! queue item) の処理

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (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 ))))
```



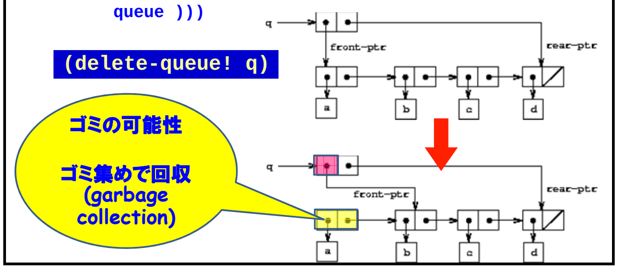
(insert-queue! q 'd)



(delete-queue queue item) の処理

```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty
                 queue" queue ))
        (else
         (set-front-ptr! queue
                         (cdr (front-ptr queue)) )
         queue )))
```

(delete-queue! q)



Exercise 3.21. print-queue

Ben Bitdiddle decides to test the queue implementation described above. He types in the procedures to the Lisp interpreter and proceeds to try them out:

```
(define q1 (make-queue))
(insert-queue! q1 'a)
((a) a)
(insert-queue! q1 'b)
((a b) b)
(delete-queue! q1)
((b) b)
(delete-queue! q1)
(() b)
```

Define a procedure **print-queue** that takes a queue as input and prints the sequence of items in the queue.

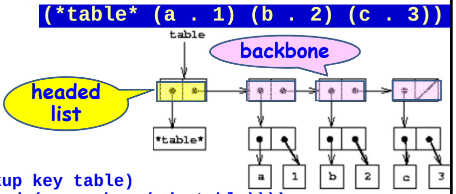
3.3.3 表の表現 (tables as mutable list)

1個のキーによる(連想)表

```
a:  1
b:  2
c:  3
```

1次元表で実現

(*table* (a . 1) (b . 2) (c . 3))

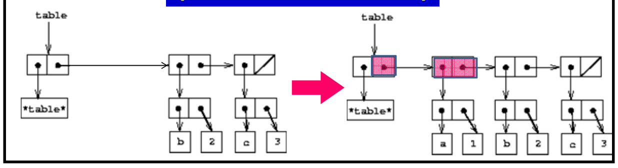



```
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        false )))

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

insert! と make-table

```
(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)

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

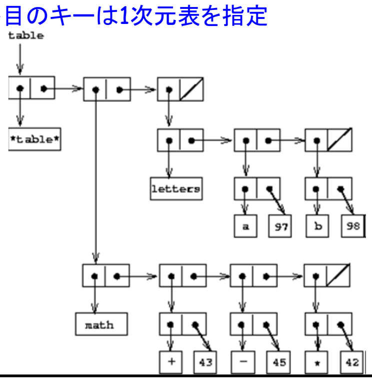
(insert! 'a '1 table)



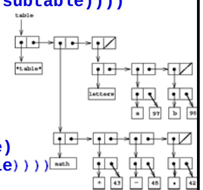
Two-dimensional tables（2次元の表）

2つのキー： 1番目のキーはsubtable 指定
2番目のキーは1次元表を指定

math:
+: 43
-: 45
*: 42
letters:
a: 97
b: 98




```
(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 )))

(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (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
             (cons (list key-1 (cons key-2 value))
                                (cdr table)))))
  'ok )
```



Creating local tables（局所的な表を作る）

lookup と insert! をtable ごとに定義したい

```
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key-1 key-2)
      (let ((subtable (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record (assoc key-2 (cdr subtable))))
              (if record (cdr record) false))
            false)))
    (define (insert! key-1 key-2 value)
      (let ((subtable (assoc key-1 (cdr local-table))))
        (if subtable
            (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! local-table
                    (cons (list key-1 (cons key-2 value))
                          (cdr local-table) ))))
      'ok)
    (define (dispatch m)
      (cond ((eq? m 'lookup-proc) lookup)
            ((eq? m 'insert-proc!) insert!)
            (else (error "Unknown operation -- TABLE" m)) ))
    dispatch ))
```

Creating local tables の別の方法

make-table を使った get と put の定義

```
(define operation-table (make-table))
(define get
    (operation-table 'lookup-proc))
(define put
    (operation-table 'insert-proc!))
```

使用法は

(get 〈key-1〉 〈key-2〉)
(put 〈key-1〉 〈key-2〉〈value〉)

Ex.3.27. Memoization (aka tabulation)

```
(define (fib n)
   (cond ((= n 0) 0)
         ((= n 1) 1)
         (else (+ (fib (- n 1)) (fib (- n 2)))) ))
```

The memoized version of the same procedure is

```
(define memo-fib
   (memoize (lambda (n)
              (cond ((= n 0) 0)
                    ((= n 1) 1)
                    (else (+ (memo-fib (- n 1))
                             (memo-fib (- n 2)) ))))))
(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previously-computed-result
              (lookup x table)) )
        (or previously-computed-result
            (let ((result (f x)))
              (insert! x result table)
              result ))))))
```

宿題：5月1日午前8時 締切

NoStudent Left Behind

1. Ex. 3.21, Ex. 3.22, Ex. 3.23, Ex.3.27
2. レポートには アルゴリズムの説明，プログラムリストを示してアルゴリズムとの対応を説明し，実行例を示してプログラムが正しく動いていることを示すこと．
3. レポート(PDF)とプログラムファイルを送付

   PROG-3@zeus.kuis.kyoto-u.ac.jp

   file 名は 学籍番号-名前-3.pdf

- **友達に教えてもらったら，その人の名前を明記すること．Webは出展を明記．**（otherwise『同じ』回答は減点）

DON'T PANIC!

47