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 1, 2013

今日のメニュー

- 前回の宿題

3.2 評価の環境モデル

3.2.4 内部定義

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

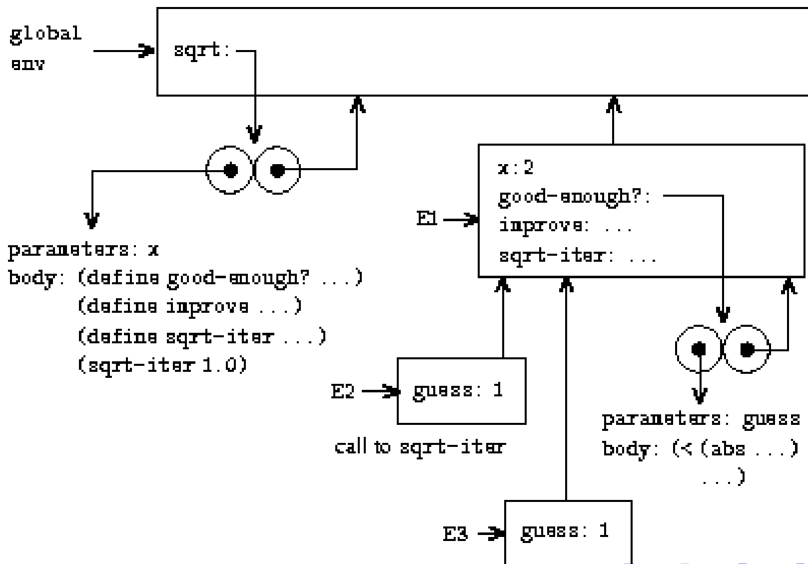
3.3.2 キュー (queue) を表現する

3.3.3 表 (table) を表現する

3.2.4 内部定義

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
(sqrt 2)
```

(sqrt 2) の評価中，最初の good-enough? 呼び出し直後の様子



評価の環境モデルまとめ

- 局所定義された手続きで使われる名前と外側の名前が干渉しない
  - $\Leftarrow$ 束縛されるフレームが違うため
- 局所定義された手続き内では，shadowing がない限り外側で定義された名前が参照できる
  - 定義を表すフレームが内側 (先頭) から数珠繋ぎになっていて，順に探索していくため
- `set!` は束縛の書き換え

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

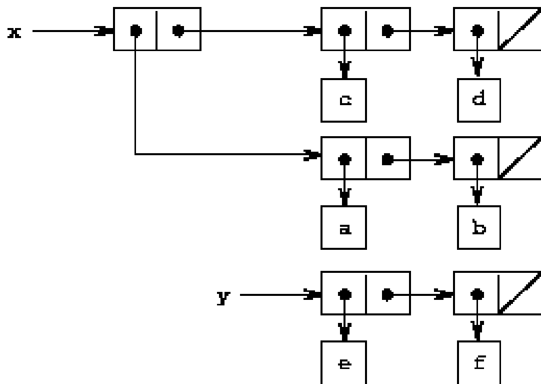
原始的なミューテータ `set-car!`, `set-cdr!`

```
(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

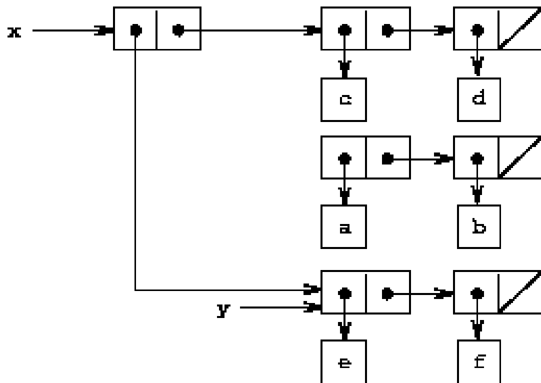
例(1/4)

```
(define x '((a b) c d))
(define y '(e f))
```



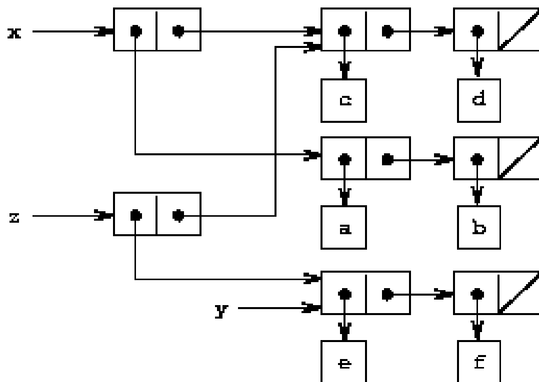
例 (2/4)

`(set-car! x y)` **をすると…**



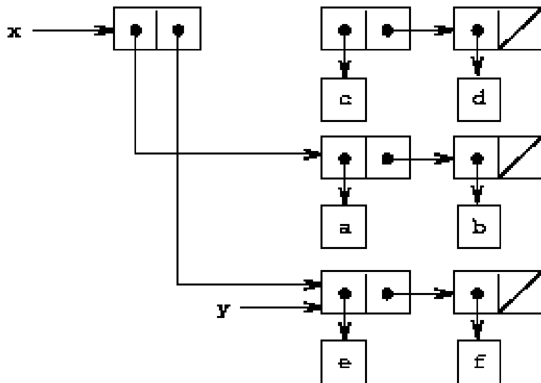
例 (3/4)

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

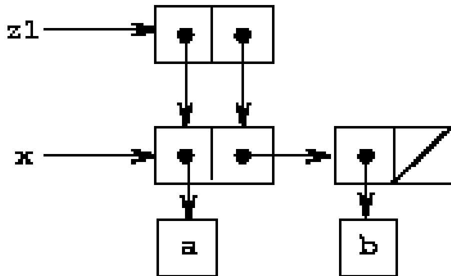
と考えることができる．あえて書くなら

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

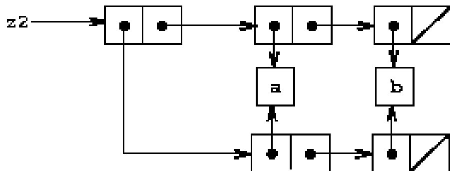
共有と同一性

- 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)
```

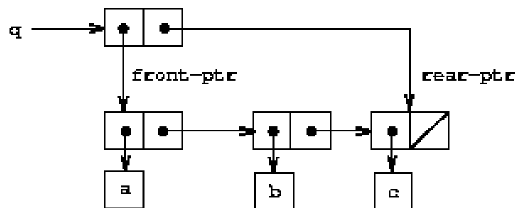
3.3.2 キュー(queue)を表現する

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

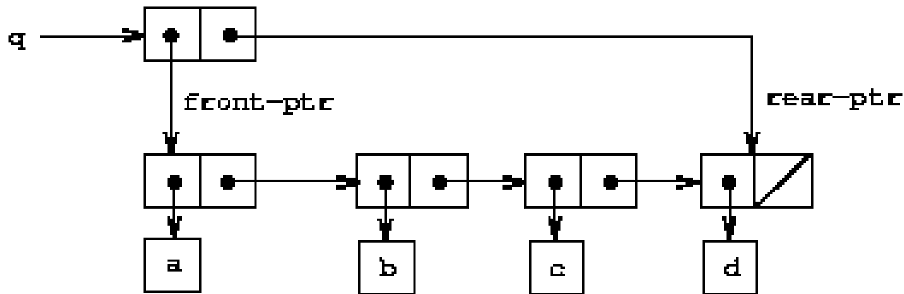
```
(define (empty-queue? queue)
  (null? (front-ptr queue)))
(define (make-queue) (cons '() '()))

(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:



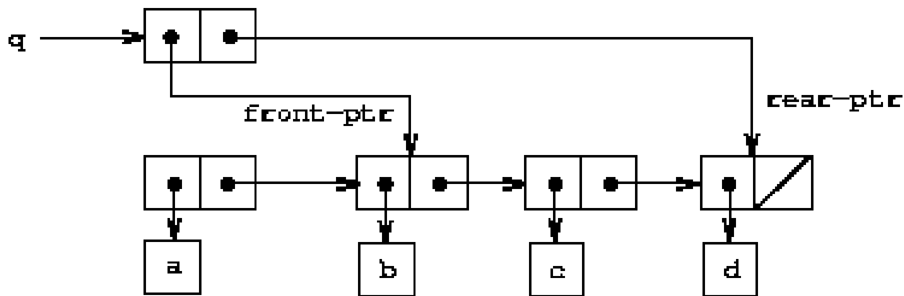
ミューテータ (2/2)

```
(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？**

(delete-queue! q) の結果:



クイズ

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

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

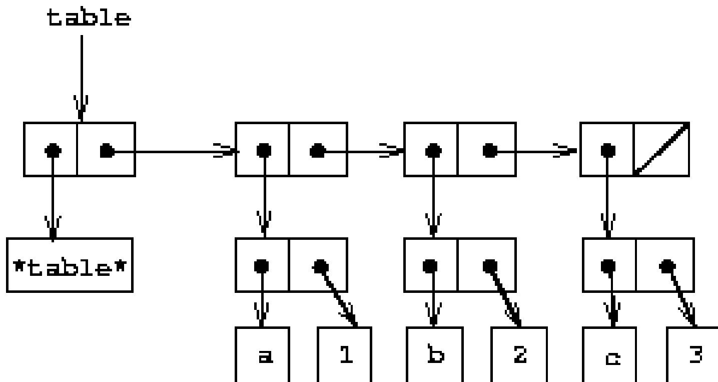
実装のアイデア

- 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

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？**

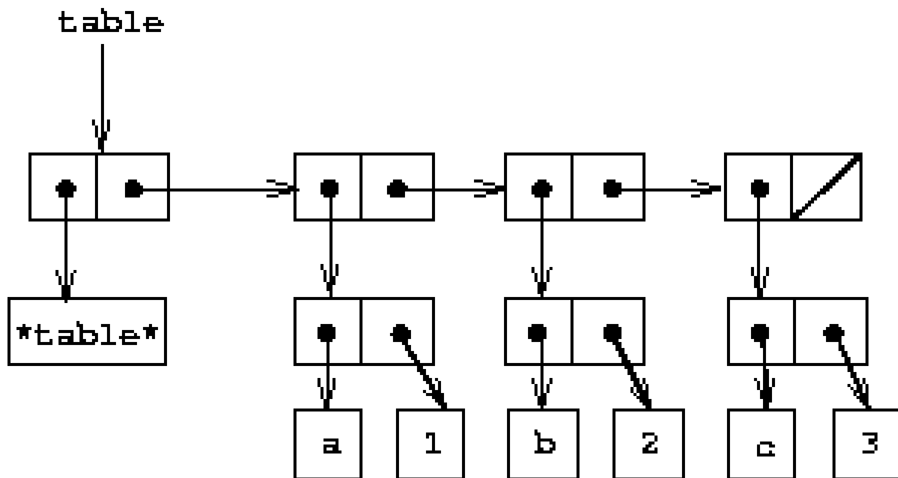
セレクタ

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

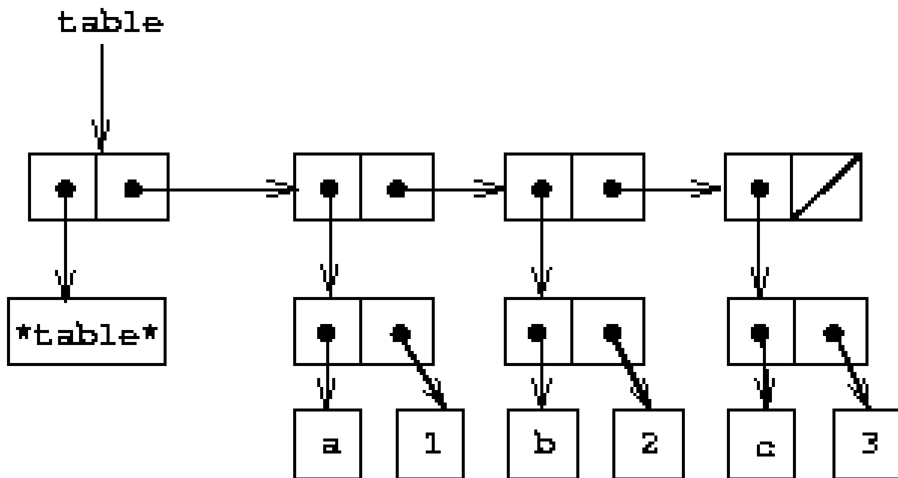
ミューテータ

```scheme
(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



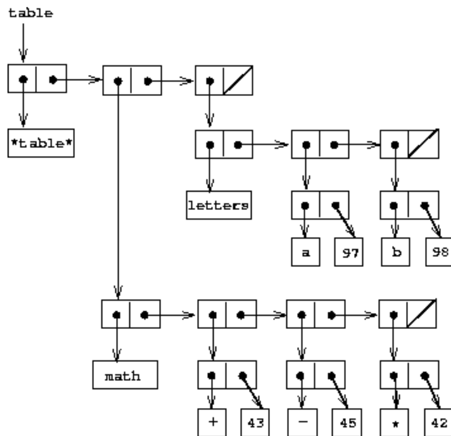
(insert! 'd 4 table) の動作の説明



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」から連想するものとちょっと違う？

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： 5/15(水) 午前8時 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ら、その人の名前を明記
- web は出典を明記 (「同じ」回答は減点)