プログラミング言語論第3回
データによる抽象化 (1)

情報工学科 山本修身

前回の復習 (1)

いくつかの命令を束ねて1つの命令にする．与えられた式が順々に評価されて，最後の式の値がこのbegin式の値となる．

```
(begin 式1 式2 式3 ... 式n)
```

名前のない関数をその場で作るにはlambdaを用いる．

```
(lambda (引き数列) 本体)
```

実は以下のように考えることもできたりする．

```
(begin 式1 ... 式n) = ((lambda () 式1 ... 式n))
```

defineで関数を定義することはじつは，以下のように変数にlambda式の値を結びつけることに他ならない．

```
(define (関数名 引数1 ... 引数n) 本体)
= (define 関数名 (lambda (引数1 ...引数n) 本体))
```

前回の復習 (2)

関数定義の一番最後の式がその関数自身の呼び出しになっていて，それ以外に関数自身を呼び出していないような再帰呼び出しを末尾再帰 (tail recursion) と呼ぶ．Scheme言語やいくつかの言語では末尾再帰は繰返しとして動作する（末尾再帰の最適化）．

1からnまでの和を返す関数sumの定義：

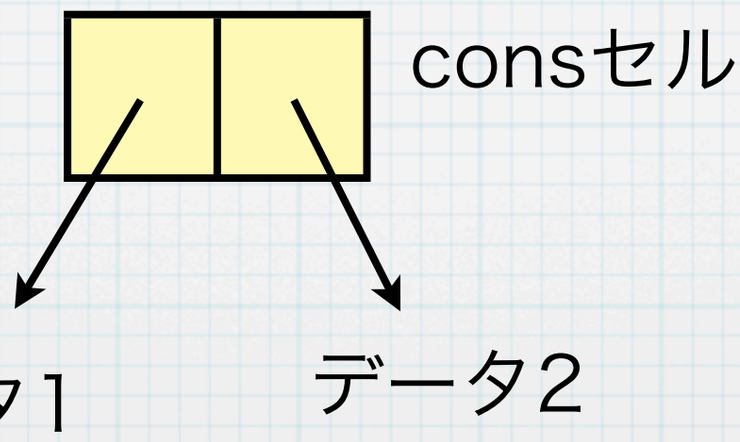
```
(define (sum n)
  (define (sum-iter i n res)
    (if (> i n) res
     (sum-iter (+ i 1) n (+ res i))))
  (sum-iter 1 n 0))

(display (sum 100))
(newline)
```

```
osami-2:yama517> kawa sum.scm
5050
```

データを抽象的に扱う (1)

われわれが定義した関数で扱えるデータは整数や文字列，ブール値などであった．これらのデータをいつもそのまま扱わなければならないとすると，複雑な複合データを扱うときとても煩雑である．そのため，複数のデータを繋いで複雑なデータを構成する方法について考える．

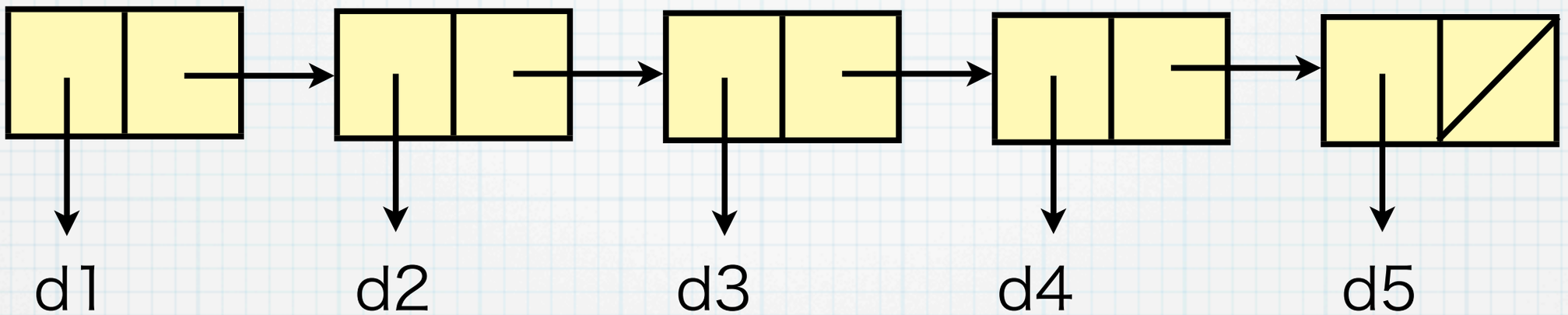
2つのデータを結んで1つのデータにするには，以下のような構造を作る．この繋げた四角のデータをconsセルと呼ぶ．



また，このconsセルをS式では (データ1 . データ2) のようにドットを挟んで括弧で括ったものとして表現する．

データを抽象的に扱う (2)

consセルを多重に重ねることで多くのデータを扱うことができる.



上記の構造を**線形リスト**と呼ぶ. 線形リストの最後のセルの右側の斜線は特殊なデータ ( ) （null値）が入っており, 何も入っていないことを表す.

上記の線形リストを前のスライドの方法でS式で表現すれば, 以下のようになる.

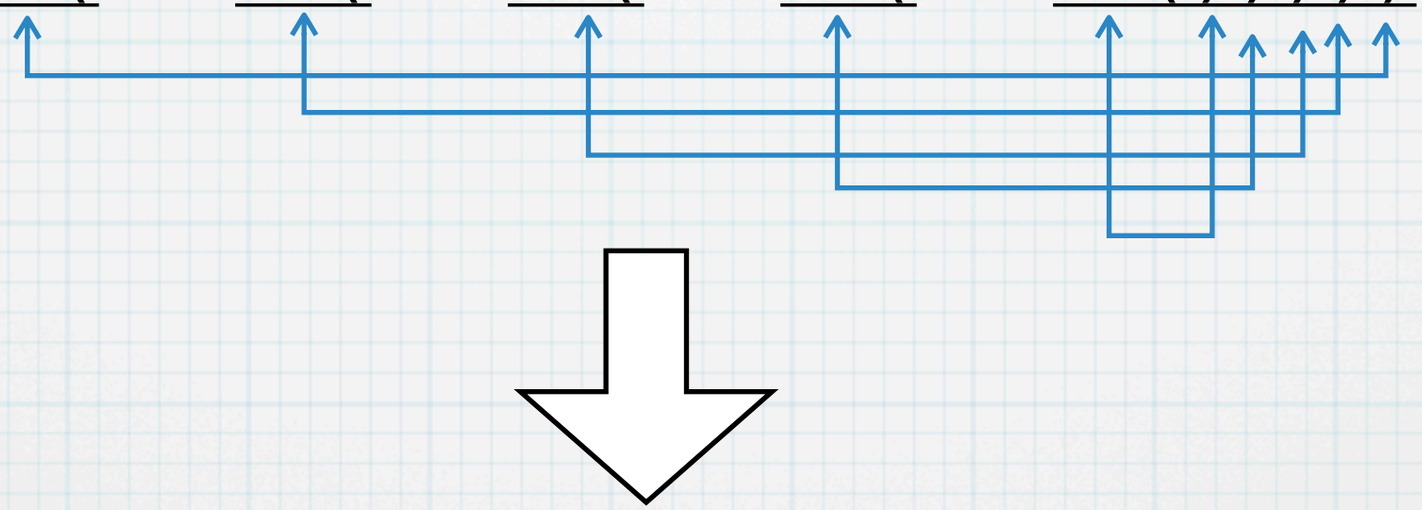
```
(d1 . (d2 . (d3 . (d4 . (d5 . ())))))
```

データを抽象的に扱う (3)

```
(d1 . (d2 . (d3 . (d4 . (d5 . ()))))))
```

上記のように線形リストを表現すると，括弧がとても多くなるので，これを表現する場合，.(と それに対応する) を省略できるというルールを設定する.

```
(d1 . (d2 . (d3 . (d4 . (d5 . ()))))))
```



```
(d1     d2     d3     d4     d5                )
```

このような線形リストのことを単に**リスト (list)** と呼ぶ.

consセルを操作するための関数

2つのデータからconsセルを作る関数：

```
(cons データ1 データ2)
```

consセルの左側を取り出す関数：

```
(car consセル)
```

「カー」と呼ぶ

consセルの右側を取り出す関数：

```
(cdr consセル)
```

「クダー」と呼ぶ

与えられた要素から線形リストを作る関数：

```
(list データ1 データ2 ... データn)
```

consセルを用いてフィボナッチ数を計算する (1)

フィボナッチ数は以下の式で定義された：

$$f(n+2) = f(n+1) + f(n), f(0) = f(1) = 1$$

前述のように，これをそのままプログラムすると，末尾再帰として計算することができないし，非常に多くの計算時間を必要とした．そこで，

$$[f(n+1), f(n)] = [f(n) + f(n-1), f(n)]$$

として，関数値のペアーを計算することを考える．

```
(define (ff n)
  (if (= n 0) (cons 1 1)
      (begin
        (define a (ff (- n 1)))
        (cons (+ (car a) (cdr a)) (car a)))))
```

```
(display (ff 38))
(newline)
```

→

```
osami-2:yama537> kawa fib1.scm
(102334155 . 63245986)
```

consセルを用いてフィボナッチ数を計算する (2)

前述のプログラムは完全な末尾再帰になっていないので，末尾再帰にするには以下のようにする.

```
(define (ff n)
  (define (ff-iter i n ans)
    (if (= i n) ans
        (ff-iter (+ i 1) n (cons (+ (car ans) (cdr ans)) (car ans)))))
  (ff-iter 0 n (cons 1 1)))
```

実際には以下のようにほとんどconsセルを使わないでプログラミングすることが可能である.

```
(define (ff n)
  (define (ff-iter i n ans1 ans2)
    (if (= i n) (cons ans1 ans2)
        (ff-iter (+ i 1) n (+ ans1 ans2) ans1)))
  (ff-iter 0 n 1 1))
```

クオートの導入 (1)

Schemeではあらゆる S式は評価される際，(...) の形になっていれば，関数を呼び出して関数値を計算する（この動作をapply (アプライ)という）．しかし，書いたS式をデータとしてそのまま使いたい場合がある．その場合，quote という特殊形式を用いる．

```
(quote S式)
```

たとえば，(1 2 3) というリストをデータとしてプログラム中で使いたければ，(quote (1 2 3)) と書けば良い．quoteは頻繁に使われるしくみなので，普通は括弧で括らなくても ' (シングルクオート) をS式の先頭に置くことで，quoteを用いたことになるように決められている．

```
#|kawa:1|# (quote (1 2 3))
(1 2 3)
#|kawa:2|# '(1 2 3)
(1 2 3)
```

クオートの導入 (2)

たとえば，リストを作成する際，(list 1 2 3 4) としても'(1 2 3 4) と直接リストを書いてクオートしても結果は同じである.

```
#|kawa:4|# (list 1 2 3 4)
(1 2 3 4)
#|kawa:5|# (quote (1 2 3 4))
(1 2 3 4)
#|kawa:6|# '(1 2 3 4)
(1 2 3 4)
```

クオートを用いれば，線形リストの末端の値は '() と表現することができる.

```
#|kawa:8|# '()
()
#|kawa:9|# ()
()
#|kawa:10|# (cons 2 '())
(2)
```

多くのSchemeではクオートをつけなくてもエラーにならない

nullを評価してもnullになってしまうということ．数字も同様にクオートを付ける必要はない

シンボル：quoteによって生成されるもの

quoteを導入すると，われわれの扱うデータとして今までになかったものが追加される．普通のシンボルは通常変数として扱われる．たとえば，(define a 10) とすれば，aは変数であり評価すると10という値に変化する．これに対して (quote a) すなわち ‘a は評価することによって，a が出てくる．この a はシンボルとよばれ，Schemeで扱うことのできるデータである．このデータは文字列とも異なる．

```
(define a 'b)
(define b 'a)

(display a)
(display b)
(newline)
```

```
OMacBook:yama504> kawa symbol.scm
ba
```

四則演算など処理はできないが，何かを表現するために利用することができ，利用価値のあるものである．

nullであるか否かを評価する関数

null値であるかどうかを判断するための関数として，null?が用意されている．

```
(null? データ)
```

データは何でも良い．値が()の場合のみ#tを返す

```
#|kawa:14|# (null? 23)
#f
#|kawa:15|# (null? "abc")
#f
#|kawa:16|# (null? ())
#t
```

リストの長さを測る

ここまでに示した道具でいくつかのリストの操作を記述することができる．まず，リストの長さを測る関数をつくる．

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

```
(display (length '(3 4 3 2 3 4 5)))
(newline)
```

→

```
osami-2:yama547> kawa length.scm
7
```

このプログラムは末尾再帰になっていない．末尾再帰バージョンは以下のとおり．

```
(define (length lst)
  (define (length-iter lst res)
    (if (null? lst) res
        (length-iter (cdr lst) (+ 1 res))))
  (length-iter lst 0))
```

リストの要素の総和を計算する

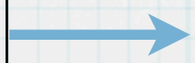
リストの要素が数であるとして，数の総和を計算する関数sumを定義する

```
(define (sum lst)
  (if (null? lst) 0
      (+ (sum (cdr lst)) (car lst))))
```

さらに n - 1から0 までn個の数が並んだリストを作る関数 range–rev を以下のように定義する.

```
(define (range-rev n)
  (if (= n 0) '()
      (cons (- n 1) (range-rev (- n 1)))))
```

```
(display (sum '(2 3 4 5 4 3 4 5)))
(newline)

(display (range-rev 10))
(newline)

(display (sum (range-rev 1000)))
(newline)
```



```
osami-2:yama559> kawa sum.scm
30
(9 8 7 6 5 4 3 2 1 0)
499500
```

リストを逆向きにする

与えられたリストの要素を逆順にしたリストを求める関数 reverse を作る.

```
(define (reverse lst)
  (define (reverse-iter lst ans)
    (if (null? lst) ans
        (reverse-iter (cdr lst) (cons (car lst) ans))))
  (reverse-iter lst '()))
```

この関数を用いて，0からn - 1までの数を並べたリストを返す range 関数を定義することができる.

```
(define (range n) (reverse (range-rev n)))
```

```
(define aa '("yama" 3 4 2 (3 2 4 3)))
(display aa)
(newline)
(display (reverse aa))
(newline)

(display (range 10))
(newline)
```

```
(yama 3 4 2 (3 2 4 3))
((3 2 4 3) 2 4 3 yama)
(0 1 2 3 4 5 6 7 8 9)
```

リストを逆向きにする（ディープバージョン）

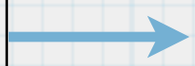
あるデータがリストであるか否かを判定する関数list?はあらかじめ定義されているとする.

```
(list? データ)
```

これを用いて，リストの内容を反転させる関数で，要素がリストであれば，さらにその中身も反転させる関数 deep-reverse は以下のように定義できる.

```
(define (deep-reverse lst)
  (define (deep-reverse-iter lst ans)
    (if (null? lst) ans
        (deep-reverse-iter (cdr lst)
                           (cons
                            (if (list? (car lst))
                                (deep-reverse (car lst))
                                (car lst))
                            ans))))
  (deep-reverse-iter lst '()))
```

```
(define a '((2 3 4) (5 6 7)))
(display a)
(newline)
(display (deep-reverse a))
(newline)
```



```
osami-2:yama568> kawa deep-reverse.scm
((2 3 4) (5 6 7))
((7 6 5) (4 3 2))
```

リストを連結する関数

２つのリストを連結する関数 append は以下のように書くことができる.

```
(define (append lst1 lst2)
  (if (null? lst1) lst2
      (cons (car lst1) (append (cdr lst1) lst2))))
```

```
(display (append '(1 2 3) '(4 5 6)))
(newline)
```

→

```
osami-2:yama573> kawa append.scm
(1 2 3 4 5 6)
```

```
(append (1 2 3) (4 5 6)) = (cons 1 (append (2 3) (4
5 6)) = (cons 1 (cons 2 (append (3) (4 5 6)))) =
(cons 1 (cons 2 (cons 3 (append () (4 5 6))))) =
(cons 1 (cons 2 (cons 3 (4 5 6)))) = (cons 1 (cons 2
(3 4 5 6))) = (cons 1 (2 3 4 5 6)) = (1 2 3 4 5 6)
```

練習問題

○ 多重の深くなっているリストを1層のリストに書き換える関数 flatten を書け．たとえば，flatten は以下のように動作する.

```
(define m '(((2 3 4)) ((4 5 ) (3 4))))
(display m)
(newline)
(display (flatten m))
(newline)
```

→

```
osami-2:yama579> kawa flatten.scm
(((2 3 4)) ((4 5) (3 4)))
(2 3 4 4 5 3 4)
```

○ 線形リストのn番目の要素を取り出す関数 (nth i lst) を定義せよ.

letの導入

局所にのみ利用可能な変数を定義して，局所的に式を評価する特殊形式としてletがある．letは以下のように形になる．いままでは局所的な変数をdefineを用いて定義していたが，あまり良い方法ではない．

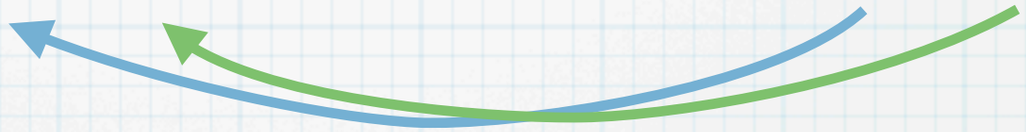
```
(let ((変数1 式1) ... (変数n 式n))
  本体の式1
  本体の式2
  ...
  本体の式m)
```

局所的な場所を作り，それぞれの変数をそこで定義して，そのなかで本体の式1～本体の式m を順に評価する．最後の式が全体の値となる．

letのlambdaによる解釈

let は lambdaを用いて実現することが可能である．以下のように考える．

```
(let ((a 10) (b 20)) (+ a b))
= ((lambda (a b) (+ a b)) 10 20)
```



letで注意しなければいけないのは，定義する変数の順番は存在しないということである．すなわち，この例の場合，aとbは同時に定義される．したがって，bを定義の中でaを参照したりすることはできない．

letを用いることでプログラムは見易くなる

たとえば，appendは以下のように書くことができる．いちいちcarやcdrでデータを分解せずに，最初に分解したものを局所変数としておき，それを利用することができる．

```
(define (append lst1 lst2)
  (if (null? lst1) lst2
      (cons (car lst1) (append (cdr lst1) lst2))))
```

```
(define (append lst1 lst2)
  (if (null? lst1) lst2
      (let ((head (car lst1))
            (tail (cdr lst1)))
        (cons head (append tail lst2)))))

(let ((ans (append '(a b c) '(x y z))))
  (display ans)
  (newline))
```

```
OMacBook:yama514> kawa appendx.scm
(a b c x y z)
OMacBook:yama515>
```

連想リスト（連想配列）(1)

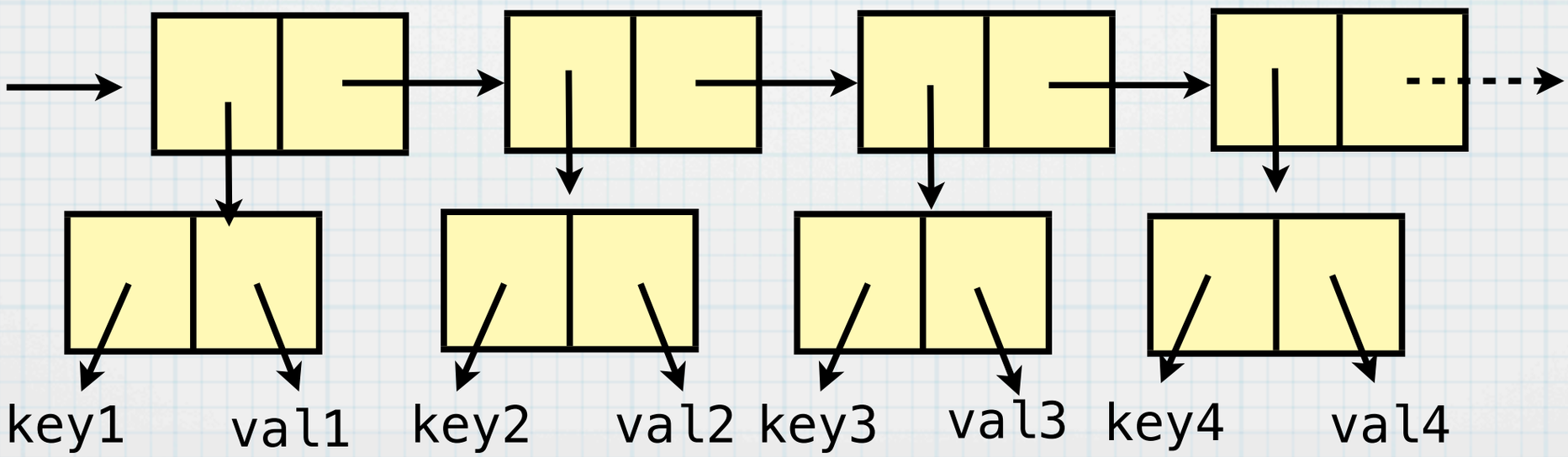
consセルを用いることにより，配列のようなものを作ることができる．ただし，アクセスの効率は良くない．

連想配列は以下のような構造をしている．

キー　値　キー　値

```
((key1 . val1) (key2 . val2) ...
               (keyn . valn))
```

このデータは a[key1] = val1 のように解釈すれば良い．ただし，key1としては，数ばかりではなく，色々なデータをとることができる．



連想リスト（連想配列）(2)

連想リスト内からあるキーに等しい要素を探し出すには，以下のように定義される assoc 関数を用いる.

```
(define (assoc key a)
  (if (null? a) '()
      (let ((ele (car a)))
        (if (eq? (car ele) key) ele
            (assoc key (cdr a))))))
```

```
(let ((alist '((yamamoto . 100)
               (suzuki . 110)
               (yamada . 120)
               (sato . 45)
               (kaneko . 22))))
  (display (assoc 'yamamoto alist)) (newline)
  (display (assoc 'yamada alist)) (newline)
  (display (assoc 'tanaka alist)) (newline))
```

```
OMacBook:yama509> kawa assoc.scm
(yamamoto . 100)
(yamada . 120)
()
```

キーが等しいか否かを判定するのに，eq? という関数を用いている.

これまでは = を用いてきたが，= は数のみに使えるものなので，一般的なデータにはeq?を用いる.

連想リスト（連想配列）(3)

配列に要素を追加するために，acons を定義する

```
(define (acons key val a)
  (cons (cons key val) a))
```

配列に要素の値を変更する

```
(set-cdr! (assoc key a) val)
```

consセルの内容を変更するには，set-car! と set-cdr! を使うことができる．（Schemeでは無理やりデータを書き換えるような命令については，!がついている．）

エラトステネスのふるい

素数の列を効率に計算するためのアルゴリズムにエラトステネスのふるいが知られている.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
#t	#t	#t	#t	#t	#t	#t	#t	#t	#t	#t	#t	#t	#t	#t

初期状態は上記にようにしておき，2からスタートして，順に値が#tの要素を見つけたら，その倍数に対応する要素の値を#fに変えて行く.

2の倍数の消去

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
#t	#t	#t	#t	#f	#t	#f	#t	#f	#t	#f	#t	#f	#t	#f

3の倍数の消去

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
#t	#t	#t	#t	#f	#t	#f	#t	#f	#f	#f	#t	#f	#t	#f

連想配列を用いたエラトステネスのプログラム

連想配列を用いて，エラトステネスのふるいを実現する

```
(define (forx i n step func)
  (when (< i n)
        (func i)
        (forx (+ i step) n step func)))

(define (init-array i n res)
  (if (>= i n) res
      (init-array (+ i 1) n (acons i #t res))))

(define (sieve n)
  (let ((a (init-array 2 n '())))
    (forx 2 n 1
          (lambda (i)
            (if (cdr (assoc i a))
                (begin
                  (display i)
                  (display " ")
                  (forx (+ i i) n i
                        (lambda (j) (set-cdr! (assoc j a) #f)))))))
    (newline)))

(sieve 1000)
```

```
OMacBook:yama514> kawa sieve.scm
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107
109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223
227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337
347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457
461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593
599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719
727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857
859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997
```

練習問題

連想リストは汎用性の高いデータ構造であるが，一列に並んだデータを表現するには，負荷が大きすぎる．一列のデータを表現するには，線形リスト（リスト）を用いるのが自然である．リストを用いてエラトステネスのふるいを実現せよ．

最初に作るのは以下のようなリストである．

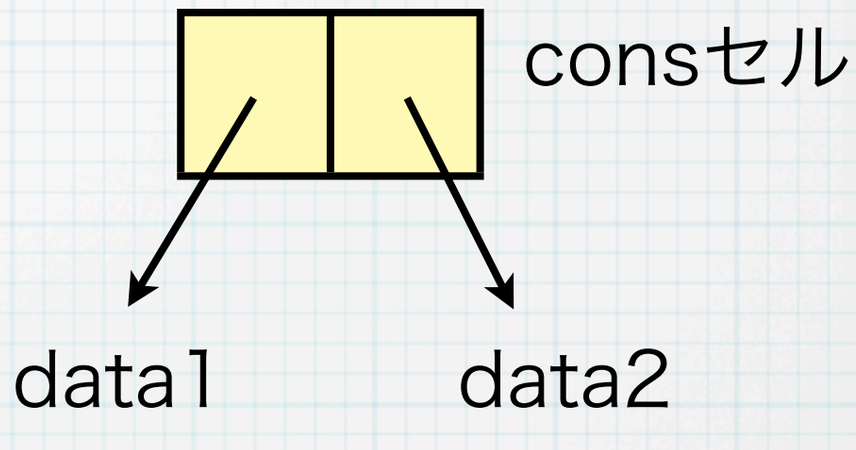
```
(#t #t #t #t #t #t #t #t #t #t #t #t #t ... )
```

このリストを配列のように使ってエラトステネスのふるいを実現すれば良い．

本日のまとめ (1)

2つのデータを組み合せて1つのデータを作るためのデータとして consセルがある．consセルを作るための関数はconsである．

```
(cons data1 data2)
```



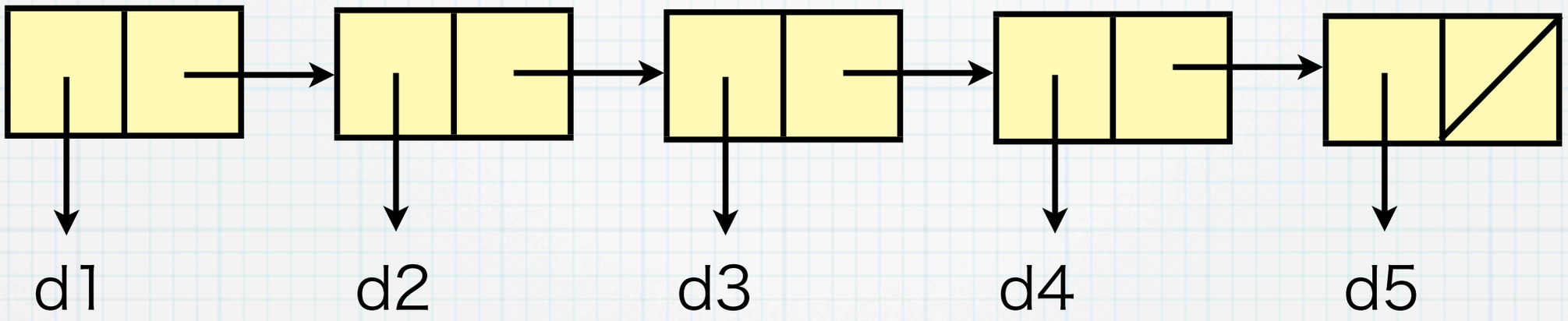
さらに，consセルのそれぞれ左側，右側のデータを取り出す関数として，carとcdrが用意されている．

```
(car pair)

(cdr pair)
```

```
p = (cons data1 data2)

    (car p) = data1
    (cdr p) = data2
```

本日のまとめ (2)

consセルを連ねて，線形リストの構造を作ることができる.



線形リストを直接作る関数として list 関数がある.

```
(list d1 d2 d3 d4 d5)
        ||
(d1 d2 d3 d4 d5)
```

最後のセルの右側には何も入っていないが，これは通常( )と表現され，null値と呼ばれる．空のリストと考えても良い．nullであるか否かを判定する関数 null? が定義されている.

本日のまとめ (3)

与えられたS式を評価しないでそのまま値とするための特殊形式として，quoteが定義されている.

```
(quote  S式)
```

通常，上記形式で書かなくても，'（クオート）をS式の先頭に置くことで同じ意味となる.

本日のまとめ (4)

特殊形式 let によって，局所変数を定義し，letで定義された内部で式を評価することができる.

```
(let ((変数1 式1) ... (変数n 式n))
  本体の式1
  本体の式2
  ...
  本体の式m)
```

本日のまとめ (5)

連想リストを定義し，それを用いることによって，色々な問題を可決することができる.

キー　値　キー　値

```
((key1 . val1) (key2 . val2) ...
               (keyn . valn))
```