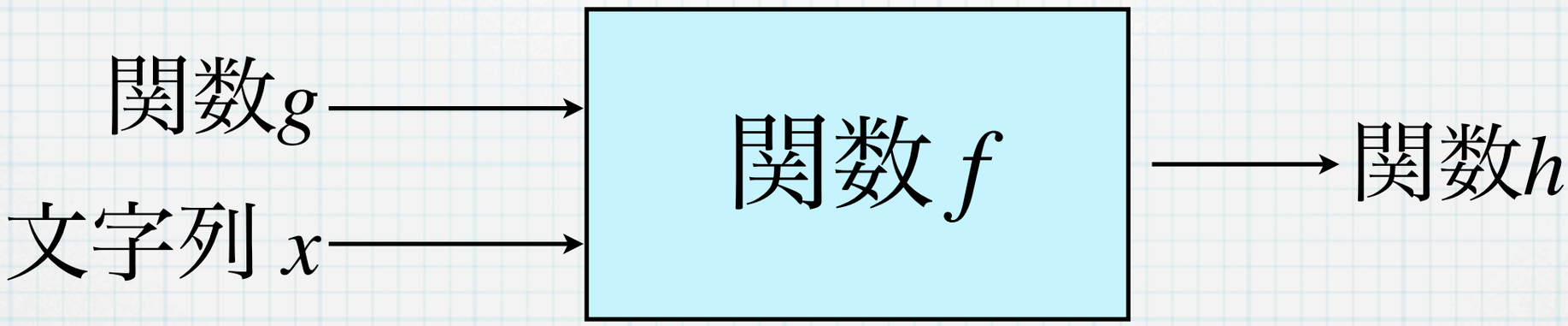
プログラミング言語論第6回

高階手続きによる抽象化

情報工学科 山本修身

高階手続き（高階関数）とは

高階手続き（高階関数）(higher order function) とは，引数として関数を取ったり，返り値として関数を返したりする関数のことである．すでにこの講義で定義した `myfor` などは，高階手続きである．



高階手続きはうまく用いればとても便利なものであり，複雑な動作を簡単に表現することができるようになる．その分，プログラムが凝縮されるので，一部の小さな変更でプログラムの動作が大きく変わることがある．今回は関数から関数を作り出す関数など高階手続きによって何ができるのか考えてみる．

myforの復習

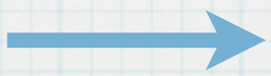
myfor は繰り返しを表現する CなどのfOr文をSchemeで使えるようにするための関数であり，標準のSchemeでは定義されていない，この講義独自のものである．SchemeでCのような繰返し文を書くには便利である．

```
(define (myfor i n func)
  (if (>= i n) '()
      (begin
        (func i)
        (myfor (+ i 1) n func))))
```

```
void kuku(int n){
  int i, j;
  for (i = 0; i < n; i++){
    for (j = 0; j < n; j++)
      printf("%d ", i * j);
    putchar('\n');
  }
}
int main(){ kuku(10); }
```

Cで書いた等価なプログラム

```
(define (kuku n)
  (myfor 0 n
         (lambda (i)
           (myfor 0 n
                  (lambda (j)
                    (display (* i j))))
           (newline))))
(kuku 10)
```



```
OMacBook:yama502> kawa kuku.scm
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 10 12 14 16 18
0 3 6 9 12 15 18 21 24 27
0 4 8 12 16 20 24 28 32 36
0 5 10 15 20 25 30 35 40 45
0 6 12 18 24 30 36 42 48 54
0 7 14 21 28 35 42 49 56 63
0 8 16 24 32 40 48 56 64 72
0 9 18 27 36 45 54 63 72 81
```

リスト全体をついて処理を施す関数 map

myforのリスト版というべき関数として，リストの要素すべてにある処理を施した結果をリストにしたものを返す関数として map がある．Schemeの処理系には map が定義されているが，ここでは改めて map として定義する．

```
(define (map func lst)
  (if (null? lst) '()
      (cons (func (car lst)) (map func (cdr lst)))))
```

リストの要素のそれぞれに関数を施す．この関数の利用方法は以下のとおりである．

(map 関数 リスト)

```
(let ((lst '(1 2 3 4 5 6 7)))
  (display lst)
  (newline)
  (display (mymap (lambda (x) (* x x)) lst))
  (newline)
  (display (mymap (lambda (x) (cons x x)) lst))
  (newline))
```

```
(1 2 3 4 5 6 7)
(1 4 9 16 25 36 49)
((1 . 1) (2 . 2) (3 . 3) (4 . 4)
(5 . 5) (6 . 6) (7 . 7))
```

filter関数を定義する

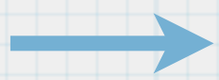
与えられた，条件をチェックする関数とリストが与えられたとき，その条件に合う要素で構成されたリストを作る関数filterを以下のように定義する.

```
(define (filter cnd lst)
  (if (null? lst) '()
      (if (cnd (car lst))
          (cons (car lst) (filter cnd (cdr lst)))
          (filter cnd (cdr lst)))))
```

または,

```
(define (filter cnd lst)
  (cond ((null? lst) '())
        ((cnd (car lst))
         (cons (car lst) (filter cnd (cdr lst))))
        (else
         (filter cnd (cdr lst)))))
```

```
(display (filter (lambda (x) (> x 2))
                 '(6 2 3 1 9 8 7)))
(newline)
```



```
(6 3 9 8 7)
```

mapとfilterで順列を作る (1)

順列を生成する関数permはリストで与えられた要素の並べ方を表現するリストを集めたリストを返す. 計算の考え方は以下のとおり.

(perm ‘(1 2 3)) = 1 を先頭にして (perm ‘(2 3))の要素をつなげたリスト
2 を先頭にして (perm ‘(1 3))の要素をつなげたリスト
3 を先頭にして (perm ‘(1 2))の要素をつなげたリスト
の3つリストをつなげたもの

(1 2 3) と 1 から (2 3) を作るには,

```
(filter (lambda (m) (not (eq? 1 m))) ‘(1 2 3))
```

を実行する.

mapとfilterで順列を作る (2)

プログラムはmapとfilterを用いて以下のように定義することができる.

```
(define (perm lst)
  (if (null? lst) '(())
      (append-all
       (map (lambda (x)
              (let ((y (perm (filter (lambda (m)
                                       (not (eq? x m))) lst))))
                (map (lambda (m) (cons x m)) y))) lst))))
```

ただし, append-all は与えられたリストのリストの中身をappendして一本のリストにする関数で以下のように定義する.

```
(define (append-all lst)
  (if (null? lst) '()
      (append (car lst) (append-all (cdr lst)))))
```

mapとfilterで順列を作る (3)

permを実行すると以下のようになる.

```
#|kawa:3|# (perm '(a b c d))
((a b c d) (a b d c) (a c b d) (a c d b) (a d b c) (a d c b) (b a c d)
 (b a d c) (b c a d) (b c d a) (b d a c) (b d c a) (c a b d) (c a d b)
 (c b a d) (c b d a) (c d a b) (c d b a) (d a b c) (d a c b) (d b a c)
 (d b c a) (d c a b) (d c b a))
#|kawa:4|#
```

個数不定引数の関数の定義 (1)

あらかじめ定義されているいくつかの関数は引数が個数が不定である．たとえば，足し算を計算する関数 + はいつくでも引数を取ることができる．

```
(+) -> 0
(+ 1) -> 1
(+ 1 2) -> 3
(+ 1 2 3) -> 6
   .......
```

このような関数を書くには，defineの引数の記述を変えれば良い．

(define (関数名 引数1 ... 引数k . 残りの引数列)
式1 式2 ... 式m)

このように書くとk個の引数は必須で，その他の引数はリストとして一つの変数に結ばれる

個数不定引数の関数の定義 (2)

複数のものを一度に出力するdisplay関数をmdisplayとして以下のように定義する. 繰り返しで定義する必要はなく, mapでまとめて実行させれば良い.

```
(define NL 'new-line)

(define (mdisplay . args)
  (map (lambda (x)
         (if (eq? x NL) (newline)
             (display x)))
       args))
```

```
(mdisplay "Hello, " "everybody!" NL "m = " 23 NL)
```

```
Hello, everybody!
m = 23
```

関数に個数不定のデータを与えて計算させる

変数の個数が不定の関数が書ける一方で，関数を呼び出すときは引数の個数がプログラミングの時点で分かっている必要があるとすると不便である．そこで関数applyが用意されている．実はScheme処理系内部ではapplyがまず実現されている．それを使って式評価 (eval)が行われる

`(apply 関数 引数のリスト)`

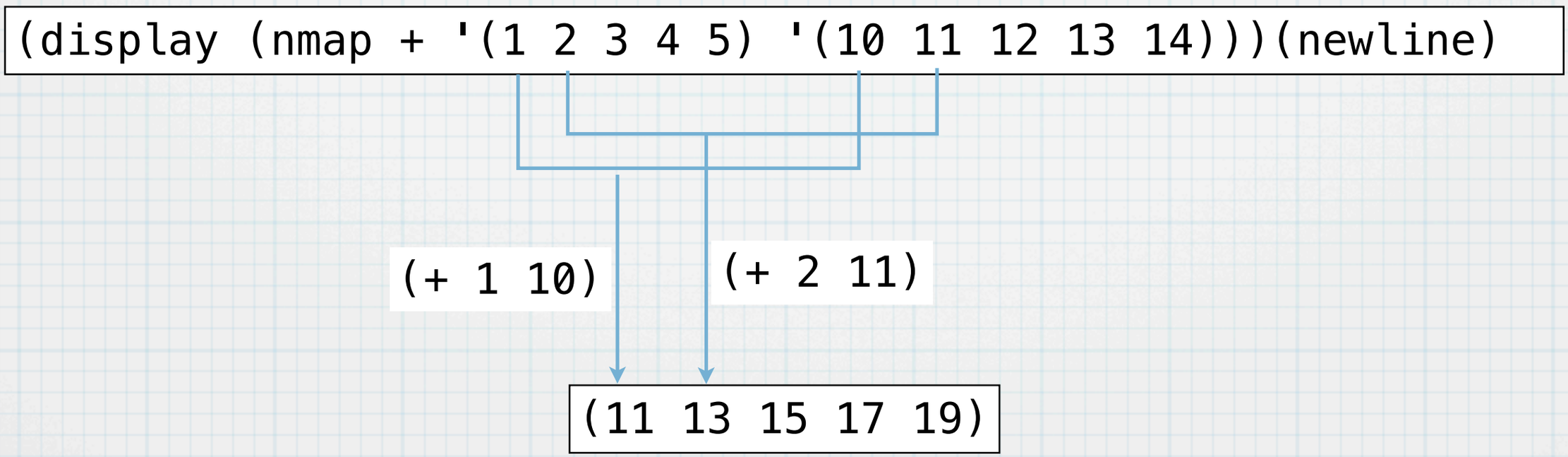
```
(display (apply + '(1 2 3 4 5))) (newline)

(mdisplay '("Hello" 23 545 NL 45 NL)) (newline)
(apply mdisplay (list "Hello" 23 545 NL 45 NL))
```

```
15
(Hello 23 545 NL 45 NL)
Hello23545
45
```

mapの拡張 (1)

mapはある１つのリストのそれぞれの要素を与えられた関数に与えて評価したものを集めるという関数であるが，複数のリストについて，それぞれのリストから１つずつ要素を取り出して，与えられた関数に入力として入れて，その結果を集めることが可能になると応用範囲が広がる．ここでは元のmapと区別するためのnmapという関数名を用いる．



mapの拡張 (2)

nmapは以下のように定義することができる.

```
(define (nmap func . lsts)
  (if (null? (car lsts)) '()
      (let ((ccc (map (lambda (d) (car d)) lsts))
            (ddd (map (lambda (d) (cdr d)) lsts)))
  (let ((res (apply func ccc)))
          (cons res (apply nmap (cons func ddd)))))))
```

関数 range とそれによる九九の表の出力 (1)

ある範囲の整数について適当な作業を繰り返す場合，Cなどではfor文などの繰返し構造を使えば良い．Schemeでもfor文に対応する繰返し構造を作って利用すれば同様にコーディングすることができる．しかし，リストとmapが使えるのであれば，0からn-1までの整数のリストを作って，それについてmapを実行することで同様のことを実行できる．そのために，0からn-1までの整数を生成する関数rangeを以下のように定義する．

```
(define (range n)
  (define (range0 i)
    (if (= i n) '()
        (cons i (range0 (+ i 1)))))
  (range0 0))
```

```
(display (range 100))(newline)
```

```
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99)
```

関数 range とそれによる九九の表の出力 (2)

九九の表を出力する 関数 range-kuku は以下のように定義できる.

```
(define (range-kuku n)
  (let ((r (range n)))
    (map (lambda (i)
           (map (lambda (j)
                  (display (i2s (* i j)))) r)
           (newline)) r)))
```

```
(define (i2s i)
  (string-append "   " (if (< i 10) " " "") i))

(range-kuku 10)
```

```
   0   0   0   0   0   0   0   0   0   0
   0   1   2   3   4   5   6   7   8   9
   0   2   4   6   8  10  12  14  16  18
   0   3   6   9  12  15  18  21  24  27
   0   4   8  12  16  20  24  28  32  36
   0   5  10  15  20  25  30  35  40  45
   0   6  12  18  24  30  36  42  48  54
   0   7  14  21  28  35  42  49  56  63
   0   8  16  24  32  40  48  56  64  72
   0   9  18  27  36  45  54  63  72  81
```

畳み込み：foldr (1)

演算$\circ$を用いて，リスト $(a_1\ a_2\ \ldots\ a_n)$ と g について

$$(a_1 \circ (a_2 \circ (\cdots \circ (a_n \circ g) \cdots)))$$

を計算することを考える．このような計算を行う関数としてfoldrがある．この関数は，以下のように定義することができる．

```
(define (foldr op g lst)
    (if (null? lst) g
        (op (car lst)(foldr op g (cdr lst)))))
```

foldrを用いると以下のように計算することができる．

```
(display (foldr + 0 '(1 2 3 4 5)))
(newline)
```

→ 15

```
(display (foldr * 1 '(1 2 3 4 5)))
(newline)
```

→ 120

畳み込み：foldr (2)

foldrを用いると意外なものが計算できる．まず，以下のように定義される length2 によってリストの長さを測ることができる．

```
(define (length2 lst) (foldr (lambda (a x) (+ x 1)) 0 lst))
```

さらに，以下のように定義されるreverse2によってリストを反転することができる．

```
(define (reverse2 lst) (foldr (lambda (x y)
                                (append y (list x))) '() lst))
```

```
(define (length2 lst) (foldr (lambda (a x) (+ x 1)) 0 lst))
(display (length2 '(1 2 3 4 5 6)))
(newline)

(define (reverse2 lst) (foldr (lambda (x y)
                                (append y (list x))) '() lst))
(display (reverse2 '(1 2 3 4 5 6)))
(newline)
```

```
6
(6 5 4 3 2 1)
```

畳み込み：foldl (1)

foldrと同様にfoldlを考えることができる．こちらは，式で書くと以下のような計算を行うことになる．この場合，繰り返しのプログラミングになる．

$$(((g \circ a_1) \circ a_2) \cdots \circ a_n)$$

```
(define (foldl op g lst)
  (if (null? lst) g
      (foldl op (op g (car lst)) (cdr lst)))))
```

この場合，foldrのようにappendを使わなくても自然にconsのみでreverseを定義することができる．

```
(define (reversex lst)
  (foldl (lambda (x y) (cons y x)) '() lst))
(display (reversex '(a b c d e f)))
(newline)
```

→

```
(f e d c b a)
```

合成関数 (1)

２つ以上の関数を合成して新たな関数を作ることができる．たとえば，２つの関数 f, g を合成して，$f \circ g$ を作ることができる．これは以下のように解釈する：

$$(f \circ g)(x) = f(g(x))$$

このような操作をする場合，関数を返す関数を定義する必要がある．

合成関数を作る関数compは以下のように定義する．また，否定notと偶数の判定を行うeven?を組み合わせると奇数判定の関数odd?を作ることができる．

```
(define (comp f g) (lambda (x) (f (g x))))
```

```
(define odd? (comp not even?))
(let ((lst '(2 3 4 5 6)))
  (display lst)
  (newline)
  (display (map odd? lst))
  (newline))
```

→

```
(2 3 4 5 6)
(#f #t #f #t #f)
```

合成関数 (2)

同様にして，この合成関数を作る関数compとfoldrを用いて，関数の合成を多重に行うことができる.

```
(define (id x) x)
(define (compose lst) (foldr comp id lst))
```

Newton法を実行するために (f(x) = x^2 - 1とする），以下のような関数を定義する：

$$F(x) = x - \frac{f(x)}{f'(x)}$$

```
(define (F x)
  (let ((f (lambda (x) (- (* x x) 2.0)))
        (fd (lambda (x) (* 2 x))))
    (- x (/ (f x) (fd x)))))
```

これにより，以下のように√2を求まる.

```
(define foo (compose (list F F F F F F F)))
(display (foo 1.0))
(newline)
```

→

```
1.4142135623730951
```

lambdaを用いてデータを表現する

lambdaを用いてconsセルと同じ機能をもつものを作ることができる.

```
(define (mycons x y)
  (lambda (m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else '()))))

(define (mycar cell) (cell 'car))
(define (mycdr cell) (cell 'cdr))
```

```
(let ((m (mycons 'a 'b)))
  (display (mycar m))(newline)
  (display (mycdr m))(newline))
```

```
a
b
```

この場合，myconsの値は関数ということになるが，直接関数として使うのではなくて，それをwrapする関数を用意する.

lambdaを用いて配列を作る (1)

前のスライドで定義したmyconsを修正してlambdaのみで配列を作ってみる.

```
(define (mycons x y)
  (lambda (m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (set! x m)))))

(define (mycar cell) (cell 'car))
(define (mycdr cell) (cell 'cdr))
(define (myset-car! cell x) (cell x))

(define   (make-array n)
  (if (zero? n) 'last
      (mycons (make-array (- n 1)))))

(define   (array-ref a i)
  (if (zero? i) (mycar a)
      (array-ref (mycdr a) (- i 1))))
(define   (array-set a i v)
  (if (zero? i) (myset-car! a v)
      (array-set (mycdr a) (- i 1) v)))
```

値を変更できるようにするための拡張

lambdaを用いて配列を作る (2)

定番のエラトステネスのふるいを書いてみる（後述の名前付きletを用いている）.

```
(load "myarray.scm")

(define (seive n)
  (let ((a (make-myarray n)))
    (let loop1 ((i 2))
      (when (< i n)
            (myarray-set a i #t)
            (loop1 (+ i 1))))
    (let loop2 ((i 2))
      (when (< i n)
            (when (myarray-ref a i)
                  (display i)
                  (display " ")
                  (let loop3 ((j (* i 2)))
                    (when (< j n)
                          (myarray-set a j #f)
                          (loop3 (+ j i)))))
            (loop2 (+ i 1)))))
  (newline))
(seive 1000)
```

```
OMacBook:yama518> kawa seive.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
```

繰り返しのためのlet: 名前付きlet (1)

繰り返しを記述する場合，いちいち内部に関数を書くのは面倒である．そのため，letに名前を付けて，そのletを再度let内部から呼び出すことができる．これによって，繰り返しを記述することができる．

```
(let loop ((i 0))
  (when (< i 10)
        (display i)
        (display " ")
        (loop (+ i 1))))
(newline)
```

```
(let ()
  (define (loop i)
    (when (< i 10)
          (display i)
          (display " ")
          (loop (+ i 1))))
  (loop 0))
(newline)
```

左のプログラムでは，loopは外で定義された関数ではなく，内部で定義された局所関数であり，右のプログラムと等価である．

繰り返しのためのlet: 名前付きlet (2)

名前付きletは普通末尾再帰（繰り返し）を表現するのに用いるが，末尾再帰でなくても定義できる.

```
(display
 (let loop ((i 10))
   (if (zero? i) 0
       (+ i (loop (- i 1))))))
(newline)

(display
 (let fib ((n 38))
   (if (<= n 1) 1
       (+ (fib (- n 1)) (fib (- n 2))))))
(newline)
```

```
OMacBook:yama531> kawa let-sample2.scm
55
63245986
```

繰り返しのためのlet: 名前付きlet (3)

myforなどを定義しなくてもkuku関数も以下のように自然にかける.

```
(define (i2s i)
  (string-append
   "  "
   (if (< i 100) " " "")
   (if (< i 10) " " "")
   (number->string i)))

(define (kuku n)
  (let loop1 ((i 1))
    (when (< i n)
          (let loop2 ((j 1))
            (when (< j n)
                  (display (i2s (* i j)))
                  (loop2 (+ j 1))))
          (newline)
          (loop1 (+ i 1)))))

(kuku 10)
```

```
OMacBook:yama534> kawa kuku-let.scm
    1    2    3    4    5    6    7    8    9
    2    4    6    8   10   12   14   16   18
    3    6    9   12   15   18   21   24   27
    4    8   12   16   20   24   28   32   36
    5   10   15   20   25   30   35   40   45
    6   12   18   24   30   36   42   48   54
    7   14   21   28   35   42   49   56   63
    8   16   24   32   40   48   56   64   72
    9   18   27   36   45   54   63   72   81
OMacBook:yama535>
```