プログラミング言語論第2回

手続きとそれが生成するプロセス

情報工学科 山本修身

前回の復習 (1)

S式によるデータ（プログラム）の表現

```
(1 2 3 (5 4 3 . 8) 2)

(abc (1 2) . (def zef))
```

数の四則演算

```
(+ 1 2 3 4 5) = 1 + 2 + 3 + 4 + 5 = 15

(- 20 3 2 1) = 20 - 3 - 2 - 1 = 14

(* 2 3 4 5 6) = 2 * 3 * 4 * 5 * 6 = 720

(/ 2 3 4) = 2 / 3 / 4 = 1/6

(+ (* 2 3) (* 4 5)) = 2 * 3 + 4 * 5 = 26
```

Schemeでは出来る限り正確な演算を行う．多くの場合，有理数の割り算などでは分数をデータとして扱うことができる.

前回の復習(2)

変数の定義

```
(define a 20)

(define b “abcde”)
```

関数の定義

```
(define (sqr x) (* x x))

(define (関数名 引数1 引数2 ... 引数n) 式1 式2 ... 式m)
```

本体

データの表示

```
(display “abc”)

(display (+ 1 2 3))

(newline)
```

前回の復習 (3)

定義されている変数の内容を変える

```
(set! a 34)

(set! b “meijo”)
```

成り立つ条件によって式の値を変える

```
(if (< a 0) (- a) a)

(if 条件式 式1 式2)

(when (< m 1) (set! m (+ 1 2)) (* m m))

(when 条件式 式1 式2 .. 式n)
```

whenは条件が成り立たないとき何も値を返さない.

前回の復習 (4)

1からnまでの和を求めるプログラム

$$\mathrm{sum}(n) = 1 + 2 + \cdots + n$$
$$\overline{\mathrm{sum}}(i, n, r) = \overline{\mathrm{sum}}(i + 1, n, r + i)$$
$$\mathrm{sum}(n) = \overline{\mathrm{sum}}(1, n, 0)$$

```
(define (sumx i n res)
  (if (> i n) res
      (sumx (+ i 1) n (+ res i))))
(define (sum n) (sumx 1 n 0))
(display (sum 100))
(newline)
```

```
OMacBook:yama508> kawa sum.scm
5050
```

```
(sum 5) = (sumx 1 5 0) = (sumx 2 5 1) = (sumx 3 5 3)
= (sumx 4 5 6) = (sumx 5 5 10) = (sumx 6 5 15) = 15
```

前回の復習 (5)

以下のようなプログラムでも同じ計算をすることができる．ただし，足す順番が異なる．

$$\mathrm{sum}(n) = 1 + 2 + \cdots + n$$
$$\mathrm{sum}(n) = \mathrm{sum}(n-1) + n$$

```
(define (sum n)
  (if (= n 0) 0
      (+ (sum (- n 1)) n)))
(display (sum 100))
(newline)
```

このプログラムは前述のものと同じ値を返すが，計算の仕方が異なる．あとで詳しく説明するが，Schemeのプログラムとしてはこのプログラムは前者と決定的に異なる．

```
(sum 5) = (+ (sum 4) 5) = (+ (+ (sum 3) 4) 5)
= (+ (+ (+ (sum 2) 3) 4) 5) = (+ (+ (+ (+ (sum 1) 2) 3) 4) 5)
= (+ (+ (+ (+ (+ (sum 0) 1) 2) 3) 4) 5)
= (+ (+ (+ (+ (+ 0 1) 2) 3) 4) 5) = 15
```

素数を順番に出力するプログラム (1)

素数とは，整数のうちで1およびその数自身以外で割り切れない数のことである．素数を求めるプログラムを考える．

```
(define (prime?x i n)
  (if (> (* i i) n) #t
      (if (= (remainder n i) 0) #f
          (prime?x (+ i 1) n))))
(define (prime? n) (prime?x 2 n))
```

真

偽

```
#|kawa:4|# (prime? 10009)
#t
#|kawa:5|# (prime? 100009)
#f
#|kawa:6|# (prime? 2009)
#f
#|kawa:7|# (prime? 20009)
#f
#|kawa:8|# (prime? 200009)
#t
```

(remainder a b)は整数aを整数bで割った余りを返す関数である.

整数nが素数であることを確認するには,
2〜floor(√n) の整数で割ってみれば良い.
割り切れれば合成数である．すべてで割り切れなければ素数である

いくつかの命令を順に実行させる特殊形式

細かい命令をいくつか順に実行させるという一つの命令を作るための構造がある．このようなことは関数を新たに定義すれば可能であるが，いちいち関数を定義するのは不便である．

```
(begin 式1 式2 ... 式n)
```

この式を評価すると，式1, 式2, ... 式nを順に評価して最後の式nの評価値をその式の評価値とする．

```
#|kawa:1|# (begin
#|(---:2|#    (define n 20)
#|(---:3|#    (set! n (* n n))
#|(---:4|#    (display n)
#|(---:5|#    (newline)
#|(---:6|#    (+ n 1))
400
401 ← この式の値
#|kawa:7|#
```

素数を順番に出力するプログラム (2)

i から n までの素数をプリントアウトするプログラム.

```
(define (print-primes i n)
  (if (> i n) #t
      (begin
        (when (prime? i) (display i) (display " "))
        (print-primes (+ i 1) n))))

(print-primes 2 1000)
(newline)
```

```
OMacBook:yama520> kawa prime.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
OMacBook:yama521>
```

九九の表を書くプログラム (1)

まずは，整数を右づめで表示するためのプログラムを作る.

(string-append s1 s2)は文字列s1とs2を連結した文字列を返す関数である

(string-length s) は文字列sの長さを返す

```
(define (spaces n)
  (if (= n 0) ""
      (string-append (spaces (- n 1)) " ")))
```

```
(define (int->string i n)
  (define s (number->string i))
  (string-append (spaces (- n (string-length s)))
                 s))
```

```
(define (check n)
  (display "|")
  (display (int->string 23 n))
  (display "|")
  (newline))

(check 3)(check 4)(check 5)(check 6)
(check 7)(check 8)
```

```
| 23|
|  23|
|   23|
|    23|
|     23|
|      23|
|       23|
```

九九の表を書くプログラム (2)

iからnまでの数にkを掛けた数をそれぞれ4桁で並べた文字列を返す関数 (kuku-i i n res)を以下のように定義する. これにより九九の表の一行分を作ることができる.

```
(define (kuku-i i n k res)
  (if (> i n) res
      (kuku-i (+ i 1) n k
              (string-append res (int->string (* i k) 4)))))
```

```
(display (kuku-i 1 9 3 ""))
(newline)
```

```
   3   6   9  12  15  18  21  24  27
```

九九の表を書くプログラム (3)

kuku-i を用いて九九の表を出力する関数 (kuku n)を定義する.

```
(define (kukux j n)
  (if (> j n) #t
      (begin
        (display (kuku-i 1 n j ""))
        (newline)
        (kukux (+ j 1) n))))
(define (kuku n) (kukux 1 n))
```

(kuku 9) →

```
 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
```

関数とはなにか？

関数は値が与えられたとき一定の計算を行う能力をもった実体（データ）である．Schemeでは（`define（foo x）...`）のようにして定義する．このように関数を定義できるが，もっと直接的に関数を作ることができる．そのための特殊形式が`lambda`である．lambdaを使うと関数が値を計算するのと同じように構成できる．

```
(lambda（引数の並び）式1 式2 ... 式n)
```


```
#|kawa:24|# (lambda (n) (+ n 1))
#<procedure gnu.expr.ModuleMethod>
#|kawa:25|# ((lambda (n) (+ n 1)) 44)
45
#|kawa:26|#
```

関数を生成する特殊形式

生成された関数

この関数を改めて生成して44を引数にして実行する

defineとはなにか？

defineは変数と値を結びつけるための特殊形式である．defineを関数定義に用いる場合，それは，lambdaで生成された関数を値として変数に結びつけているにすぎない．

```
(define (foo n) (* n n))
```

等価

```
(define foo (lambda (n) (* n n)))
```

関数を生成する

ここからわかるように，Schemeでは関数はどこでも，いつでも作ることができる．

defineで関数を定義することの意味

普通にdefineを用いて関数を定義しているが，これはlambdaを使った関数の生成と変数への結びつけを同時に行っていることにすぎない．すなわち，

```
(define (関数名 引数1 引数2 ... 引数n)
          式1 式2 ... 式n)
```

は以下の式と全く同じである．変数を定義して値を代入している．

```
(define 関数名
    (lambda (引数1 引数2 ... 引数n)
              式1 式2 ... 式n)))
```

ニュートン法 (1)

ニュートン法は，関数の値と微分係数を用いて関数の零点を求めるアルゴリズムである．f(x) = 0 を求めるために，以下の反復を行う．

$$x' \leftarrow x - \frac{f(x)}{f'(x)}$$

かならず収束する保証はないが，xが十分に真の零点に近く，f(x)がxの近傍で十分に直線として近似できれば収束する．Schemeによるプログラムは非常に簡単で以下のようになる．

(abs x)はxの絶対値を返す関数である．

```
(define (newton x f fd eps)
  (if (< (abs (f x)) eps) x
      (newton (- x (/ (f x) (fd x))) f fd eps)))
```

ただし，f は零点を求める関数であり，fdはfの導関数であるとする．それぞれSchemeの関数として与えられるとする．

ニュートン法 (2)

関数newtonは以下のようにして使うと便利である．たとえば，√2を計算する場合，以下のようにする．

```
(display
 (newton 1.0
         (lambda (x) (- (* x x) 2.0))
         (lambda (x) (* 2 x))
         1.0e-10))
(newline)
```

```
1.4142135623746899
```

下線部はその場で直接関数を生成している．このようにして生成された関数は名前を持たない関数（無名関数）であり，このような関数のことを**クロージャー（closure: 閉包）**と呼ぶ．関数そのものをデータとして扱えることにより，色々便利なことが出来るようになる．

ニュートン法 (3)

lambdaを使わなければ，以下のような書き方になる.

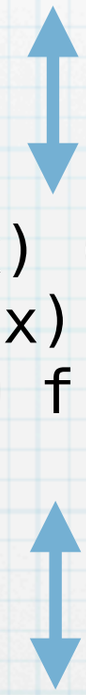
```
(define (f x) (- (* x x) 2.0))
(define (fd x) (* 2 x))
(display (newton 1.0 f fd 1.0e-10))
(newline)
```

この場合，関数fやfdを直接定義することになるが，lambdaを使えば，名前をつけずにその場で生成して渡すことができる.

ニュートン法 (4)

結局，「ニュートン法 (2), (3)」で示したプログラムは同じことの書き換えであることがわかり，lambdaでそのまま書くのが一番単純である.

```
(define (f x) (- (* x x) 2.0))
(define (fd x) (* 2 x))
(display (newton 1.0 f fd 1.0e-10))
(newline)
```



```
(define f (lambda (x) (- (* x x) 2.0)))
(define fd (lambda (x) (* 2 x)))
(display (newton 1.0 f fd 1.0e-10))
(newline)
```



```
(display (newton 1.0
                 (lambda (x) (- (* x x) 2.0))
                 (lambda (x) (* 2 x))
                 1.0e-10))
(newline)
```

多倍長計算で√2の近似値を高精度に求める (1)

kawaでは桁の大きな整数を扱うことができ，この性質を用いて√2の高精度な近似値を計算してみる．アルゴリズムはNewton法を用いる．

高精度な数を表すには，大きな整数 i を 暗黙の了解である数 b で割ったもの i / b と考える．bとしてたとえば，b = 10^{100} とすれば，10進数で小数点以下100桁の数を表現することができる．

和 $\frac{i}{b} + \frac{j}{b} = \frac{i+j}{b}$

積 $\frac{i}{b} \cdot \frac{j}{b} = \frac{ij}{b^2} = \frac{ij/b}{b}$

差 $\frac{i}{b} - \frac{j}{b} = \frac{i-j}{b}$

商 $\frac{i}{b} / \frac{j}{b} = i/j = \frac{ib/j}{b}$

実際には分子のみを記録してその数と考える

多倍長計算で√2の近似値を高精度に求める (2)

プログラムとしては以下のようになる.

```
(define (ruijo a n)
  (if (<= n 0) 1
      (* (ruijo a (- n 1)) a)))

(define base (ruijo 10 1000))
(define (tabai-add a b) (+ a b))
(define (tabai-sub a b) (- a b))
(define (tabai-mult a b) (quotient (* a b) base))
(define (tabai-div a b) (quotient (* a base) b))

(define one base)
(define seven (* base 7))
(display (tabai-div one seven))
(newline)
```

7分の1を計算してみる

```
OMacBook:yama528> kawa tabai.scm
14285714285714285714285714285714285714285
71428571428571428571428571428571428571428
5714285714285714285
```

多倍長計算で√2の近似値を高精度に求める (3)

ここで定義した演算を使ってNewton法のプログラムを書く.

```
(define two (* 2 base))
```

定数２の定義

```
(define (work)
  (define + tabai-add)
  (define - tabai-sub)
  (define * tabai-mult)
  (define / tabai-div)
```

それぞれの演算をtabai-*に置き換える

```
  (define (newton x f fd eps)
    (if (< (abs (f x)) eps) x
        (newton (- x (/ (f x) (fd x))) f fd eps)))
```

以前定義したNewton法の定義

```
  (newton one
          (lambda (x) (- (* x x) two))
          (lambda (x) (* two x))
          10000))

(display (work))
(newline)
```

Newton法の実行
epsは10000/base

```
OMacBook:yama542> kawa tabai2.scm
14142135623730950488016887242096980785696718753769480731766797379907324784621070388503875343276416017
```

多倍長計算で√2の近似値を高精度に求める (4)

base = 10^{1000} とすれば，形式上1000桁の精度で求まる（実際には丸め誤差があるので1000桁付近に誤差がある）．

```
(define m (number->string (work)))
(display (substring m 0 1))
(define (output-string s i)
  (if (= (string-length s) 0) #t
      (begin
        (if (= (remainder i 5) 0) (newline))
        (define ll (min 10 (string-length s)))
        (display (substring s 0 ll))
        (display " ")
        (output-string (substring s ll (string-length s)) (+ i 1)))))

(output-string (substring m 1 (string-length m)) 0)
(newline)
```

多倍長計算で√2の近似値を高精度に求める (5)

前のページのプログラムを実行すると，以下のように出力される．ただし，base = 10^{1000} とした．

```
1
4142135623 7309504880 1688724209 6980785696 7187537694
8073176679 7379907324 7846210703 8850387534 3276415727
3501384623 0912297024 9248360558 5073721264 4121497099
9358314132 2266592750 5592755799 9505011527 8206057147
0109559971 6059702745 3459686201 4728517418 6408891986
0955232923 0484308714 3214508397 6260362799 5251407989
6872533965 4633180882 9640620615 2583523950 5474575028
7759961729 8355752203 3753185701 1354374603 4084988471
6038689997 0699004815 0305440277 9031645424 7823068492
9369186215 8057846311 1596668713 0130156185 6898723723
5288509264 8612494977 1542183342 0428568606 0146824720
7714358548 7415565706 9677653720 2264854470 1585880162
0758474922 6572260020 8558446652 1458398893 9443709265
9180031138 8246468157 0826301005 9485870400 3186480342
1948972782 9064104507 2636881313 7398552561 1732204024
5091227700 2269411275 7362728049 5738108967 5040183698
6836845072 5799364729 0607629969 4138047565 4823728997
1803268024 7442062926 9124859052 1810044598 4215059112
0249441341 7285314781 0580360337 1077309182 8693147101
7111168391 6581726889 4197587165 8215212822 9518488473
```

√2が1000桁求まる．ただし，最下位の1,2桁は誤差を含む可能性があるので，すべてが完全に正しいというわけではない．

繰り返すための構造を考える（1）

C言語やJavaにはforやwhileのように動作を繰り返す構造があらかじめ用意されている．Schemeの場合には，実はdoという構造があるが，doを使わなくても繰返し動作を記述することが簡単にできる．

```
(define (myfor i n func)
  (when (< i n)
        (func i)
        (myfor (+ i 1) n func)))

(myfor 0 10 (lambda (i)
              (display i)
              (newline)))
```

```
osami-2:yama503> kawa myfor.scm
0
1
2
3
4
5
6
7
8
9
```

繰り返すための構造を考える（2）

```
(define (myfor i n func)
  (when (< i n)
        (func i)
        (myfor (+ i 1) n func)))

(define (spaces n)
  (if (= n 0) ""
      (string-append (spaces (- n 1)) " ")))

(define (int->string i n)
  (define s (number->string i))
  (string-append (spaces (- n (string-length s)))
                 s))

(define (work n)
  (myfor 0 n
         (lambda (i)
           (myfor 0 n
                  (lambda (j)
                    (display (int->string (* i j) 5))))
           (newline))))
(work 10)
```

```
osami-2:yama504> kawa myfor2.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
```

iに関する繰返し

jに関する繰返し

九九の表を出力するのであれば左のように書けば良い.

練習問題

while文に対応するmywhile関数をつくってみよ．mywhileの仕様はつぎのとおり

> (mywhile 条件を調べる関数 繰返し内部の動作)
> 「条件を調べる関数」の結果が真であるかぎり「繰返し内部の動作」を実行し続ける

実際に利用する場合には以下のようにする．

```
(define i 2)
(mywhile (lambda () (< i 10))
         (lambda ()
           (display i)
           (newline)
           (set! i (+ i 1))))
```

フィボナッチ数列（1）

以下のように定義される関数を**フィボナッチ数列 (Fibonacci sequence)**と呼ぶ.

$$F(n+2) = F(n+1) + F(n)$$
$$F(1) = F(0) = 1$$

数列の下から計算するのであれば以下のようにすれば良い.

```
(define a 1)
(define b 1)
(myfor 2 30
       (lambda (i)
         (define c (+ a b))
         (display i)
         (display " ")
         (display c)
         (newline)
         (set! a b)
         (set! b c)))
```

```
osami-2:yama508> kawa fibx.scm
2 2
3 3
4 5
5 8
6 13
7 21
8 34
9 55
10 89
11 144
12 233
13 377
14 610
15 987
16 1597
17 2584
18 4181
19 6765
20 10946
21 17711
22 28657
23 46368
24 75025
25 121393
26 196418
27 317811
28 514229
29 832040
```

フィボナッチ数列（2）

フィボナッチ数列は再帰的に定義して計算することができる.

```
(define (fib n)
  (if (<= n 1) 1
      (+ (fib (- n 1)) (fib (- n 2)))))

(myfor 2 30
       (lambda (i)
         (display i)
         (display " ")
         (display (fib i))
         (newline)))
```

$$F(n+2) = F(n+1) + F(n)$$

$$F(1) = F(0) = 1$$

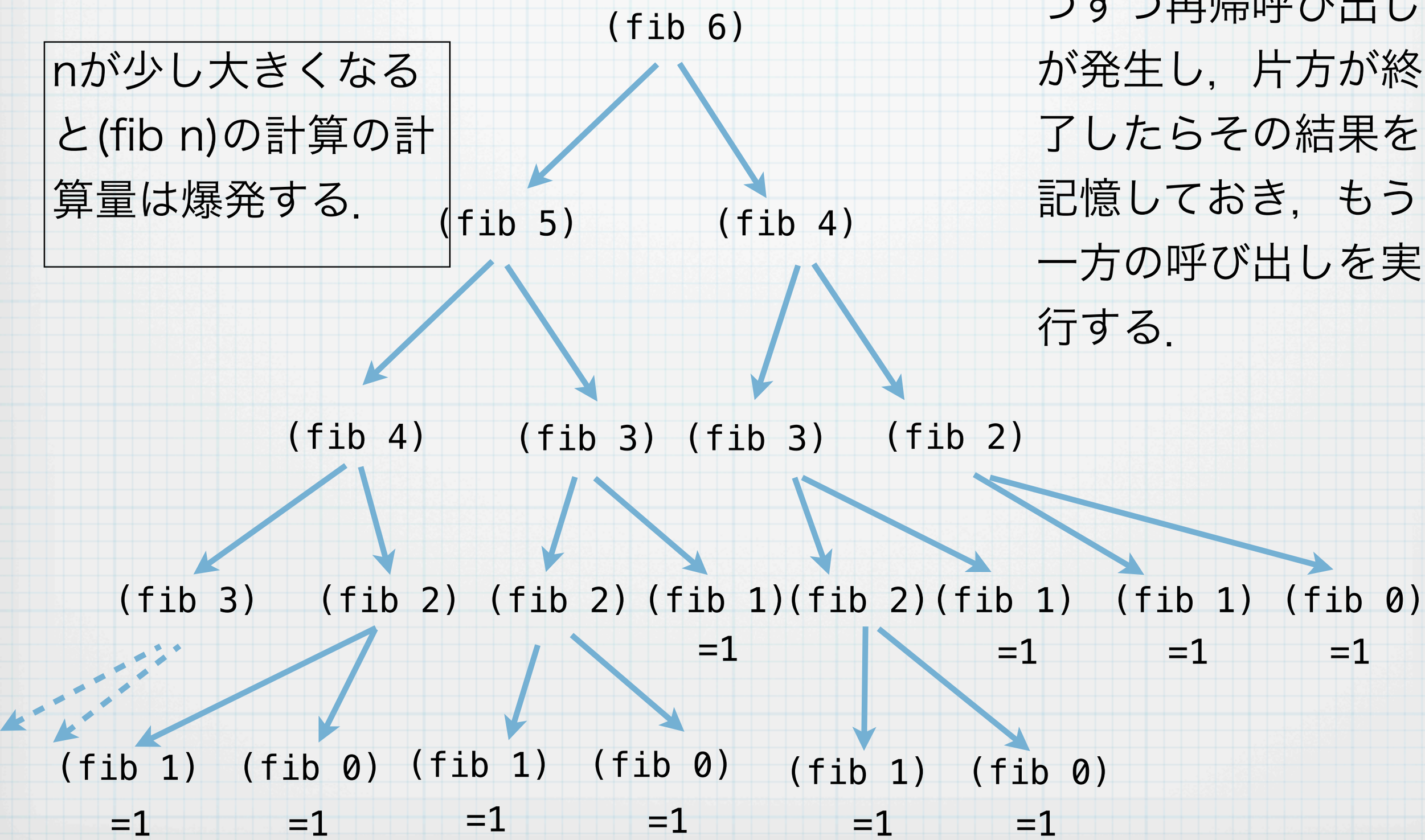
```
osami-2:yama511> kawa fib.scm
2 2
3 3
4 5
5 8
6 13
7 21
8 34
9 55
10 89
11 144
12 233
13 377
14 610
15 987
16 1597
17 2584
18 4181
19 6765
20 10946
21 17711
22 28657
23 46368
24 75025
25 121393
26 196418
27 317811
28 514229
29 832040
```

```
(define (fib n)
  (if (<= n 1) 1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

フィボナッチ数列（3）

このプログラムは完全な再帰である.

nが少し大きくなると(fib n)の計算の計算量は爆発する.



再帰を行うごとに2つずつ再帰呼び出しが発生し，片方が終了したらその結果を記憶しておき，もう一方の呼び出しを実行する.

フィボナッチ数列（4）

再帰でフィボナッチ数を計算すると，計算量が爆発する.

```
(define t1
(java.lang.System:currentTimeMillis))
(define res (fib 38))
(define t2
(java.lang.System:currentTimeMillis))
(display res)
(newline)
(display (- t2 t1))
(display " msec")
(newline)
```

実際，フィボナッチ数はベンチマークのための計算として利用されている.

```
osami-2:yama514> kawa fib.scm
63245986
4719 msec
```

この関数を再帰で書くかぎり，再帰するたびに枝分かれするので，計算量が爆発する．それでは枝分かれしない再帰はあり得るのか？

→ **それが「繰返しである」**

繰り返すための再帰

同じ再帰呼び出しでもmyforのような再帰について調べてみる.

```
(define (myfor i n func)
  (when (< i n)
        (func i)
        (myfor (+ i 1) n func)))
```

```
(myfor 0 5 func)
        ↓
(myfor 1 5 func)
        ↓
(myfor 2 5 func)
        ↓
(myfor 3 5 func)
        ↓
(myfor 4 5 func)
        ↓
(myfor 5 5 func)
```

myforを再帰的に呼び出したあと呼び出し元に帰ってくる必要がない.

このように最後に再帰するような構造は戻ってくる必要がない. これを**末尾再帰（tail recursion)** と呼ぶ. 末尾再帰の場合には, これを繰返しとして実行することができる. すなわちループ構造と同じ動作になる. このように実行することを**末尾再帰の最適化**と呼び, Schemeなどは常に最適化されて実行される.

繰返しの例（cos関数）

三角関数を繰返しを用いて計算してみる．倍角公式より，

$$\cos x = 2\cos^2 \frac{x}{2} - 1$$

であり，さらに，角度xが小さいとき，テーラー展開より，

$$\cos x \approx 1 - \frac{x^2}{2}$$

以上より，cos x を計算する繰返しのプログラムは以下のとおり．

```
(define (mycos theta)
  (define eps 1.0e-6)
  (define (degree t n)
    (if (< t eps) (mycosx t n (- 1.0 (* t t 0.5)))
        (degree (/ t 2) (+ n 1))))
  (define (mycosx t n res)
    (if (= 0 n) res
        (mycosx (* t 2) (- n 1) (- (* 2 res res) 1.0))))
  (degree theta 0))
```

```
(display (mycos 1.0))
(newline)
(display (cos 1.0))
(newline)
```

```
osami-2:yama505> kawa mycos.scm
0.5403023037797887
0.5403023058681398
```

末尾再帰で定義された2つの関数によって，表現されている．式のとおり定義をプログラム化すると末尾再帰にならない．

今回のまとめ (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))
```

今回のまとめ (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
```