2024 機械情報夏学期
ソフトウェア第二

担当：岡田 慧 k-okada-soft2@jsk.t.u-tokyo.ac.jp

4. ソフトウェアの抽象化 (手続き)

1　手続きの抽象化

1.1　高階手続き (higher-order procedures) による抽象

手続き（関数）は，引数の数値に依存せずに，ある一定の手続きパターンを実現する手法である．例えば，数の三乗を計算する例を考えると

```
>>> 3 * 3 * 3
27
>>> 4 * 4 * 4
64
```

という個別の特定の数の計算を行うこともできるが，

```
def cube(x):
    return x * x * x
```

という手続きを定義することで，乗算や同じ数値を 3 回書くような不便さから解放されるだけでなく，

```
>>> cube(3)
27
>>> cube(4)
64
```

と手続きに名前をつけて，この名前を使って手続きを呼び出せばよくなる．これにより，プログラムで三乗を計算できるだけでなく，三乗の概念を表す能力をもたせたことになると言える．つまり，よくある計算パータンに名前をつけて抽象化し，その抽象された手続きを使って仕事を行う能になる．

ここでは引数（パラメタ）が数値の例を示しているが，同じプログラムパターンを異なる手続きで利用することが求められる．このようなことを実現するには，手続きを引数に取ったり，手続きを返り値とする手続きがあるとよい．

手続きを扱う手続きを高階手続き (higher-order procedures) と呼ぶ．より高いレベルで手続きを抽象化するための重要なテクニックの一つである．

1.2 引数としての手続き

以下の 3 つの手続きを考える．それぞれ a から b までの整数の和を計算するものと，それぞれ a から b までの整数の三乗の和を計算するもの，さらに，級数 $\frac{1}{1\cdot3}+\frac{1}{5\cdot7}+\frac{1}{9\cdot11}+\cdots$ の並びの和[1]を計算するものである．

左右のプログラムは同じ計算の再帰版と反復版である．どちらか理解しやすい方を見てもらえばよい．また，すでに cube 関数は定義されていることを前提にしている．

```
def sum_integers(a, b):
  if a > b:
    return 0
  else:
    return a + sum_integers(a + 1, b)

def sum_cubes(a, b):
  if a > b:
    return 0
  else:
    return cube(a) + sum_cubes(a + 1, b)

def pi_sum(a, b):
  if a > b:
    return 0.0
  else:
    return (1.0 / (a * (a + 2.0))) + \
       pi_sum(a + 4.0, b)
```

```
def sum_integers(a, b):
   total = 0
   while a <= b:
      total, a = total + a, a + 1
   return total

def sum_cubes(a, b):
   total = 0
   while a <= b:
      total, a = total + cube(a), a + 1
   return total

def pi_sum(a, b):
   total = 0
   while a <= b:
      total, a = total + \
             1.0 / (a * (a + 2.0)), \
             a + 4.0
   return total
```

この 3 つの手続きを見る共通のパターンがあることがわかる．手続き名と，加算する項を計算する a の関数と，a の次の値を用意する関数である．それぞれ，<name>, <term>, <next>とすると，それぞれ以下の同じ雛形のスロットを埋めることで手続きを作成することができる．

```
def <name>(a, b):
   if a > b:
      return 0
   else:
      return <term>(a) + \
             <name>(<next>(a), b)
```

```
def <name>(a, b):
    total = 0
    while a <= b:
        total, a = total + \
               <term>(a), \
               <next>(a)
    return total
```

このようなパターンを数学では総和記号を用い級数の総和 (summation of a series) として抽象している．すなわち

$$\sum_{n=a}^{b} f(n) = f(a) + \cdots + f(b)$$

がこのパターンを表している．

これにより数学の世界では特定の和を扱うのではなく，級数とは無関係に総和の一般的結果を導くといった，総和自身の概念を扱うことを可能にしている．

これをプログラムとして実現するには，以下の様に，上の雛形のスロットを仮パラメタとして手続きを取ることができるような定義をすれば良い．

[1] これは $\pi/8$ 収束する

```
def sum(term, a, next, b):
 if a > b:
    return 0
 else:
    return term(a) + \
           sum(term, next(a), next, b)
```

```
def sum(term, a, next, b):
    total = 0
    while a <= b:
        total, a = total + \
                   term(a), next(a)
    return total
```

再帰版でも反復版でもどちらの方法でも sum 関数を定義することができれば，これを使って三乗の和は以下の様に計算できる．

```
def inc(n): return n + 1

def sum_cubes(a, b):
   return sum(cube, a, inc, b)
```

また，整数の和は identity を使って以下の様に定義できる．

```
def identity(x): return x

def sum_integers(a, b):
   return sum(identity, a, inc, b)
```

また，pi_sum も以下のように定義できる．

```
def pi_sum(a, b):
   def pi_term(x): return 1.0 / (x * (x + 2.0))
   def pi_next(x): return x + 4.0
   return sum(pi_term, a, pi_next, b)
```

これで，print (8.0 * pi_sum(1, 1000)) として，π の近似値を計算することができる．

1.2.1 定積分の計算

さらに，sum を使うと異なる概念を作成するための構成部品とすることができる．例えば，a, b の間の関数 f の定積分は数値的には

$$\int_a^b f = \left[f(a + \frac{dx}{2}) + f(a + dx + \frac{dx}{2}) + f(a + 2\,dx + \frac{dx}{2}) + \cdots \right] dx$$

を使って近似値を得ることができる．これを手続きとして表すと

```
def integral(f, a, b, dx):
   def add_dx(x): return x + dx
   return sum(f, a + (dx / 2.0), add_dx, b) * dx
```

となる．

1.2.2 0 と 1 の間の cube の積分の計算

0 と 1 の間の cube の積分を実際に計算するには以下のようにする[2]．再帰深さの制限にかかるため，sys.setrecursionlimit を用いている．

```
import sys
sys.setrecursionlimit(2000)
print (integral(cube, 0.0, 1.0, 0.001))
```

1.3 lambda を使う手続きの構築

1.3.1 無名関数 (anonymous function): ラムダ式

前節では，sum を使うためだけに，pi_term や pi_next などの些細な手続きを定義し手続きとして名前つけている．一方で,「入力に 4 を増やし返す手続き」など，手続きを直接指定する方法がラムダ (lambda) 式である．

ラムダ式を使うことで名前を持たない関数を定義することが出来る．無名関数とも呼ばれる．

ラムダ式は

```
>>> lambda x : x + 4
<function <lambda> at 0x7f1ce43b1e50>
```

と書く．

ラムダ式は変数に代入することができる．以下では f が関数を示し，引数を取ることで，関数呼び出しが可能になっている．

```
>>> f = lambda x : x + 4
>>> f(1)
5
```

また，以下のように直接 operator にラムダ式を, operands に引数を入て呼び出すことも可能である．

```
>>> (lambda x : x + 4)(1)
5
```

1.3.2 lambda を使う手続きの構築

lambda 式を用いると pi_sum, integral については，補助手続きを定義せずに，lambda 式を用いて以下のように書くことができる．

```
def pi_sum(a, b):
   return sum(lambda x: 1.0 / (x * (x + 2.0)), a, lambda x: x + 4.0, b)

def integral(f, a, b, dx):
   return sum(f, a + (dx / 2.0), lambda x: x + dx, b) * dx
```

[2] 0 と 1 の間の cube の積分の正確な値は $\frac{1}{4}$ である

1.4 値として返される手続き (procedures as returned values)

これまで手続きの抽象化の例として，引数に手続きを渡すことでプログラムの表現力を向上させることを示してきた．本節では返り値もまた手続きとなる手続きを作ることで，さらなる表現力を得る方法を示す．

微分は関数 $x \mapsto x^3$ の微分は $x \mapsto 3x^2$ であるというように，関数を関数に変換する手続きと考えることができる．一般に，g が関数で dx が微小なら，g の微分 Dg は, x における値が

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}$$

で与えられる関数である．

すなわち，dx を 0.00001 として，以下の手続きで表すことができる．

```
dx = 0.00001
def deriv(g):
   return lambda x: (g(x + dx) - g(x)) / dx
```

例えば関数 $x \mapsto x^3$ の 5 での微分は $x \mapsto 3 \cdot 5^2 = 75$ だが，以下のように計算することができる．

```
>>> def cueb(x): return x * x * x
>>> deriv(cube)(5)
75.00014999664018
```

また以下のような二次の導関数の計算も可能になっている．

```
>>> deriv(deriv(cube))(5)
30.000251172168642
```

1.5 手続きの抽象化の例

1.5.1 Newton 法による平方根の計算

平方根を求める手続きには Newton 法がある．すなわち，数 x の平方根の値の予測値 y を使って，y と x/y の平均をとることで，更に良い予測値を得るものである．

希望的思考 (wishful thinking) によるプログラミングの考え方に従うと，まず，数 x とその予測値 y から出発する．予測値が目的に十分ならそこで終了とする．そうでなければ，予測値 y を改善し同じ手順を繰り返す，という基本戦略を立てる．

```
def sqrt_iter(guess, x):
    if good_enougth(guess, x):
        return guess
    else:
        return sqrt_iter(improve(guess, x), x)
```

改善された予測値 y は数 x を予測値 y による商との平均値として計算する．

```
def improve(guess, x):
    return average(guess, (x / guess))

def average(x, y):
    return (x + y) / 2.0
```

最後に,「十分に良い」の定義が必要になる．abs，square は前節までに定義してあるはずだが再掲する．

```
def square(x): return x * x
def abs(x): return -x if x < 0 else x

def good_enougth(guess, x):
    return abs(square(guess) - x) < 0.001

def my_sqrt(x):
    return sqrt_iter(1.0, x)
```

プログラムを実行すると以下の様に計算できる．実は Python の math モジュールあらかじめ組み込まれている sqrt を用いた場合との結果を比較すると良い．

```
>>> my_sqrt(10)
3.162277665175675
>>> import math
>>> math.sqrt(10)
3.1622776601683795
```

1.5.2 一般的方法としての手続きとして，不動点探索を用いた平方根の計算

前節では数値演算のパターンを特定の値は依存しないように抽象する機構として合成手続き (compound procedures: 関数を用いて関数を定義すること) を紹介した．高階手続き (higher-order procedures) では，特定の関数に依存しない，計算の一般的方法を示すための方法を示した．

本節では，不動点を探す一般的方法を手続きとして記す方法を示す．

x が $f(x) = x$ を満たすとき，x を関数 f の不動点 (fixed point) と呼ぶ．この値を求めるためには，ある関数 f について，最初の予測値からはじめ $f(x)$ を計算し，その結果に対して更に f を作用させる $f(f(x))$ を計算し,，f の値が変化しなくなる（あるいは十分に小さい変化になる）まで繰り返し作用させる．

この考え方を使うことで，入力として関数と最初の予測値を取り，その関数の不動点の近似値を返す手続き fixed_point を作ることができる．関数を 2 つの連続する値の差が以下の許容量 tolerance より小さくなるまで繰り返し作用させる．

```
tolerance = 0.00001
def close_enough(v1, v2):
    return abs(v1 - v2) < tolerance

def try_proc(f, guess):
    next = f(guess)
    if close_enough(guess, next):
        return next
    else:
        return try_proc(f, next)

def fixed_point(f, first_guess):
    return try_proc(f, first_guess)
```

このプロセスは平方根を求めるプロセスと類似している．平方根の計算は $y^2 = x$ となる y を探すことであることから，関数 $y \mapsto x/y$ の不動点を計算していると考え以下のように記すことができる．

```
def my_sqrt(x):
    return fixed_point(lambda y: x / y, 1.0)
```

しかしこの不動点探索計算は収束しない．最初の予測値を y_1 とすると，次の予測値は $y_2 = x/y_1$ で，その次の予測値は $y_3 = x/y_2 = x/(x/y_1) = y_1$ と無限ループになる．

これを制御するためには，予測値の大きな変化を防ぐことにある．答えは予測値 y と x/y の間にあることから，これらの平均とすることができる．つまり，y の次の予測値は x/y ではなく，$\frac{1}{2}(y + \frac{x}{y})$ とするとよい．すなわち以下のように書くことができる．`average` は前掲してある．

```
def my_sqrt(x):
    return fixed_point(lambda y: average(y, x / y), 1.0)
```

この修正により以下のように平方根の計算を行うことができる．この様に解への逐次近似を平均化する方法を平均緩和法 (average damping) と呼ぶ．

```
>>> my_sqrt(10)
3.16227766017
```

1.5.3 値として返される手続きとして，Newton 変換を用いた平方根の計算

値として返される手続き

前節では $\sqrt{x}$ が関数 $y \mapsto x/y$ の不動点であるとの認識から，不動点探索の手続きを用いた平方根の計算方法を示した．また，収束のために平均緩和の考え方を用いた．平均緩和はそれ自体を有用な一般的な技法と考えることができる．つまり以下の手続きとして定義できる．

```
def average_damp(f):
    return lambda x: average(x, f(x))
```

これは，引数として手続き `f` を取り，その値として，`lambda` で作り出された，ある数 `x` と `(f x)` の平均を返す手続き，を返す手続きである．例えば，`average_damp` を `square` に作用させると，

ある数 x と x * x の平均を返す手続きを返す．すなわち average_damp(square) を評価すると以下のように手続きが返る．

```
>>> average_damp(square)
<function <lambda> at 0x7f7d7c285050>
```

この返された手続きに 10 を作用させると 10 と 100 の平均として 55 が返る．

```
>>> average_damp(square)(10)
55
```

これにより不動点探索を用いた平方根の手続きは以下の様に書くことができる．

```
def my_sqrt(x):
    return fixed_point(average_damp(lambda y: x / y), 1.0)
```

ここでは，不動点の探索，平均緩和，$y \mapsto x/y$ を使って平方根を計算している．この書き方は本節でここまで紹介してきた my_sqrt と同じプロセスを表しているが，抽象度合いが異なっていることに注意しよう．この様に一つの手続きでもその表現は抽象度に応じて様々あるが，どのレベルの部分を選ぶか，また，どの部分が他の応用にも再利用できるようなまとまった存在か，という観点が重要である．

再利用の例として，x の立方根が関数 $y \mapsto x/y^2$ の不動点であることに注意すれば，以下のように記述できる．

```
def my_cube_root(x):
    return fixed_point(average_damp(lambda y: x / square(y)), 1.0)
```

Newton 変換による平方根の計算

Newton 法は $x \mapsto g(x)$ が部分可能であれば，方程式 $g(x) = 0$ の解は以下のニュートン変換

$$f(x) = x - \frac{g(x)}{Dg(x)} (\text{ただし } Dg(x) \text{ は } x \text{ における } g \text{ の微分})$$

として，$x \mapsto f(x)$ の不動点である．したがって，微分を表す関数を定義できれば，Newton 法の一般解を示す手続きを表すことができる．

微分もまた，手続きを返す手続きとして定義できる．一般に，g が関数で dx が微小なら，g の微分 Dg は, x における値が

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}$$

で与えられる関数であり，以下のように記すことができる事を見てきた．

```
dx = 0.00001
def deriv(g):
   return lambda x: (g(x + dx) - g(x)) / dx
```

deriv を使うことで, Newton 法を不動点のプロセスとして表すことができる. newton_transform は本節の最初の式を表し，それを使うことで newtons_method を定義することができる．

```
def newton_transform(g):
    return lambda x: x - g(x) / deriv(g)(x)

def newtons_method(g, guess):
    return fixed_point(newton_transform(g), guess)
```

平方根を見つけるためには，関数 $y \mapsto y^2 - x$ の零点を見つける Newton 法を使うことができる．

```
def my_sqrt(x):
    return newtons_method(lambda y: square(y) - x, 1.0)
```

1.5.4 関数の変換の不動点探索手続きを用いた平方根の計算

ここまで平方根を計算する方法を，不動点探索として，また，Newton 法の応用として２つの手続きを見てきた．Newton 法自体は不動点探索プロセスとして表せることから，双方とも関数のある変換の不動点を見つけていると言える．この手続きは以下のように書ける．

```
def fixed_point_of_transform(g, transform, guess):
    return fixed_point(transform(g), guess)
```

この手続きは引数としてある関数を計算する手続き g，g を変換する手続き transform，最初の予想値 guess を取り，結果として変換された関数の不動点を返す．

この高度に抽象化された手続きを用いると，関数 $y \mapsto x/y$ を平方緩和した不動点を用いた平方根の計算も，関数 $y \mapsto y^2 - x$ の Newton 変換の不動点を用いた平方根の計算の双方を記述することができる．

```
def my_sqrt(x):
    return fixed_point_of_transform(lambda y: x / y,
                                    average_damp,
                                    1.0)
def my_sqrt(x):
    return fixed_point_of_transform(lambda y: square(y)-x,
                                    newton_transform,
                                    1.0)
```

1.6 抽象と第一級手続き (first-class procedures)

ここまで合成式，引数としての手続き，値として返される手続きと様々な（かつ強力な）抽象を作り出すための手法を紹介してきた．一般に抽象化をすることで，より強力なプログラムを書くことができるが，プログラムを可能な限り抽象的に書くべき，というわけではない．経験を積んだプログラマは自分の仕事に適した抽象のレベルを選ぶことの大切さを知っている．しかし，抽象を使って考えることができることは，新しい状況に対してすぐに応用できるため大切な考え方である．

プログラム言語において，制約の殆ど無い要素は第一級 (first class) 身分を持つとする．第一級要素の権利と特権は

- 変数を用いて名前をつけることができる

- 手続きへ引数として渡すことができる
- 手続きを結果として返すことができる
- データ構造の中に含めることができる

である．

これ以外に，関数内で新しい関数を定義することができる，という条件をつける場合もある．ここで見てきた Python は関数第一級要素である．

また，C 言語でも，関数ポインタを用いることで引数に関数を渡したり，関数を返り値として返すことが可能になっている．また，最近の C++の仕様では無名関数も含めて柔軟な記述が可能になっている．

2 宿題

提出先：ITC-LMS を用いて提出すること

提出内容：以下の問題の実行結果の画面をキャプチャしファイル名は「問題番号.png」とし，また講義中にでてきたキーワードについて知らなかったもの, 興味のあるものを調べ「学籍番号.txt」としててアップロードすること．テキストファイルはワードファイルなどだと確認出来ないことがあるため，emacs/vi 等のテキストエディタを使って書こう．プログラムが長くなりキャプチャ画面に入り切らなくなってきたらプログラムファイルと実行結果を「問題番号.txt」にまとめてアップロードしてよい．
画像で提出する場合は，各自のマシンの Mac アドレスが分かるようにすること．例えば画面中に ifconfig というコマンドを打ち込んだターミナルを表示すればよい．
ITC-LMS にアップロードする際には講義・宿題の感想を必ずコメントに記すこと．また授業中に質問した者はその旨を記すこと．質問は成績評価時の加点対象となる．

キーワード：高階関数，無名関数，第一級オブジェクト

1. 以下に示す Simpson の公式を用いてより正確な数値積分が可能になる．すなわち，a から b までの関数 f の積分は以下の式で近似できる．

$$\int_a^b f = \frac{h}{3}\left[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n\right]$$

なお偶整数 n に対し $h = (b-a)/n$，$y_k = f(a+kh)$ とする．

引数として，`f`，`a`，`b`，`n` を取る積分値を返す関数を `sum` を用いて実装し，n = 5, 10, と 100 について 0 から 1 までの `cube` の積分を計算せよ．

また，1.2.2 節の `integral` を用いた 0 から 1 までの `cube` の積分の計算結果と比較せよ．

ヒント Simpson の公式は

$$\frac{h}{3}\sum_{k=0}^{n}(g\ y(k))$$

となる．ただし，g は k が奇数のときは 4, 偶数のときは 2, また，k が 0, n のときは 1 とする．すなわち，$(g\ y(k))$ は

```
    def term(k):
        if k == 0 or k == n:
            return 1 * y(k)
        elif k % 2 == 0: # even
            return 2 * y(k)
        elif k % 2 == 1: # odd
            return 4 * y(k)
```

と書くことが出来る.

2. 引数として手続きをとり, 受け取った手続きを二回作用させる手続きを返す手続き double は以下のように定義できる. 例えば inc が引数に 1 を足す手続きとすれば, double(inc) は 2 を足す手続きとなる.

```
>>> def inc(x) : return x + 1
>>> def double(f) : return (lambda x : (f(f(x))))
>>> double(inc)(0)
2
```

このとき, (double(double))(inc)(0) は 4 を返すが, その理由を述べよ

3. （発展課題）本資料の内容を Python 以外で書いてみよ. Scheme/Lisp であれば付録を参照せよ. また最近の C++を使ってもよい.

```
// https://gcc.gnu.org/projects/cxx-status.html
// c++ -o sum sum.cpp
#include <iostream>

#define dx 0.00001

double cube(double x) { return x * x * x; }

auto deriv(auto g){
  return [&, g](auto x) { return ((g(x + dx) - g(x)) / dx); };
}

int main() {
  auto four = [](auto x) { return x + 4; };
  printf("lambda x : x + 4 = %d, %d\n", four(1), [](auto x) { return x + 4; }(1));

  printf("deriv(cube)(5) = %f\n", (deriv(cube))(5.0));
}
```

宿題 1

与える関数によって出力が異なる．動くプログラム，動かないプログラム，答えが正しいもの，正しくないもの混在するので注意すること．

simpson の公式を用いて0から1までのcubeの積分を計算するプログラムを記せ

以下は Python での実装例です。

python Copy code

```python
def cube(x):
    return x * x * x

def sum(term, a, next, b):
    if a > b:
        return 0
    else:
        return term(a) + sum(term, next(a), next, b)

def simpson_integral(f, a, b, n):
    h = (b - a) / n
    y = lambda k: f(a + k * h)

    integral = (h / 3) * (
        sum(y, 0, lambda k: k + 1, n // 2 - 1) +
        4 * y(n // 2) +
        sum(y, n // 2 + 1, lambda k: k + 1, n - 1)
    )

    return integral

result = simpson_integral(cube, 0, 1, 1000)
print(result)  # 結果は 0.25 に近い値になるはず
```

ここで、`simpson_integral` 関数が Simpson の公式に基づいて積分を計算しています。`f` は被積分関数、`a` と `b` は積分区間、`n` は分割数を表します。具体的には、区間 `[a, b]` を `n` 等分し、それぞれの部分区間で Simpson の公式を適用した値の和を返します。`sum` 関数は区間 `[a, b]` を等分するために使用されています。`y` 関数は、`k` 番目の区間の中央点での `f` の値を計算するために使用されます。

simpson の公式を用いて0から1までのcubeの積分を計算するプログラムを記せ．ただし，

```
def sum(term, a, next, b):
 if a > b:
  return 0
 else:
  return term(a) + sum(term, next(a), next, b)
```

のsum関数を用いること．

以下がsimpsonの公式を用いたプログラムです。cube(x)はxの3乗を返す関数です。

python Copy code

```python
def sum(term, a, next, b):
    if a > b:
        return 0
    else:
        return term(a) + sum(term, next(a), next, b)

def cube(a):
    return a * a * a

def simpson(n):
    def y(k):
        return cube(k/n)

    def term(k):
        if k == 0 or k == n:
            return 1 * y(k)
        elif k % 2 == 0: # even
            return 2 * y(k)
        elif k % 2 == 1: # odd
            return 4 * y(k)

    return (b - a) * sum(term, 0, inc, n) / (3 * n)

a, b, n = 0, 1, 100 # 積分区間[0, 1], 分割数100
print(simpson(n))
```

出力:

Copy code

```
0.25000000000000006
```

このプログラムでは、関数`simpson(n)`に分割数nを渡すことで、simpsonの公式を用いた積分値を計算しています。ここで、内部に定義された`y(k)`関数は、積分の被積分関数f(x)に相当します。また、`term(k)`関数では、各項の重みを計算しています。最終的に、`sum`関数を用いて、計算された各項

simpson の公式を用いて0から1までのcubeの積分を計算するプログラムを記せ．ただし，

```
def cube(a) return a * a * a
def sum(term, a, next, b):
 if a > b:
  return 0
 else:
  return term(a) + sum(term, next(a), next, b)
def inc(x): return x + 1

def term(k):
    if k == 0 or k == n:
     return 1 * y(k)
    elif k % 2 == 0: # even
     return 2 * y(k)
    elif k % 2 == 1: # odd
     return 4 * y(k)
```

の各関数を用いること．

以下が、simpsonの公式を用いて0から1までの`cube`関数の積分を計算するプログラムです。

python Copy code

```python
def cube(a): return a * a * a

def sum(term, a, next, b):
    if a > b:
        return 0
    else:
        return term(a) + sum(term, next(a), next, b)

def inc(x): return x + 1

def term(k):
    if k == 0 or k == n:
        return 1 * y(k)
    elif k % 2 == 0: # even
        return 2 * y(k)
    elif k % 2 == 1: # odd
        return 4 * y(k)

def integral(f, a, b, n):
    h = (b - a) / n
    y = lambda k: f(a + k * h)
    return (h / 3) * sum(term, 0, inc, n)

# 積分の計算
n = 1000
result = integral(cube, 0, 1, n)
print(result)  # 積分値の出力
```

ここで、`integral`関数は`f`が被積分関数、`a`と`b`が積分区間、`n`が区間を分割する数、`y`が中間点に対応する関数を表します。

また、`term`関数はsimpsonの公式の中の積分項を表し、`sum`関数は区間を分割して得られた値を足し合わせる関数です。`inc`関数は、区間を分割するための関数で、`n`回目の計算で用いる中間点

宿題２
返り値が間違えているので注意.

def inc(x): return x+1
def double(f) :return(lambda x:(f(f(x))))
と定義したとき，(double(double))(inc)(0) の結果は何か，またその理由を述べよ

`(double(double))(inc)(0)` の結果は、数値3になります。

これは、`double(double)` が `lambda x:(f(f(x)))` を二回適用することにより、`inc` 関数が二回適用され、引数 `0` が2増加して `2` になり、最後に `inc` 関数が1回適用されて、結果が `3` になるためです。

具体的には、以下のように式を展開できます。

scss Copy code

```
(double(double))(inc)(0)
= (lambda x:(double(double))(inc)(inc(x)))(0)  # double(double) = lambda x:(
= (double(double))(inc)(inc(0))
= (double(lambda x:(inc(inc(x)))))(inc(0))
= lambda x:(lambda x:(inc(inc(x))))(lambda x:(inc(inc(x))))(inc(0))
= (lambda x:(inc(inc((inc(inc(x)))))))(inc(0))
= inc(inc(inc(inc(0))))
= 3
```

以上のように、`(double(double))(inc)(0)` は `3` と評価されます。

A Emacs による例

A.1 宣言文的記述と命令文的記述

プログラムの記述方法には宣言文的記述 (Declarative programming) と命令文的記述 (Procedural programming) の 2 つがある．前者はある出力を得るための手続きを，その出力の性質で持って表そうとするものであり，後者はその出力をえるための手続きを示したものである．

例えば，x の平方根 y を計算する関数を考えると，

$$\sqrt{x} = y \geq 0 \text{ かつ, } y^2 = x \text{ であるような } y$$

と定義できる．これは宣言文的記述（なにであるか？）であるが，プログラミングでは命令文的記述（どうしたらいいか？）として，平方根を見つけるための手続きを示す必要がある．

A.1.1 線形的プロセスと反復的プロセス

階乗を計算するプログラムを書いてみよう．これは，

$$n! = n \cdot (n-1)!$$

と定義できるところから，以下の様な再帰プログラムとして書くことができる．

```
(defun ! (n) (if (= n 1) 1 (* n (! (- n 1)))))
```

慣れないとビックリするが，! が関数名である．(! 6) と打ち込んだ所で C-j を打てば以下の様に結果を得ることが出来る[3]．

```
(! 6)
720
```

一方，別の計算手続きとして，まず 1 に 2 を掛け，その結果に 3 を掛け，n になるまで続ける手続きを考える．カウンタを 1 から n まで数える間，部分積を保持しつつ，

$$\text{積} \leftarrow \text{カウンタ} \cdot \text{積}, \quad \text{カウンタ} \leftarrow \text{カウンタ} + 1$$

を毎ステップ計算する．

```
(defun factoral (n) (fact-iter 1 1 n))
(defun fact-iter (product counter max-count)
  (if (> counter max-count)
      product
    (fact-iter (* counter product)
       (+ counter 1)
       max-count)))
```

両方の関数ともに，関数自体が自らを呼び出すという意味で再帰手続き (recursive procedure) であるが，その実行過程（プロセス）の観点から区別すると前者は線形再帰プロセス，後者を線形反復的プロセスと呼ぶことができる（fact-iter は再帰手続きが反復的プロセスで実行されている)．

[3] tak 関数は以下のように定義できる．ただし emacs 上で評価すると ‘max-lisp-eval-depth‘となるだろう clisp などのインタプリタを立ち上げて実行してみよう
(defun f (x y z) (if (<= x y) z (f (f (- x 1) y z) (f (- y 1) z x) (f (- z 1) x y))))

実行過程を以下に示す．なお，これは手続きの呼び出し時に，各仮パラメタを対応する引数で置き換え，手続きの本体を評価する，という作業を繰り返している．これを置き換えモデルと言う．

```
(! 6)
(* 6 (! 5))
(* 6 (* 5 (! 4)))
(* 6 (* 5 (* 4 (! 3))))
(* 6 (* 5 (* 4 (* 3 (! 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (! 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

```
(factoral 6)
(fact-iter   1 1 6)
(fact-iter   1 2 6)
(fact-iter   2 3 6)
(fact-iter   6 4 6)
(fact-iter  24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
```

線形再帰プロセスでは膨張している時はプロセスの遅延演算が発生し，収縮していくときに実際の演算が行われている．これは，(! 5) が呼ばれるときに，この結果と 6 を掛けなければいけないというふうに，後で実行する演算を覚えておく必要があり，遅延演算と呼ばれ，この分だけ n に比例して覚えておくのに必要な記憶量が必要となる．

一方，後者を線形反復的プロセスでは，各ステップで覚えておくべき変数は状態変数 (product, counter, max-count) の数だけである．計算途中のどの時点でも状態変数でプロセスの状態を持っており，計算を途中で止めても，それを再開することが容易である．反復的プロセスは再帰的手続きとして記述されていても固定スペースで実行できる．この様な性質の実装を末尾再帰的 (tail recursion) と呼ぶ．

ハードウェアの観点からすると，前者を実行するにはスタックというデータ構造を必要とするが，後者はレジスタだけで実現できる．

ちなみに，C などの言語では再帰手続きの実行時に必要となる記憶量が常に手続きの数だけ増えるように設計されており，そのために do, for, while などの反復プロセス (繰り返し) 用の構文を必要としているが，これは欠点である．再帰手続きで記述されていても，反復的プロセス (線形反復的プロセス) として固定記憶量で実行する言語も存在する.) .

A.2 手続きの抽象化

A.2.1 Newton 法による平方根の計算

平方根を求める手続きには Newton 法がある．すなわち，数 x の平方根の値の予測値 y を使って，y と x/y の平均をとることで，更に良い予測値を得るものである．

希望的思考 (wishful thinking) によるプログラミングの考え方に従うと，まず，数 x とその予測値 y から出発する．予測値が目的に十分ならそこで終了とする．そうでなければ，予測値 y を改善し同じ手順を繰り返す，という基本戦略を立てる．

```
(defun sqrt-iter (guess x)
  (if (good-enough? guess x)
      guess
    (sqrt-iter (improve guess x) x)))
```

改善された予測値 y は数 x を予測値 y による商との平均値として計算する．

```
(defun improve (guess x) (average guess (/ x guess)))
(defun average (x y) (/ (+ x y) 2))
```

最後に,「十分に良い」の定義が必要になる．abs，square は前節までに定義してある．

```
(defun good-enough? (guess x) (< (abs (- (square guess) x)) 0.001))
(defun my-sqrt (x) (sqrt-iter 1.0 x))
```

プログラムを実行すると以下の様に計算できる．実は EmacsLisp にはあらかじめ sqrt は組み込まれていた．結果を比較すると良い．

```
(my-sqrt 10)
3.162277665175675 <-- 返り値
(sqrt 10)
3.1622776601683795 <-- 返り値
```

A.2.2 一般的方法としての手続き

前節では数値演算のパターンを特定の値は依存しないように抽象する機構として合成手続き (compound procedures: 関数を用いて関数を定義すること) を紹介した．高階手続き (higher-order procedures) では，特定の関数に依存しない，計算の一般的方法を示すための方法を示した．

本節では，不動点を探す一般的方法を手続きとして記す方法を示す．

x が $f(x) = x$ を満たすとき，x を関数 f の不動点 (fixed point) と呼ぶ．この値を求めるためには，ある関数 f について，最初の予測値からはじめ $f(x)$ を計算し，その結果に対して更に f を作用させる $f(f(x))$ を計算し,，f の値が変化しなくなる（あるいは十分に小さい変化になる）まで繰り返し作用させる．

この考え方を使うことで，入力として関数と最初の予測値を取り，その関数の不動点の近似値を返す手続き fixed-point を作ることができる．関数を 2 つの連続する値の差が以下の許容量 tolerance より小さくなるまで繰り返し作用させる．

```
(setq tolerance 0.00001)

(defun close-enough? (v1 v2)
  (< (abs (- v1 v2)) tolerance))
(defun try (f guess)
  (let ((next (funcall f guess)))
    (if (close-enough? guess next)
        next
      (try f next))))
(defun fixed-point (f first-guess)
  (try f first-guess))
```

このプロセスは平方根を求めるプロセスと類似している．平方根の計算は $y^2 = x$ となる y を探すことであることから，関数 $y \mapsto x/y$ の不動点を計算していると考え以下のように記すことができる．

```
(defun my-sqrt (x) (fixed-point (lambda (y) (/ x y)) 1.0))
```

しかしこの不動点探索計算は収束しない．最初の予測値を y_1 とすると，次の予測値は $y_2 = x/y_1$ で，その次の予測値は $y_3 = x/y_2 = x/(x/y_1) = y_1$ と無限ループになる．

これを制御するためには，予測値の大きな変化を防ぐことにある．答えは予測値 y と x/y の間にあることから，これらの平均とすることができる．つまり，y の次の予測値は x/y ではなく，$\frac{1}{2}(y + \frac{x}{y})$ とするとよい．すなわち以下のように書くことができる．

```
(defun my-sqrt (x) (fixed-point (lambda (y) (average y (/ x y))) 1.0))
```

この修正により以下のように平方根の計算を行うことができる．この様に解への逐次近似を平均化する方法を平均緩和法 (average damping) と呼ぶ．

```
(my-sqrt 10)
3.162277660168379 <-- 返り値
```

A.2.3 値として返される手続き

前節では $\sqrt{x}$ が関数 $y \mapsto x/y$ の不動点であるとの認識から，不動点探索の手続きを用いた平方根の計算方法を示した．また，収束のために平均緩和の考え方を用いた．平均緩和はそれ自体を有用な一般的な技法と考えることができる．つまり以下の手続き[4]として定義できる．

```
(setq lexical-binding t)
(defun average-damp (f) (lambda (x) (average x (funcall f x))))
```

これは，引数として手続き `f` を取り，その値として，`lambda` で作り出された，ある数 `x` と `(f x)` の平均を返す手続き，を返す手続きである．例えば，`average-damp` を `square` に作用させると，ある数 `x` と `x * x` の平均を返す手続きを返す．すなわち `(average-damp #'square)` を評価すると以下のように手続きが返る．

```
(average-damp #'square)
(closure ((f . square) t) (x) (average x (funcall f x))) <-- 返り値
```

この返された手続きに 10 を作用させると 10 と 100 の平均として 55 が返る．

```
(funcall (average-damp #'square) 10)
55 <-- 返り値
```

これにより不動点探索を用いた平方根の手続きは以下の様に書くことができる．

```
(defun my-sqrt (x)
   (fixed-point (average-dump (lambda (y) (/ x y))) 1.0))
```

ここでは，不動点の探索，平均緩和，$y \mapsto x/y$ を使って平方根を計算している．この書き方は本節でここまで紹介してきた `my-sqrt` と同じプロセスを表しているが，抽象度合いが異なっていることに注意しよう．この様に一つの手続きでもその表現は抽象度に応じて様々あるが，どのレベルの部分を選ぶか，また，どの部分が他の応用にも再利用できるようなまとまった存在か，という観点が重要である．

[4] ここでは lexical-binding を nil 以外の値にセットし，レキシカル拘束として定義する必要がある．

再利用の例として，x の立方根が関数 $y \mapsto x/y^2$ の不動点であることに注意すれば，以下のように記述できる．

```
(defun my-cube-root (x)
   (fixed-point (average-dump (lambda (y) (/ x (square y)))) 1.0))
```

A.2.4 Newton 法

Newton 法は $x \mapsto g(x)$ が微分可能であれば，方程式 $g(x) = 0$ の解は以下のニュートン変換

$$f(x) = x - \frac{g(x)}{Dg(x)} (\text{ただし } Dg(x) \text{ は } x \text{ における } g \text{ の微分})$$

として，$x \mapsto f(x)$ の不動点である．したがって，微分を表す関数を定義できれば，Newton 法の一般解を示す手続きを表すことができる．

微分もまた，手続きを返す手続きとして定義できる．一般に，g が関数で dx が微小なら，g の微分 Dg は, x における値が

$$Dg(x) = \frac{g(x + dx) - g(x)}{dx}$$

で与えられる関数であり，以下のように記すことができる．

```
(setq dx 0.00001)
(defun deriv (g) (lambda (x) (/ (- (funcall g (+ x dx)) (funcall g x)) dx)))
```

例えば関数 $x \mapsto x^3$ の 5 での微分は $x \mapsto 3 \cdot 5^2 = 75$ だが，以下のように計算することができる．

```
(defun cube (x) (* x x x ))
(funcall (deriv #'cube) 5)
```

deriv を使うことで，Newton 法を不動点のプロセスとして表すことができる．newton-transform は本節の最初の式を表し，それを使うことで newtons-method を定義することができる．

```
(defun newton-transform (g)
    (lambda (x) (- x (/ (funcall g x) (funcall (deriv g) x)))))
(defun newtons-method (g guess)
  (fixed-point (newton-transform g) guess))
```

平方根を見つけるためには，関数 $y \mapsto y^2 - x$ の零点を見つける Newton 法を使うことができる．

```
(defun my-sqrt (x) (newtons-method (lambda (y) (- (square y) x)) 1.0))
```

ここまで平方根を計算する方法を，不動点探索として，また，Newton 法の応用として 2 つの手続きを見てきた．Newton 法自体は不動点探索プロセスとして表せることから，双方とも関数のある変換の不動点を見つけていると言える．この手続きは以下のように書ける．

```
(defun fixed-point-of-transform (g transform guess)
  (fixed-point (funcall transform g) guess))
```

この手続きは引数としてある関数を計算する手続き g，g を変換する手続き transform，最初の予想値 guess を取り，結果として変換された関数の不動点を返す．

この高度に抽象化された手続きを用いると，関数 $y \mapsto x/y$ を平方緩和した不動点を用いた平方根の計算も，関数 $y \mapsto y^2 - x$ の Newton 変換の不動点を用いた平方根の計算の双方を記述することができる．

```
(defun my-sqrt (x)
   (fixed-point-of-transform (lambda (y) (/ x y))
                             #'average-damp
                             1.0))
(defun my-sqrt (x)
   (fixed-point-of-transform (lambda (y) (- (square y) x))
                             #'newton-transform
                             1.0))
```

A.3 データの抽象化

手続きを組み合わせたものを合成手続きと呼び，データを組み合わせたものを合成データ (compound data) と呼ぶ．合成データを作ることでデータを抽象化できる．

例えば線型結合

$$ax + by$$

を作る手続きは以下の様になる．

```
(defun linear-combination (a b x y)
  (+ (* a x) (* b y)))
```

これで，以下の様に計算できる．

```
(linear-combination 1 2 3 4)
11
```

しかし，数値だけでなく，有理数，複素数，多項式の線型結合を考えたい場合は以下のようにすればいい．

```
(defun linear-combination (a b x y)
  (add (mul a x) (mul b y)))
```

add, mul はデータの種類に合わせた演算を実行する手続きになる．ポイントは，linear-combination の手続きからは，a,b,x,y のデータの中身が何であるかは無関係であり，プログラムはデータを扱うのではなく，合成データを扱うことになる．

A.3.1 希望的思考によるプログラミング

さて，これを有理数でも使えるようにするために，まず以下の様に考える．

- (make-rat n d) 分子が n, 分母が d の有理数を返す
- (numer x) 有理数 x の分子を返す.
- (denom x) 有理数 x の分母を返す.

の 3 つの手続きが使えるとする[5].

このように．手続きを実際に実装する前に，それを使ってプログラムを考え，組み立てていくことを希望的思考 (wishful thinking) と呼ぶ．これは，プログラミングに置ける重要な戦略である．

これを使った add, mul の手続きは以下になる．

```
(defun add-rat (x y)
  (make-rat (+ (* (numer x) (denom y))
       (* (numer y) (denom x)))
    (* (denom x) (denom y))))
(defun mul-rat (x y)
  (make-rat (* (numer x) (numer y))
    (* (denom x) (denom y))))
```

A.3.2 対による有理数の作成

次に，有理数を作るため，cons という手続きで作成する，対 (pair) というデータ構造を導入する．cons は 2 つの引数をとり，それを部分として含むデータを返す．また，car, cdr という手続きにより，それぞれの部分を取り出すことが出来る．

```
(setq x (cons 1 2))
(1 . 2)
(car x)
1
(cdr x)
2
```

これをつかって，有理数を対で表そう．

```
(defun make-rat (n d) (cons n d))
make-rat
v(defun numer (x) (car x))
numer
(defun denom (x) (cdr x))
denom
```

また，結果表示用の手続きを以下の様に定義する．

```
(defun print-rat (x) (prin1 (numer x))(prin1 '/)(prin1 (denom x))(print nil))
```

これで以下の様に有理数の計算が出来る．

[5]有理数（rational number），分子（numerator），分母（denominator）である

```
(setq one-half (make-rat 1 2))
(1 . 2)
(setq one-third (make-rat 1 3))
(1 . 3)
(print-rat one-half)
1/2
nil
(print-rat one-third)
1/3
nil
(print-rat (add-rat  one-half one-third))
5/6
nil
(print-rat (mul-rat one-half one-third))
1/6
nil
```

A.3.3 データの種類に見合った演算

これで，数値か有理数に関係なく演算を行う例として以下の mul, add を定義する．

```
(defun add (x y)
  (cond ((consp x) (add-rat x y))
(t         (+ x y))))
add
(defun mul (x y)
  (cond ((consp x) (mul-rat x y))
(t         (* x y))))
```

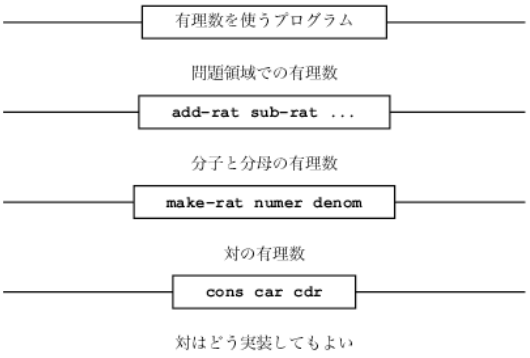
これで以下の様にデータに種類に依存せずに linear-combination が使えるようになる．

```
(defun add (x y)
  (cond ((consp x) (add-rat x y))
(t         (+ x y))))
add
(defun mul (x y)
  (cond ((consp x) (mul-rat x y))
(t         (* x y))))
mul
(linear-combination 1 2 3 4)
11
(print-rat (linear-combination one-half one-half one-third one-third))
12/36
```

A.3.4 データの抽象のレベル

ここで作ったプログラムは以下のような抽象のレベルがあることが分かる．つまり，add-rat, mul-rat の有理数を操作するレベル，make-rat, numer, denom の有理数を生成，選択するレベル，cons, car, cdr の対により有理数を扱うレベル．である．各レベルの手続きはデータ抽象のレベルとなり，異なるデータ構造を使って有理数を扱う場合のインターフェースを決めていることになる．例えば対による計算が非常に効率が悪いなどの問題がある場合，一番したのレベルを取り替えれば変更することができ，プログラムの全てでなく，一部を変えることで対応できるようになる[6]．

[6]Unix 哲学でもでてきたモジュール化と共に考えてみると良い


有理数を使うプログラム
問題領域での有理数
add-rat sub-rat ...
分子と分母の有理数
make-rat numer denom
対の有理数
cons car cdr
対はどう実装してもよい