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

5. ソフトウェアの抽象化 (データ)

1　データの抽象化

1.1　データによる抽象の構築

前回は式（手続き）を組み合わせた合成式 (compound expression) を用いた抽象の構築を紹介したが，本章では，データオブジェクトを組み合わせた合成データ (compound data）を作って抽象を構築できる機能を扱う．

合成データを用いるとプログラムの言語能力を格段に大きくすることができる．例えば線型結合 $ax + by$ を考えて見よう．引数が数値なら，この線型結合を作る手続きは以下の様になる．

```
def linear_combination(a, b, x, y):
   return 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を考えよう．例えば，有理数，複素数，多項式やその他の線型結合ができることを手続き的に表したいとしよう．これは，手続きとして，

```
def linear_combination(a, b, x, y):
   return add(mul(a, x), mul(b, y))
```

と表すことができる．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．

例えば引数が数値であれば

```
def add(a, b):
    return a + b

def mul(a, b):
    return a * b
```

となるが，add, mul が有理数など引数 a，b，c，d に渡したデータに見合った計算をすることができれば，それぞれのデータに応じた線型結合の計算ができる．

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/種類が何であるかは無関係であり，必要なことは手続き add，mul がデータの形式に応じた適切な操作をすることだけである．

1.2 希望的思考 (wishful thinking) によるプログラミング

さて，線型結合を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[1]．

ここでは，有理数がどう表されているか，また，手続き make_rat，numer，denom がどう実装されているかは説明していない．しかし，この 3 つの手続きがあれば，次の有理数の四則演算を実装することができる．

このように，すべての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．

上に示した手続き使った add，mul の手続きは以下になる．

```
def add_rat(x, y):
   return make_rat(numer(x)*denom(y) + numer(y)*denom(x), denom(x)*denom(y))

def mul_rat(x, y):
   return make_rat(numer(x)*numer(y), denom(x)*denom(y))
```

これで，make_rat，numer，denom を使って有理数に関する演算を定義することができた．しかし，make_rat，numer，denom はまだ定義していない点に注意しよう．

1.3 タプル (tuple) による有理数の作成

1.3.1 タプル (tuple)

データ抽象の具体的な実装をするために，Python では以下のタプル (tuple) と呼ばれる合成構造 (compound structure) を提供している．

```
>>> (1, 2)
(1, 2)
```

タプル (tuple) はリスト (list) と () か [] の違いだけに見えるが，リストと異なり，一旦生成したあとは変更ができないデータ形式になっている．この性質のことを，イミュータブル (immutable) と呼ぶ．変更ができるデータは (mutable) と呼ぶ．

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

```
>>> a = (1, 2)
>>> b = [1, 2]
>>> a
(1, 2)
>>> b
[1, 2]
>>> a[0]
1
>>> b[0]
1
```

```
>>> a[0] = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not
  support item assignment
>>> a
(1, 2)
>>> b[0] = 100
>>> b
[100, 2]
```

タプルの要素は以下のようにして取得することができる.

```
>>> pair = (1, 2)
>>> pair
(1, 2)
>>> pair[0]
1
>>> pair[1]
2
```

```
>>> x, y = pair
>>> x
1
>>> y
2
```

1.3.2 有理数の作成

次に，タプル (tuple) をつかって，有理数を表そう.

```
def make_rat(n, d):
    return (n, d)
def numer(x):
    return x[0]
def denom(x):
    return x[1]
```

また，結果表示用の手続きを以下の様に定義する.

```
def str_rat(x):
        return '{}/{}'.format(numer(x), denom(x))
```

これで以下の様に有理数の計算が出来る.

```
>>> one_half = make_rat(1, 2)
>>> str_rat(one_half)
'1/2'
>>> one_third = make_rat(1, 3)
>>> str_rat(one_third)
'1/3'
>>> str_rat(add_rat(one_half, one_third))
'5/6'
>>> str_rat(mul_rat(one_half, one_third))
'1/6'
>>> str_rat(add_rat(one_third, one_third))
'6/9'
```

最後の例は，有理数の簡約しないことがわかる．２つの整数の最大公約数を計算する関数があれば，make_rat を変更することでこれを改善できる．

以下は Python のライブラリを用いて用いた例になっている．もちろん自分で計算しても良い．

```
from fractions import gcd
def make_rat(n, d):
    g = gcd(n, d)
    return (n//g, d//g)
```

//は整数の商の算出をするオペレータである．

ここで重要なのは，この修正は make_rat の変更だけで実現できる点にある．すなわち，add_rat, mul_rat などの実際に演算を実装する手続きは変しない点に注意してもらいたい．これは，後述する「抽象の壁 (Abstraction Barriers)」の節でも説明する．

```
>>> str_rat(add_rat(one_third, one_third))
'2/3'
```

1.4 データの種類に見合った演算

ここ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．データがタプルかどうかは type(pair) is tuple) として評価することができる．

```
def add(a, b):
    if type(a) is tuple and type(b) is tuple:
        return add_rat(a, b)
    else:
        return a + b;

def mul(a, b):
    if type(a) is tuple and type(b) is tuple:
        return mul_rat(a, b)
    else:
        return a * b;
```

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[2]．

```
>>> linear_combination(1,2,3,4)
11
>>> linear_combination(one_half, one_half, one_third, one_third)
(1, 3)
```

1.5 抽象の壁 (abstraction barriers)

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，タプル (tuple) により有理数を扱うレベル．である．

[2]ここまで示してきた定義だけでは，a, b, c, d のすべてのデータ型が一致しているという制限がある．すなわち数値と有理数の足し算などは計算できない．

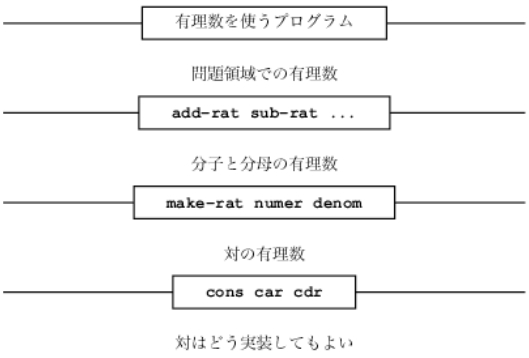
各レベルの手続きはデータ抽象のレベルとなり，異なるデータ構造を使って有理数を扱う場合のインターフェースを決めていることになる．

つまり，以下の図のように水平の線が異なる「レベル」を隔離する抽象の壁 (abstraction barrier) を提供している．それぞれのレベルで壁は，データの抽象を扱う（上の）プログラムと，データ抽象自体を実装する（下の）プログラムを明確に隔ている．

例えば，有理数を扱うプログラムでは，有理数がパッケージが「パブリック」なものとして提供する `add_rat`，`mul_rat` などの手続きは，`make_rat`，`numer`，`denom` などの基本関数だけを用いて実装してあり，これが tuple の要素を得る `x`，`y = pair` として実装されているか，あるいは，`pair[0]`，`pair[1]` として実装しているかの細部は無関係である．

このような，データ抽象の壁を考え方には多くの長所がある．一つはプログラムのメンテナンス（維持），修正が容易なことである．例えば，前節で有理数の簡約を見てきた．ここでは，`make_rat` のレベルで処理がなされ，`add_rat`，`mul_rat` 等，有理数の演算を提供するレベルや，それを用いた計算の手続きは全く修正する必要ない．

また，例えばタプルによる計算が非常に効率が悪いなどの問題がある場合，一番下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[3]．



1.6 データとは何か

本節ではデータと手続きの違いについて考えよう．これまで見てきた有理数の例では，有理数を表すデータ `x` が `make_rat(n, d)` として生成できる場合，任意の整数 `n`, 非零の整数 `d` に対して，以下の条件を満たす必要がある．しかもこれは `make_rat`，`numer`，`denom` で有理数を表現するために満たさなければいけないただ一つの条件になっている．

$$\frac{numer(x)}{denom(x)} = \frac{n}{d}$$

これまでは実際には有理数はタプルを用いて，このデータを表現してきた．ここで，データの生成のための手続きを `cons`，データの要素を取得する手続きを `car`，`cdr` として定義してみよう．なお，`cons`，`car`，`cdr` の記法は SICP で利用されている Scheme/Lisp においてペアを作成する手続き名に由来する．

[3] 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

```
def cons(x, y):
   return (x, y)
def car(z): return z[0]
def cdr(z): return z[1]

def make_rat(n, d): return cons(n, d)
def numer(x): return car(x)
def denom(x): return cdr(x)
```

```
>>> cons(1, 2)
(1, 2)
>>> car(cons(1, 2))
1
>>> cdr(cons(1, 2))
2
>>> one_half = make_rat(1, 2)
>>> numer(one_half)
1
>>> denom(one_half)
2
```

この場合，cons(1, 2) で返されるデータはタプルであり，これまで同様にタプルを用いて有理数を作成することが出来る．

一方，任意のオブジェクト x，y に対して，cons(x, y) により z を作成した場合，car(z) が x であり，cdr(z) が y であるという条件を満たせば，それは有理数を表すデータとして利用できるはずである．それを示したのが以下になる．

```
def cons(x, y):
   def dispatch(m):
      if (m == 0):
         return x
      elif (m == 1):
         return y
   return dispatch

def car(z): return z(0)
def cdr(z): return z(1)
```

```
>>> cons(1, 2)
<function dispatch at 0x7faedb6b3b50>
>>> car(cons(1, 2))
1
>>> cdr(cons(1, 2))
2
```

ここでは，cons(x, y) で返される値は dispatch として定義された手続きであり，これは引数が 0 であれば x を，1 であれば，y を返す，という仕様を持っている．すなわち，手続きによるペアのデータを実装することが出来ている．

dispatch のようなデータ（実際には手続き）に対して引数を与えてメッセージを送ることで，そのデータに対する操作（必要な情報を引き出す）プログラミングスタイルはメッセージパッシング (Message Passing) と呼ばれる．これは賢いデータと言える．つまり，データに対して必要な演算を入力とし，出力をその入力に応じた手続きとして表現している．

2 モジュール性，オブジェクト，状態

大規模システムを作成する時は，システムを幾つかのまとまりに分割するモジュール化を行う．この時各モジュールへの分割は，容易に開発，メンテナンス，再利用が可能なように「自然」であることが重要になる．モジュール化に際しては，モデル化する対象に構造に基づいてプログラムの構造を決める方法がある．つまり実際の対象となるシステムのそれぞれのオブジェクトに対応する計算オブジェクトを構築する．この計算オブジェクトの構築では対象のモデルをどう見るか，という「世界観」に依存する．

一つは，時間に応じて状態の変化するオブジェクト (Object) の集合としてシステムを捉える見方，もう一つはシステムを流れる情報のストリーム (Stream) としてシステムを捉える見方がある．

本節ではシステムをオブジェクトとして捉えてみよう．オブジェクトの振る舞いが時間と共に変わる（過去に影響される）場合，そのオブジェクトは「状態を持つ」と呼ばれる．その状態はオブジェクト内の状態変数 (State Variable) によって管理される．

オブジェクト指向プログラミング言語においてはこの状態変数はメンバ変数として定義され，外部から隠蔽される．これにより，データの抽象化の壁 (Abstraction Barriers) が作られる．またメンバ関数の呼び出しは，メッセージパッシング (Message Passing) と呼ばれる機構の実現法の一つと捉えられる[4]．

2.1 局所状態変数

時間と共に状態が変化する計算オブジェクトの例として銀行口座からお金を引き下ろす例を考える．ここでは，引き下ろす金額 `amount` を引数に取る手続き `withdraw` を考える．口座に十分なお金がなければ，”Insufficient funds” と表示する．例えば口座に 100 ドルあるところから初めると，withdraw を使い以下の返り値となることが期待される[5]．

```
>>> withdraw(25)
75
>>> withdraw(25)
50
```

```
>>> withdraw(60)
'Insufficient funds'
>>> withdraw(15)
35
```

ここでは，`withdraw(25)` を二度評価（実行）すると異なる返り値が返ってくるこに注意して欲しい．これはここまで見てきた手続きとは異なる．すなわち非純粋関数となっている．また，これにより，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，これまでの評価の置き換えモデルが利用できなくなっている．

`withdraw` の実装は以下の左のようにの 1 つめは口座の残高に対応する大域変数 `balance` を用いるものである．しかしこれは，balance にどの関数からも操作が可能なため危険である．そこで，balance を withdraw 内部に置きつつ，呼び出す度にその値を変更し，かつ記憶しておく様な局所状態変数として扱う方法として，以下の右の実装がある．

ただしこの実装は Python3 でないと動かないため，Ubuntu 18.04 を用いているものは python3 としてインタプリタを立ち上げること．

```
balance = 100
def withdraw(amount):
    global balance
    if balance >= amount:
        balance = balance - amount
        return balance
    else:
        return 'Insufficient funds'
```

```
# Only for Python3
def make_withdraw():
    balance = 100
    def withdraw(amount):
        nonlocal balance
        if balance >= amount:
            balance = balance - amount
            return balance
        else:
            return 'Insufficient funds'
    return withdraw

withdraw = make_withdraw()
```

[4] 本節の内容は SICP 3.1, 2.2 章の要約になっている．`https://sicp.iijlab.net/fulltext/x300.html`

[5] ここでも希望的思考によるプログラミングが使われている．すなわち，withdraw の関数の定義はまだされていない

この技法はクロージャ（Closure：関数閉包）と呼ばれている．クロージャの説明の前に，自由変数 (Free Variable) と呼ばれる概念を紹介する．これは，ある関数内で局所変数と引数以外で使われている変数に対応する．この逆は束縛変数 (Bound Variable) と呼ばれている．また，束縛（バインディング：Binding) とは，値に識別子（名前）を対応付けること意味する．一般に変数はある値への参照を与えている．

クロージャは自由変数のバインディングを，その関数が定義された環境（静的スコープ）において解決する．すなわち，関数とそれを評価する環境の組で表される．一般的には関数の中で関数を定義し，その外側の関数（エンクロージャ）で宣言された変数を，暗黙的に内側の関数から操作可能になっている．例えば，以下では関数 g がクロージャ，関数 f がエンクロージャであり，エンクロージャで定義された変数がクロージャ関数内で利用されている．

例えば以下のプログラムで h(1000) とした時，どのような値が返るか考えてみよう．

```
a = -10
b = -100
c = -1000
def f(a):
    b = 100
    def g(c):
        return a + b + c
    return g # Return a closure.

# Assigning specific closures to variables.
h = f(10)
```

Python3 では自由変数を nonlocal で宣言しておくことで，束縛されている変数をクロージャの中から更新することが出来る[6]．以下のプログラムは，口座の初期金額を設定した上でオブジェクトを生成している例になる．これにより，kei，bill は，それぞれの局所状態変数を持った独立なオブジェクトとなり，一方の口座からのお金の引き出しは他の口座に影響を与えないようになっている．以下は Python2 用のコードである．

```
def make_withdraw(balance):
    class context:                # Python2 needs context, https://t.ly/SUbs
        _balance = balance
    def withdraw(amount):
        # nonlocal balance on Python3 you can use nonlocal
        balance = context._balance
        if amount > balance:
            return 'Insufficient funds'
        balance = balance - amount
        context._balance = balance        # Send back to context for Python2
        return balance
    return withdraw

kei = make_withdraw(100)
bill = make_withdraw(1000)

print (kei(25))
print (bill(25))
print (kei(25))
print (bill(25))
print (kei(60))
print (bill(60))
```

[6]Python のマニュアルではクロージャという単語を使わず説明がなされている．クロージャは計算機プログラム一般で利用する概念を表す用語であり，各言語毎に個別の用語で実装されることがある．https://docs.python.org/ja/3/reference/simple_stmts.html#the-nonlocal-statement

2.2 評価の環境モデル (Enviroment Model)

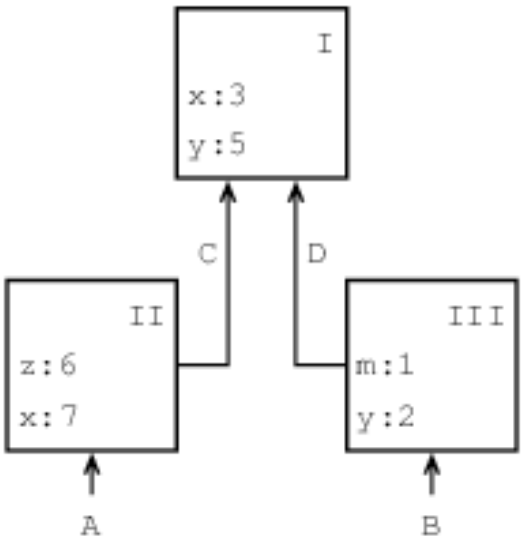
置き換え評価モデルでは，以下の置き換えモデル (Substitution Model) を用いて評価していた．

- 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．

つまり，

```
def square(x):
    return x * x
def sum_of_squares(x, y):
    return square(x) + square(y)
def f(a):
    return sum_of_squares(a + 1, a * 2)
```

という手続きがあった時，手続きを用いて定義された手続き（合成手続き，組み合わせ）`f(5)` を評価する際には，まず `f` の本体を取り出し `sum_of_squares(a + 1, a * 2)` 次に，仮パラメタ `a` を引数 `5` で置き換え，`sum_of_squares` を評価する．`sum_of_squares` の評価にもまた置き換えモデルを適用する．



一方で，局所状態変数や束縛（変数）を導入したプログラミングでは，これらの値を保持する環境 (Environment) と呼ばれる構造を導入する．環境はフレーム (Frame) の列から構成され，各フレームは拘束の表を持っている．拘束は，変数名とその値を対応付けている．各フレームは大域 (Global) 以外のものは外側の環境（Enclosing Environment）へのポインタを持っている．

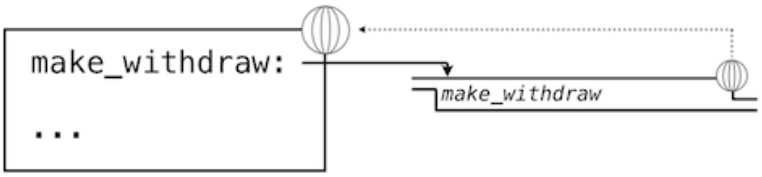
ある環境において変数の値（Value of a Variable）とは，その変数に対する束縛を含んでいる最初のフレームのなかで，その変数の束縛で与えられる値になっている．フレームの並びのどれもがその変数に対する束縛が規定されていなければ，その変数は，その環境で未束縛 (Unbound) と呼ぶ．

上の図は `I`，`II`，`III` の３つのフレームを持ち，`A`，`B`，`C`，`D` は環境である．それぞれの環境は，フレームへのポインタ（図中の矢印）を持っている．ここでは，環境 `C`，`D` の元では，フレーム `I` を参照するため，`x` の値は `3` となる．一方，環境 `A` では，最初のフレームである `II` を参照するこ

とで，x の値は 7 となる．環境 A は，フレーム II と I の列から構成され，フレーム II における x と 7 の束縛が，I における x と 7 の束縛を隠蔽 (shadow) しているという．

環境をモデルを用いた評価の例を見てみよう[7]．

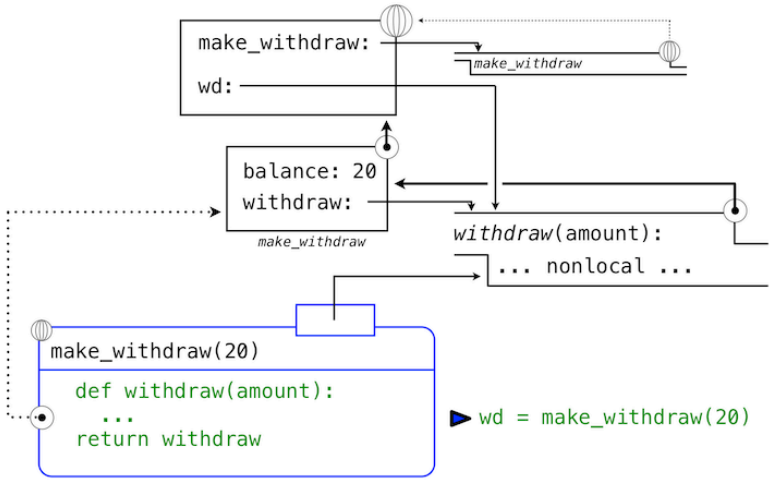
まず，関数 make_withdraw と定義した際には，以下のように大域フレームに関数を定義し，その中でを make_withdraw という名前に束縛している．左の四角がフレームであり，右側で大域フレームで定義した関数を表している．関数からフレームへの破線矢印は，この関数がこのフレームで事項されていることを示し，地球儀のマークで大域（Global) フレームであることを示す．



関数 make_withdraw を定義した状態

次に，これを引数 20 で呼び出す．

```
>>> wd = make_withdraw(20)
```



make_withdraw(20) を呼び出した状態

これにより，大域フレームにおいて，wd という名前に，make_withdraw の返り値となる関数 *withdraw* を束縛している．*withdraw* 関数は，make_withdraw が実行された際に作成される局所フレームに定義され，また，変数 balance も，この局所フレーム内で定義されている．

withdraw 関数から balance:，withdraw:を含むフレームへの実線矢印は，この関数が局所フレーム内で定義されていることを示す．また，左下では make_withdraw(20) という式の評価がグローバルフレームで評価され，その本文は局所フレームを参照している様子を示している．

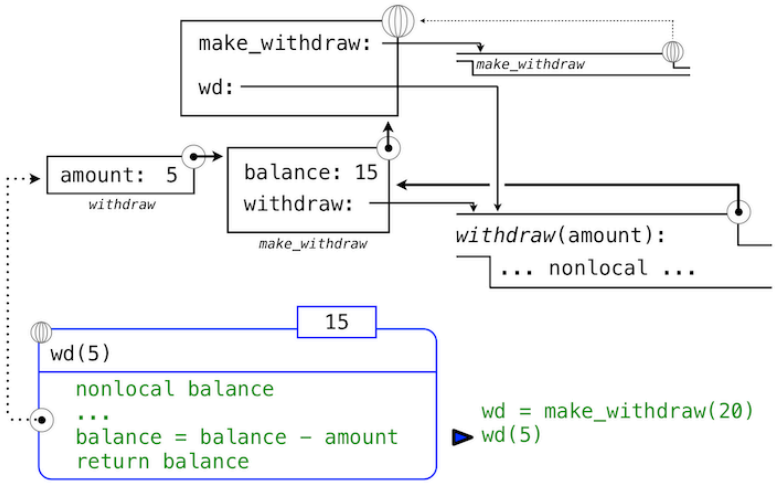
次に，wd (= *withdraw*) 関数を引数 5 で呼び出した際には

```
>>> wd(5)
15
```

[7]以下は図の出典も含めて http://wla.berkeley.edu/~cs61a/fa11/lectures/objects.html#object-oriented-programming による

となる．

このときの振る舞いを以下に示す．

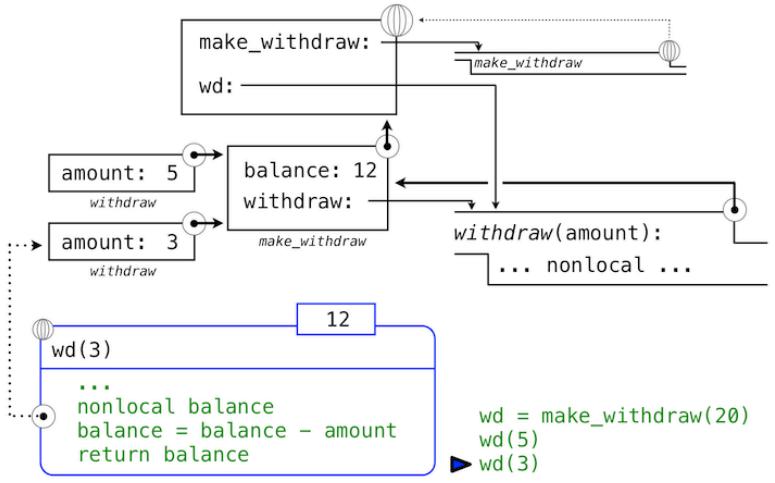


`wd(5)` を呼び出した状態

`wd` は *withdraw* 関数を束縛しているが，*withdraw* 関数を実行すると，その本文は *withdraw* 関数が束縛されていた環境を延長し新しい環境を作成し，実行する．

新しい環境のフレームでは仮引数 5 が `amount` 束縛されているが，`balance` に関する束縛は，この新しいフレームにはなく，親フレームを探し束縛を見つけている．このフレームにおいて `balance` を変更することで，*withdraw* 関数内の `balacne` の値も変更されている．すなわち，次に *withdraw* を呼ぶ際には `balance` の値は，20 でなく 15 となっている．

また，左下では `wd(5)` という式の評価がグローバルフレームで評価され，その本文は `amount:` を参照する局所フレームを参照している様子を示している．



`wd(3)` を呼び出した状態

以上をまとめると，手続き適用の環境モデルは以下の規則になる．

- 引数への手続きの適用は，まずフレームを新しく構築する．手続きの仮引数を呼び出し引数に束縛し，新しく構築した環境の文脈（新しく構築されたフレームから始まる列）の中で，

手続きの本体を評価する．新しく構築するフレームは，適用する手続きの環境を外側の環境として持つ

- 手続きは，与えられた環境で本文を評価しすることで作られる．結果としてつくられる手続きのオブジェクトは，本文のテキストと，手続きが作成された環境へポインタのペアになります．

3 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：抽象の壁, 隠蔽，無名関数，束縛，クロージャ，

1. accumulator は，一つの数値の引数で繰り返し呼び出される手続きで，呼び出される度に引数を変数 sum に足す．また，呼び出される度に，現在の sum の値を返すような挙動を示す．この関数を大域変数を利用する場合と利用しない場合で作成せよ．

```
>>> a = make_accumulator(5)
>>> a(10)
15
>>> a(10)
25
```

2. ある関数が何回呼ばれたかを数える機能を追加したい．引数に関数 `f` を取る手続き `make_monitored` を記述せよ．`make_monitored` が返す関数は `mf` と呼ばれる手続きで，この手続きは内部にカウンタを保持し，呼び出された回数を記憶する．`mf` の引数が `'how-many-calls?'` の時は，カウンタの値を返し，`'reset-count'` の場合はカウンタの値をゼロにする，それ以外の場合は引数を用いて `f` を呼び出だし，カウンタを 1 増やす．例えば以下のような挙動になる．

```
>>> import math
>>> sqrt = make_monitored(math.sqrt)
>>> sqrt(9)
3.0
>>> sqrt(100)
10.0
>> sqrt('how-many-calls?')
2
```

宿題１

accumulator は,一つの数値の引数で繰り返し呼び出される手続きで,呼び出される度に引数を 変数sum に足す.また,呼び出される度に,現在の変数 sum の値を返すような挙動を示す.この関数を大域変数を利用する場合と利用しない場合で作成せよ

Pythonにおいて、`accumulator`関数を定義します。まず、大域変数（グローバル変数）を利用するバージョンを作成します。次に、クロージャを利用したバージョンを作成します。クロージャは、外側の関数によって生成される内側の関数で、その内側の関数は外側の関数の変数にアクセスできます。これにより、大域変数を使用せずに状態を維持することが可能となります。

1. 大域変数を利用する場合:

python Copy code

```python
sum = 0

def accumulator(n):
    global sum
    sum += n
    return sum
```

このバージョンでは、大域変数`sum`を利用しています。この変数は関数の外部で定義され、関数内部からその値を変更するためには`global`キーワードを使用します。

2. 大域変数を利用しない場合（クロージャを利用）:

python Copy code

```python
def make_accumulator():
    sum = 0
    def accumulator(n):
        nonlocal sum
        sum += n
        return sum
    return accumulator
```

このバージョンでは、関数`make_accumulator`の内部で変数`sum`を定義し、内側の`accumulator`関数がこの変数を参 Regenerate response . ここでは`nonlocal`キーワ

宿題 2

ある関数が何回呼ばれたかを数える機能を追加したい.引数に関数 f を取る手続き make_monitored を記述せよ. make_monitored が返す関数は mf と呼ばれる手続きで,この手続きは内部にカウンタを保持し,呼び出された回数を記憶する. mf の引数が 'how-many-calls?' の時は,カウンタの値を返し, 'reset-count' の場合はカウンタの値をゼロにする,それ以外の場合は引数を用いて f を呼び出し,カウンタを 1 増やす.例えば以下のような挙動になる.

以下の Python コードは、あなたが指定した仕様に基づいて `make_monitored` 関数を実装しています。

python Copy code

```python
def make_monitored(f):
    count = 0

    def mf(*args):
        nonlocal count
        if args[0] == 'how-many-calls?':
            return count
        elif args[0] == 'reset-count':
            count = 0
            return count
        else:
            count += 1
            return f(*args)

    return mf
```

この `make_monitored` 関数は、他の関数 `f` を引数として受け取り、新しい関数 `mf` を返します。この新しい関数は、内部でカウンタ（`count`）を保持し、それが何回呼び出されたかを記録します。その後、`mf` が引数 `'how-many-calls?'` を受け取ると、カウンタの現在の値を返します。`'reset-count'` を受け取ると、カウンタをゼロにリセットします。それ以外の場合、引数を使用して関数 `f` を呼び出し、カウンタを 1 増やします。

A Emacs による例

A.1 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.1.1 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[8]．

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．

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

```
(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.1.2 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．

```
(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.1.3 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．

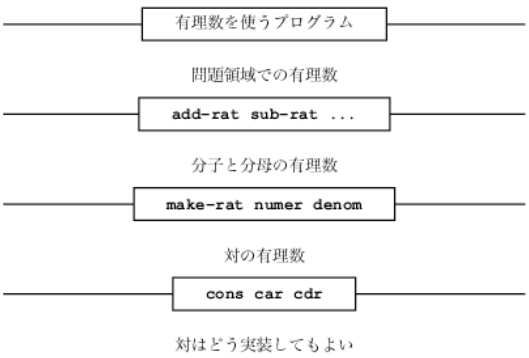
```
(defun add (x y)
  (cond ((consp x) (add-rat x y))
(t          (+ x y))))
add
(defun mul (x y)
  (cond ((consp x) (mul-rat x y))
(t          (* x y))))
```

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.1.4 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[9]．



[9]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