プログラミング入門
第 9 回「代入」

二宮 崇（ ninomiya@cs.ehime-u.ac.jp ）

教科書

Structure and Interpretation of Computer Programs, 2nd Edition: Harold Abelson, Gerald Jay Sussman, Julie Sussman, The MIT Press, 1996

第 3 章 標準部品化力、オブジェクトおよび状態

より大きなシステムの作成には、これまで習ってきた抽象化だけでは不十分で、システム全体をより系統だったモジュールに分割する必要があります。モジュールに分割されることで、分割して開発、保守ができるようになり、大きなシステムの開発、バグとりが可能となります。そのモジュール化の戦略として、オブジェクトに注目する方法と、ストリームに注目する方法があり、この章ではそれらを学んでいくことになります。オブジェクトは状態をもつため、計算の環境モデルが必要となり、今までのような置き換えモデルによる説明はできなくなります。

3.1 代入と局所状態

オブジェクトは SmallTalk などのオブジェクト指向言語と呼ばれるプログラミング言語で中心的に扱われる概念です。基本的には、各オブジェクトは固有のデータを持ち、オブジェクトに対し、命令(メッセージ)を与えることにより、オブジェクトはその状態を変えながら計算を行うモデルです。オブジェクトの状態は過去の命令の結果で変化し、同じ命令を与えても多くの場合は異なる結果が返ってきます。オブジェクトの状態という表現は抽象的ですが、過去に実行した命令の結果を集約し、新しい命令の結果に影響を与える変数を状態変数と呼び、オブジェクトの状態は状態変数で代表されると考えます。大きなシステム全体の状態はこういったたくさんのオブジェクトにより構成され、

オブジェクト間の相互作用により全体の計算が行われる、ということになります。まず局所状態変数と状態を変えるための代入について説明します。

3.1.1 局所状態変数

銀行口座の例

銀行口座に預金があって、withdraw という関数でお金を引き出せて、残額が返値として返ってくるということをやりたい、とします。残額は 100 からスタートします。

```
> (withdraw 25)
75
> (withdraw 25)
50
> (withdraw 60)
"Insufficient funds"
> (withdraw 15)
35
```

といったことができればよいとします。今までは状態というものがなかったので、関数に同じ引数を渡せば必ず同じ結果が返ってきましたが、上の例をみると、今までの手法では実現できそうにないことがわかります。この withdraw を実現するためには、残額を保持した状態変数が必要となります。次のコードが withdraw を実現します。

```
(define balance 100)
(define (withdraw amount)
  (if (>= balance amount)
      (begin
        (set! balance (- balance amount))
        balance)
```

```
      "Insufficient funds"))
```

ここで、新しく set!と begin がでてきました。set!は特殊形式で、

(set! ⟨*name*⟩ ⟨*new-value*⟩)

という構文になります。set!は⟨*name*⟩の値を⟨*new-value*⟩を評価した値に変更します。begin は

(begin ⟨*exp* 1⟩ ⟨*exp* 2⟩ … ⟨*exp k*⟩)

という構文になっていて、⟨*exp* 1⟩から⟨*exp k*⟩まで順に評価し、最後の式⟨*exp k*⟩の値を begin 全体の値として返します。

次に、大域変数に直接残高を記憶させるのではなく、もっとオブジェクト指向らしく口座オブジェクトなるものをつくるようにしてみましょう。口座を作る関数は次のように実装できます。関数を返すと関数内に出現した変数が記憶されたまま関数が返されたことを思い出しましょう。

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "Insufficient funds")))
```

これに対し、例えば、W1 と W2 の別々の口座を作れます。

```
> (define W1 (make-withdraw 100))
> (define W2 (make-withdraw 100))
> (W1 50)
50
```

```
> (W2 70)
30
> (W2 40)
"Insufficient funds"
```

このようなオブジェクトに対し、withdraw 以外の様々な機能が備わっていると大変便利です。次のように、オブジェクトを操作する関数を内部にもった関数を返すようにすれば実現することができます。

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (display "Unknown request -- MAKE-ACCOUNT")
                (display m))))
  dispatch)
```

記号と比較することでそのオブジェクトに対する関数を呼び出す dispatch という関数を内部で定義して、その dispatch を返します。すると次のようなことができるようになります。

```
> (define acc (make-account 100))
> ((acc 'withdraw) 50)
50
> ((acc 'withdraw) 60)
"Insufficient funds"
> ((acc 'deposit) 40)
90
> ((acc 'withdraw) 60)
30
```

このようにオブジェクトにメッセージを渡すことで計算を進めるプログラムの書き方のことをオブジェクト指向といいます。

手続き型プログラミング

set!は代入と呼ばれる操作になります。set!を用いると手続き型プログラミングという作法でプログラムを書くことができるようになります。まず、今まで何度か書いてきた 1 から n までの和を求めるプログラムについて考えてみましょう。関数型プログラミングでは、次のように書きました。

```
(define (sum n)
  (if (= n 1)
      1
      (+ n (sum (- n 1)))))
```

set!を使えば次のようにプログラムを書くこともできます。

```
(define (sum n)
  (define result 0)
  (define i 1)
  (define (iter)
    (if (<= i n)
        (begin
          (set! result (+ result i))
          (set! i (+ i 1))
          (iter))
        'done))
  (iter)
  result)
```

最初にresultに0を代入し、iに1を代入します。iterの中では、iの値を1つずつ増やしながら、resultにiの値を足すことを繰り返しています。手続きで書くと次のようになっています。←は代入操作を表します。

1. result ← 0
2. i ← 1
3. i ≤ n の間以下を繰り返す。i ≤ n でなければ7.に進む。
4. result ← result + i
5. i ← i + 1
6. 3.に戻る。

7. result を返す。

どうでしょう？関数型プログラミングと手続き型プログラミングのどちらがよりわかりやすいでしょうか。手続き型プログラミングのほうがわかりやすい、という人もいるだろうし、関数型プログラミングのほうがわかりやすい、という人もいるのではないかと思います。しかし、手続き型プログラムをもっとわかりやすいものにすることができます。次の for があるとします。

for

```
(define (for seq proc)
  (if (null? seq)
      'done
      (begin (proc (car seq))
             (for (cdr seq) proc))))
```

for はリスト seq と関数 proc を受け取って、seq の各要素を順に関数 proc に適用することを繰り返します。最後に'done を返していますが、これはダミーで実際にはこの返値は使いません。この for は map と似ているのですが、map の中の cons の代わりに begin があって、リストを返さずにただ seq の各要素に proc を適用することを繰り返します。また、引数の順序が逆になっています。(map は(map proc seq)で、for は(for seq proc)となっています)

for を使うと sum をもっと簡単な手続き型のプログラムにすることができます。

```
(define (sum n)
  (define result 0)
  (for (range 1 n)
       (lambda (i) (set! result (+ result i))))
  result)
```

だいぶ簡単になったような気がしませんか？最初に result に 0 を代入します。次に、range で 1 から n までのリスト、つまり、(1 2 3 … n)というリストを作ります。続いて、for でリストの各要素 i に対し、(set! result (+ result i))を実行します。そして、最後に result を返します。

関数型プログラミングと手続き型プログラミング。どちらがより使いやすかったでしょうか。最後に begin 以外に連続して式を評価する関数や書式を紹介します。

(define (f x) ⟨*exp* 1⟩ ⟨*exp* 2⟩ … ⟨*exp* *k*⟩)

(let ((a x)..(c z)) ⟨*exp* 1⟩ ⟨*exp* 2⟩ … ⟨*exp* *k*⟩)

(cond ((条件式) ⟨*exp* 1⟩ ⟨*exp* 2⟩ … ⟨*exp* *k*⟩)

((条件式) ⟨*exp* 1⟩ ⟨*exp* 2⟩ … ⟨*exp* *k*⟩)

…

(else ⟨*exp* 1⟩ ⟨*exp* 2⟩ … ⟨*exp* *k*⟩))

(lambda (x) ⟨*exp* 1⟩ ⟨*exp* 2⟩ … ⟨*exp* *k*⟩)

いずれも⟨*exp* 1⟩ ⟨*exp* 2⟩ …と順に評価していき最後の⟨*exp* *k*⟩を返値として返します。