# プログラミング入門
# 第 2 回「関数と置き換えモデル」

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

## プログラムの構造と解釈（第 1 章のつづき）

### 1.1.4 関数

関数も define を使って定義できます。次は 2 乗の計算をする関数の定義です。

```
(define (square x) (* x x))
```

この定義は、数式としては$square(x) = x \times x$という関数の定義を与えたのと同じことになります。x という引数（正確には仮引数もしくは仮パラメータと呼ばれる）をとって、(* x x)を計算する関数です。一般的には、

(define (⟨関数の名前⟩ ⟨引数の列⟩) ⟨本体⟩)

となります。値と同様にすでに定義された関数を用いて新しい関数を定義することもできます。

```
(define (sum-of-squares x y)
    (+ (square x) (square y)))
```

また、引数のない関数を定義することもできます。例えば、次のように定義できます。

```
(define (f) (+ 3 5 6))
```

このｆという関数に対し、(f)を評価すると 14 が返ってきます。このように、引数のない関数を評価することができます。

### 1.1.5 置き換えモデル

square、sum-of-squares と新しい関数ｆをみてみよう。

```
(define (square x) (* x x))

(define (sum-of-squares x y) (+ (square x) (square y)))

(define (f x) (sum-of-squares (+ x 1) (* x 2)))
```

ここで、(f 5)を評価するとします。すると、

```
(f 5)
```

は次のように置き換えられます。

```
(sum-of-squares (+ 5 1) (* 5 2))
```

ここで、(+ 5 1)と(* 5 2)を評価すると、それぞれ 6, 10 になるから、

```
(sum-of-squares 6 10)
```

となる。次にまた sum-of-squares を置き換えると、

```
(+ (square 6) (square 10))
```

になり、square の定義にでさらに置き換えると、

```
(+ (* 6 6) (* 10 10))
```

となり、(+ 36 100)を評価すると、136 という結果になります。まとめると次のように計算されることになります。

```
(f 5)
=(sum-of-squares (+ 5 1) (* 5 2))
=(sum-of-squares 6 10)
=(+ (square 6) (square 10))
=(+ (* 6 6) (* 10 10))
=(+ 36 100)
```

=136

このように置き換えて計算を進める計算モデルは「置き換えモデル」と呼ばれます。当たり前のようにもみえますが、これは実際の計算の方法とは関係なく、プログラムはこのように解釈される、というプログラムの意味を提供してくれています。この置き換えモデルは非常に単純なモデルですが（後にこれは通用しなくなることがわかりますが）、与えられたプログラムはこのような計算結果を返せば良い、という解釈の定義を明確に与えてくれています。このようなモデルがあるということは、良いモデル設計ができれば、例外なく正しい計算を安心してコンピュータにやってもらえる、ということです。実装によって異なる答えがでたり、複数の異なる解釈がありえたりすると、例外的に解釈しなくてはいけないケースが増えたり、何より安心してプログラムを書くことができないでしょう。また、このような明確な定義が与えられると同じ結果を与えるより効率的なコンパイラやインタプリターの実装を考えることができるようになります。

## 作用的順序と正規順序

関数型言語の面白いところの一つに、評価を行う手順が必ずしも固定された手順でなくてもかまわない、ということがあります。上記の 1.1.3 で説明した順序では、まず引数を評価し、それから関数を置き換える、という順で評価をしましたが、これだけが評価の方法ではありません。先に関数の置き換えを行って、引数部分の計算をあとにすることもできます。

上の(f 5)の計算に対し、先に関数部分の置き換えをしてから引数の評価をするようにしてみましょう。つまり、一番外側の関数から置き換えて、基本演算子がでてくるまで置き換えを繰り返します。次に、同様の手続きを引数に対して繰り返し行います。基本演算子の計算をすぐには行わず、関数を完全に置き換えてから(展開してから)、基本演算子の計算を行うことになります。

```
(f 5)
=(sum-of-squares (+ 5 1) (* 5 2))
=(+ (square (+ 5 1)) (square (* 5 2)))
=(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
=(+ (* 6 6) (* 10 10))
=(+ 36 100)
=136
```

同じ結果が得られることがわかります。この「完全に展開してから簡約する」という評価方法は正規順序の評価(normal-order evaluation)と呼ばれます。一方、「引数を評価し、作用させる」方法は作用的順序の評価(applicative-order evaluation)と呼ばれ、実際のコンピュータ上では、作用的順序の評価で計算されています。置き換えをつかってモデル化でき、正しい値が得られる手続きについては正規順序と作用的順序は同じ値になることが示されています。

### 1.1.8 ブラックボックス抽象としての関数

今までいくつかの関数を定義してきました。(square x)は x という値の二乗を返す関数で(define (square x) (* x x))で与えてきました。しかし、square は他にも、

```
(define (square x) (exp (double (log x))))

(define (double x) (+ x x))
```

として与えることもできるし、double は

```
(define (double x) (* 2 x))
```

として与えることもできます。プログラマは、square や double がどのような値を受け取って、どのような値を返すかということだけ知っていれば、square や double をつかったプログラムを書くことができて、square や double がどのように実装されているかはむしろどうでも良い、ということがわかります。これは関数による手続きの抽象化と呼ばれます。プログラマは全ての関数を自分自身で頑張って書かなくても、他のプログラマからブラックボックスとしての関数をもらってくれば良くて、それがどのように実装されているかは知らなくても良い、ということです。(今つかっているライブラリがどのように実装されているかを知ることは重要なことではありますが…)

#### 局所変数

関数の引数として定義される変数は、外部で定義される変数や他の関数で定義される変数の影響を受けません。例えば、次のようなプログラムがあるとします。

```
(define x 100)

(define (square x) (* x x))
```

```
(define (f x) (+ (square (* 3 x)) x))
```

このとき、大域環境の x と square の中の x と f の中の x は異なる x として処理されます。例えば、(f 5)を評価すると、f の中では x に 5 が代入され、(+ (square 15) 5)を評価します。続いて(square 15)を評価すると、square の中では x に 15 が代入され、(* 15 15)を評価します。つまり、大域環境の x には 100、f の中の x には 5、square の中の x には 15 が割り当てられることになり、異なる変数が用いられていることがわかります。このように関数の中でのみ有効になるように定義され、外部で定義された変数や他の関数で定義された変数の影響を受けない変数のことを局所変数と呼びます。また、このような変数は関数の定義の中で束縛されている、といいます。束縛されている変数は束縛変数とも呼ばれます。変数が束縛されていなければ、その変数は自由であると言われます。

変数が束縛されている範囲はスコープ(scope)と呼ばれます。関数定義の中の引数はその関数本体の中がスコープとなっています。上の例では関数 f の中で x が引数として束縛されていて、square は自由になっています。つまり、square は環境の中でその定義を探すことになります。一方、関数 square の中でも x が引数となっていますが、束縛されていて、外部の x に影響を与えることはありません。

局所変数の特徴として、関数の定義の中で一貫して同じ変数名が使われていればどんな変数名にしてもかまわない、という特徴があります。例えば、次の二つの関数の定義は同じになります。

```
(define (square x) (* x x))

(define (square y) (* y y))
```

これは数学における関数と同じで$f(x) = x^2$と定義しても、$f(y) = y^2$と定義しても同じ関数が定義されることと同様です。

プログラムをみてそのスコープを決定できることは静的スコープ(lexical scoping)と呼ばれます。