

付録 S Scheme 超簡易入門

Scheme は、Lisp の一方言である。Scheme は関数型言語であるが、Haskell と異なり、変数への代入など命令的な特徴を残している。また遅延評価ではなく、関数の引数を先に評価する、先行評価を採用している。

S.1 Scheme でのプログラミング

関数適用 関数適用 (function application) は次のような形である。

- (関数 引数₁ 引数₂ ... 引数_n) のような 丸括弧 (parenthesis) でくくった式の列

Scheme では + や × などの算術演算子に、通常の 中置記法 (infix notation) ではなく、前置記法 (prefix notation) を用いることが特徴的である。例えば、(+ 1 2) という式では、+ が関数 (function)、1 と 2 が引数である。

変数と代入 例えば、

```
(define x 5)
```

という式で、5 という値の入った “x” という名前の変数を用意する。これ以降は x という変数は 5 に評価される。

Scheme の場合、変数名の中には、アルファベット、数字の他に

```
+ - . * / < = > ! ? : $ % _ & ~ ^
```

などの記号を用いることができる。(もちろん空白はダメ) アルファベットの大文字と小文字は 区別しない。(つまり、Japan と japan は 同じ 変数である。)

また set! という命令によって、変数の値を変更する (代入するという) ことができる。(C 言語の 「=」 演算子に対応する。)

```
(set! x 4) ; 変数 x の値を 4 に変更する。  
; それ以前に x を define しておく必要がある。
```

これは、Scheme が 命令型言語 としての側面を持つことを示す。なお、Scheme では 「;」 から行末までがコメントである。

リスト リストを入力するためには、組み込み関数 `list` を用いる。`list` は任意の数の引数を取ることができる。

```
1 > (list 1997 5 6)
2 (1997 5 6)
3 > (list "kagawa" "university")
4 ("kagawa" "university")
```

単に `(1997 5 6)` と入力すると、Scheme の処理系は、`1997` という関数を `5` と `6` という引数に適用しているのだと判断してしまう。

このように、Scheme (一般に Lisp) では小括弧「(、)」が2つの意味に使われる。ユーザが入力するときは「[関数適用](#)」の意味に、処理系が出力するときは「[リスト](#)」の意味になる。もっと正確に言うとユーザが「リスト」を入力すると、処理系はそれを「関数適用」だと解釈するのである。このような処理系の振舞いは Lisp の強力さの源であるが、一方で混乱のもとでもある。

上記のデータは「'」(クォート記号・引用記号)を用いて次のようにも入力できる。

```
1 > '(1997 4 22)
2 (1997 4 22)
```

「' 式」は、「(quote 式)」とも書く。(むしろ、後者が正式な書き方である。)

```
1 > (quote (1997 5 6))
2 (1997 5 6)
```

ここで `quote` は、その [次の式を評価しない](#) ことを表す。だから、`(1997 5 6)` は関数適用ではなくリストと解釈される。

空リスト (要素を1つも含まないリスト) は `'()` または `(list)` のように入力する。

```
1 > '()
2 ()
3 > (list)
4 ()
```

また `cons` ([「こんす」と読む](#))、`car` ([「カー」と読む](#))、`cdr` ([「くだー」と読む](#)) などが、リストを操作するための最も基本的な関数である。ここで `cons` はリストを組み立てるための関数、`car` と `cdr` はリストを分解するための関数である。

cons — 第2引数として与えられるリストの先頭に、第1引数として与えられる要素を付け加えたリストを返す

car — リストの先頭の要素を返す

cdr — リストの先頭を除いた残り (のリスト) を返す

null? — リストが空ならば真、空でなければ偽を返す

関数定義 関数の定義には次の形式の define を用いる。

```
(define (関数名 変数1 ... 変数n) 定義)
```

変数₁ ... 変数_n はこの関数の仮引数である。

```
1 > (define (square x) (* x x))
2 square
3 > (square 4)
4 16
```

条件判断 条件判断は次のような形式で行なう。

```
(if 条件式 式1 式2)
```

条件式が **真なら式₁** を、**偽なら式₂** を評価 (計算) する。(C の if 文と異なり、値を返すことに注意する。むしろ、C の?:オペレータに対応する。)

逐次実行

```
(begin 式1 式2 ... 式n)
```

式₁ から、式_n を順に評価し、最後の式_n の値を全体の値として返す。通常、式₁ から、式_{n-1} は **副作用** のために実行される。C や JavaScript のブロック { ~ } と意味は似ているが、C や JavaScript のブロックは“文”の一種であるので値を持たないのに対し、Scheme の begin 式は値を持つ。

なお、関数の定義の本体で、

```
1 (define (hen_na_square x)
2   (begin (set! x (* x x))
3         x))
```

のように順に式を評価するときは、上のように begin を使う必要はなく、

```
1 (define (hen_na_square x)
2   (set! x (* x x))
3   x)
```

のように単に式を並べて書くだけで良い。(これを“暗黙”の begin という。)

局所変数 (let) 関数の定義の他に let という構文で局所変数を導入することができる。

```
(let ((変数1 式1)
     ...
     (変数m 式m))
  式0)
```

この let 文では、式₁ から式_m を評価した結果が、変数₁ から変数_m に入れられ、最後に式₀ を評価する。変数₁, ..., 変数_m のスコープは式₀ である。

ラムダ式 (匿名関数)

```
(lambda (変数1 ... 変数n) 定義)
```

これは変数₁ ... 変数_nを引数とする関数である。例えば、`(lambda (x) (* x x))`は2乗する関数である。`((lambda (x) (* x x)) 2)`は4になる。`lambda`はギリシャ文字の λ のことである。これらは`define`を用いて定義した名前付きの関数:

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

の`square`と同じ関数になる。つまり、`(define (square x) (* x x))`は`(define square (lambda (x) (* x x)))`と同じ意味なのである。

S.2 Scheme の call-with-current-continuation

Scheme では、プログラマが接続を直接操作することができる。このことをSchemeは[第1級の接続](#) (first-class continuation) を持つという言い方をするときもある。

```
(call-with-current-continuation thunk)
(call/cc thunk)
```

この`call-with-current-continuation`という名前は長いので、省略形の`call/cc`がよく使われる¹。

ここで`thunk`は1引数の関数であり、`(call/cc thunk)`は[現在の接続](#)を引数として、`thunk`を呼び出す。この`thunk`のなかで、この接続を呼び出せば、そのときの接続は無視されて (= ジャンプして)、`call/cc`が呼ばれたときの接続にその値が返される。また`thunk`が接続を呼び出さなければ、`thunk`自身の戻り値が`call/cc`式全体の戻り値になる。

例えば、

```
1 (define (bar x)
2   (call/cc (lambda (k)
3     (+ 100 (if (= x 0) 1 (k x))))))
```

という関数を考える。`(bar 0)`を評価すると普通に足し算が計算され、値は101になる。一方、`(bar 1)`の場合は、接続`k`が呼び出されるので100を足す部分はスキップされて、戻り値は1となる。

よくある`call/cc`の使い方は、`try ~ catch`と同じような大域脱出である。

¹Scheme は、`-`や/`/`のような文字も変数の名前の中で使用できるので、`call-with-current-continuation` や `call/cc` でひとつの名前である。ただし、`call/cc`はSchemeの標準仕様には含まれていないので、処理形によっては、

```
(define call/cc call-with-current-continuation)
```

のように自分で定義しておく必要がある。

```

1 (define (multlist xs)
2   (call/cc (lambda (k)
3     (define (aux xs)
4       (if (null? xs) 1
5           (if (= 0 (car xs))
6               (k 0)
7               (* (car xs) (aux (cdr xs)))))))
8
9   (aux xs))))

```

この関数はリストの要素の掛け算を求める。要素の中に0が見つかったら、大域脱出して multlist 全体の値は 0になる。

しかし、このような大域脱出だけならば、言語の仕様に call/cc のような大がかりな仕掛けをいれておく必要はない。本当の call/cc の価値はコルーチンなどの普通でない制御構造を実現できるところにある。

S.3 コルーチン (coroutine)

コルーチンとは、2 つ以上のプログラムの実行単位が、交互に制御を受け渡しながら実行されていく方式のことである。サブルーチン (subroutine) のように、実行単位の間には主と副といった従属関係はなく、コルーチンを構成する個々のルーチンは互いに対等な関係である。

例えば、

```

1 (define (increase n k)
2   (if (> n 10) '()
3       (begin (display "_i:") (display n)
4              (increase (+ n 1) (call/cc k)))))
5 (define (decrease n k)
6   (if (< n 0) '()
7       (begin (display "_d:") (display n)
8              (decrease (- n 1) (call/cc k)))))

```

という2つの関数を定義して

```
(increase 0 (lambda (k) (decrease 10 k)))
```

という式を実行すると、

[i:0 d:10 i:1 d:9 i:2 d:8... i:10 d:0](#)

というように画面へ出力される。2つの関数 increase と decrease が交互に実行されていることがわかる。

このように call/cc はひじょうに強力なプリミティブで、コルーチンの他にこれまでに紹介したエラー処理 (try ~ catch) や非決定性などのプリミティブも、call/cc を用いて定義できることがわかっている。ある意味でオールマイティなプリミティブである。

しかし、call/cc は効率的な実装の難しいプリミティブでもある。素直な実装では call/cc を実現するためには、スタック全体のコピーを行なう必要がある。一方、はじめからスタックをヒープの中に取り、スタックのコピーを行なわないという方式もある。この方式では不要になったスタック領域も ゴミ集め で回収する。

S.4 さらに詳しく知りたい人のために...

文献 [1] は Scheme の仕様書である。通常、略して R6RS と呼ばれる。中に call/cc の簡単な解説もある。

この章の参考文献

- [1] Richard Kelsey, William Clinger, and Jonathan Rees (Editors),
「Revised⁶ Report on the Algorithmic Language Scheme」
<http://www.r6rs.org/>