

第S章 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 _____ (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 _____ (infix notation) ではなく、_____ (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 _____。(つまり、Japan と japan は _____ 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 _____ 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「 」の意味に、処理系が出力するときは「 」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 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` (と読む)、`car` (と読む)、`cdr` (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 を、 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「?:」オペレーターに対応する。）

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 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の範囲は式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 first-class continuation (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 (Scheme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。)

ここで `thunk` は 1 引数の関数であり、`(call/cc thunk)` は first-class continuation 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、

