Chaitin先生のToy Lispで遊ぶ

■ Chaitin先生のToy Lisp

Gregory J. Chaitinは「Kolmogorov-Chaitin Complexity」の理論=「アルゴリズム的情報理論」の創始者として知られている。
Springerから出版された三部作[1]は、自らが作った簡易 Lispを使って、理論を実際にコンピュータを使って確める内容となっている。その簡易Lispインタプリタには Mathematica, C, Javaと三つの言語の版がある[2]。例えば、C版は1000行程度のプログラムである。コードを完全に追うことも、その気になればそれほど大変ではない。
以下ではこのChaitinの簡易Lispを紹介する。ただし、三部作の中で、このLispは二進テープを持つ汎用チューリング機械マシンのエミュレータとして使われるのであるが、そのための特殊な機能についてはここでは全く触れない。
さて、このLisp、自然数しか扱えない簡易 Lispであって、非常に簡単に作られている。しかし、そうであっても例えばC言語では簡単にはできないような芸当も見せてくれる。ただし、簡単に作られているために、 本格的な Lispである Scheme や Common Lispとは異なった振る舞いをする場合があり、注意する必要がある。
Chaitin Lisp の組み込み関数は引数の数が関数毎に定まっているので[3]、Lisp が演算に関して前置形式を採用しているのと相まって、組み込み関数に関しては括弧が不要となっている[4]。ただし、 自分で定義した関数を使う際には括弧が必要となる。

■ 簡単な計算

リスプの演算は前置形式

```
+ 2 * 7 - 12 5
expression  (+ 2 (* 7 (- 12 5)))
value       51
```

リスト： cons, cdr, car

リスプの基本データ構造はリスト
nil ：空リスト

```
nil         [<= 入力 ]
expression  nil
value       ()

= nil '()   [<= 入力 ]
expression  (= nil (' ()))
value       true
```

cons ：要素をリスト先頭に追加(⇒リストを構成)

```
cons 3 cons 2 cons 1 nil
expression  (cons 3 (cons 2 (cons 1 nil)))
value       (3 2 1)
```

car ： リスト先頭要素を取り出す

```
car '(3 2 1)
expression  (car (' (3 2 1)))
value       3
```

cdr ： リスト先頭要素を除いた残り

```
cdr '(3 2 1)
expression  (cdr (' (3 2 1)))
value       (2 1)
```

1. "The Limit of mathematics (1998)" , "The Unkowable (1999)", "Exploring Randomness (2001)"
2. Chitin氏のサイトから入手可能 (直接には http://www.cs.auckland.ac.nz/~chaitin/lisp.c)
3. 逆に言えば、定まった引数の関数しか組み込み関数として許されていない。
4. 括弧の開閉チェック機能を持たないエディタを使う際には非常にありがたい。

■ リスプ式、アトム、リスト

リスプ式はアトムとリストからなる。
式中に現れるシンボルや数はアトムである。
nil=空リスト はアトムかつリストとなる唯一の例外。

```
atom? '(a b c)
expression  (atom? (' (a b c)))
value       false

atom? '()
expression  (atom? (' ()))
value       true
```

アトムにその値として、何らかのリスプ式を関連付けることを「束縛」という。アトムは生成された段階では自分自身に束縛されている。アトムが評価されるとそのアトムに束縛されているリスプ式が返される。

```
a           [<= 入力 ]
expression  a
value       a
```

define を使って現在の計算環境において有効な大域的な束縛を作ることができる。
lambda を使って一つのリスプ式の中だけで有効な局所的な束縛を作ることができる。

```
define a 7 [<= 入力 ]
define      a
value       7
* a a      [<= 入力 ]
expression  (* a a)
value       49
('lambda (a) * a a  5)   [<= 入力 ]
expression  ((' (lambda (a) (* a a))) 5)
value       25
* a a      [<= 入力 ]
expression  (* a a)
value       49
```

上の('lambda (a) * a a 5)は、次のようにletを使って書くこともできる。letを使った式は**糖衣構文**で両者は全く同等である。(expressionを見よ。)

```
let a 5 * a a
expression  ((' (lambda (a) (* a a))) 5)
value 25
```

■ 関数と関数適用、式の評価

繰り返すと、アトムが評価されるとそのアトムに束縛されているリスプ式が返される。リストは**基本的に**次のように評価される。

```
( <func> <arg_1> <arg_2> ... <arg_n> )
```

リストの先頭要素が評価され関数と解釈される。リストの二番目以降の要素(=引数)がそれぞれ評価され、リスト第一要素の評価結果として得られた関数に適用される。
関数はlambda式によって作られる。次に示すのは、一変数xを引数とし、x^2をその値とする関数である。

```
lambda (x) * x x                    (1)
```

先の説明では

```
('lambda (x) * x x  5)
expression  ((' (lambda (x) (* x x))) 5)
value       25
```

の計算を、局所的に xに 5を束縛させ、その束縛下で$x \times x$を計算させる、とした。これを、「関数(1)に数5を**適用**する」と言う。
さて、関数(1)は名前がない**無名の関数**である。関数を名前で参照するには、関数(1)を値とするアトムを用意すればよい。 つまり、defineを使って

```
define square 'lambda (x) * x x
define      square
value       (lambda (x) (* x x))
```

とする。

```
(square 7)
expression  (square 7)
value       49
```

この計算例ではリストの評価則に従い、squareが評価されて関数(1)が返され、それに7が適用されて49 が得られている。

defineには関数定義のための次の形式の糖衣構文も用意されている。

```
define (square x) * x x
define      square
value       (lambda (x) (* x x))
```

これら二つ

```
    define (square x) * x x
    define square 'lambda (x) * x x
```

は全く同等の式である。

リスト評価の例外はリスト先頭要素が define, lambda, ', `, if の場合に限る。'はquoteと呼ばれ、一つの引数を取る。'の効果は、引数を評価せずにそのまま返すことである。

```
'+ 2 3
expression  (' (+ 2 3))
value       (+ 2 3)
```

■ 反復(再帰)計算

反復計算の典型例として$1+2+3+\cdots+n$ を計算してみよう。Lispでは、次の二通りの方法が典型的である。

```
[1. ]
define (sum1 n) if = n 0 0 + n (sum1 - n 1)
define      sum1
value       (lambda (n) (if (= n 0) 0 (+ n (sum (-
            n 1)))))
(sum 100)
expression  (sum1 100)
value       5050

[2. ]
define (sum2 n s) if = n 0 s (sum2 - n 1 + s n)
define      sum2
value       (lambda (n s) (if (= n 0) s (sum2 (- n
            1) (+ s n))))
(sum2 100 0)
expression  (sum2 100 0)
value       5050
```

リストを生成する例としてfibonacci数列を計算してみよう。

```
define (fibos n p1 p2)
  if = n 0
     nil
     cons p1 (fibos - n 1 p2 + p1 p2)
define      fibos
value       (lambda (n p1 p2) (if (= n 0) nil (cons
            p1 (fibos (- n 1) p2 (+ p1 p2)))))
(fibos 10 0 1)
expression  (fibos 10 0 1)
value       (0 1 1 2 3 5 8 13 21 34)
```

問題 1. (intgen 1 100)によって自然数のリスト (1 2 3 ... 100)を作る関数 (intgen m n)を定義せよ。

問題 2. (revers lst1 nil)により、与えられたリストlst1の要素を、逆順に並べて返す関数 (revers lst1 lst2)を定義せよ。

次の例は素因数分解を行う関数である。

```
define (prmf p n)
  if = n 1
     nil
     let q / n p let r remainder n p
       if = r 0
          cons p (prmf p q)
          (prmf + p 1 n)
define      prmf
value       (lambda (p n) (if (= n 1) nil ((' (lambd
            a (q) ((' (lambda (r) (if (= r 0) (cons
            p (prmf p q)) (prmf (+ p 1) n)))) (remai
            nder n p)))) (/ n p))))
[ 実行例 ]
(prmf 2 876543212)
expression  (prmf 2 876543212)
value       (2 2 29 2389 3163)
```

■ 引数に関数を取る例：リスト操作

関数を引数に取ることにより、柔軟性の高い(抽象度の高い)プログラムを書くことができる。そのような例としてリストを操作する一般的な関数を取り上げてみよう。

```
define (mapapp f proc init lst)
  if = lst nil
     init
     (proc (f car lst)
           (mapapp f proc init cdr lst))
```

例えば「与えられた数のリストの二乗和を取る関数」は上の関数を使って次のように書ける。mapapp の引数に fとして二乗を計算する無名関数、procとして和を取る基本関数 +、init(初期値)として0を与えればよい。

```
define (sqsum lst)
  (mapapp 'lambda (x) * x x "+ 0 lst)
[実行例]
(sqsum (intgen 1 1000))
expression  (sqsum (intgen 1 10000))
value       333383335000
```

■ 関数を返す関数

関数を返す関数も定義することができる。しかし、Scheme やCommon Lispのように格好良くはできない。

例えば一変数関数$f(x)$と$g(x)$の合成関数$g(f(x))=(g\circ f)(x)$を返す関数 (comp f g) は次のようになる。

```
define (comp f g)
  cons "lambda
  cons '(x)
  cons cons cons "' cons g nil
       cons cons cons "' cons f nil
            cons 'x
            nil
       nil
  nil
[実行例]
(comp f g)
expression  (comp (' f) (' g))
value       (lambda (x) ((' g) ((' f) x)))

((comp `lambda (x) ^ x 3
       `lambda (x) + * 2 x 1) 3)
expression  ((comp (' (lambda (x) (^ x 3))) (' (lamb
            da (x) (+ (* 2 x) 1)))) 3)
value       55
```

最後の例では $2x^3+1|_{x=3}$ を計算してる。
与えられた関数fをn回合成した関数$\underbrace{f \circ f \circ \cdots \circ f}_{n}$を返すLisp関数`(fpow n f)`を作ってみよう。実行例では$f(x)=x^2$を5回合成した関数を作り$x=2$での値を計算している。

```
define (fpow n f)
  if = n 0
     'lambda (x) x
     (comp f (fpow - n 1 f))
[実行例]
((fpow 5 'lambda (x) * x x) 2)
expression  ((fpow 5 (' (lambda (x) (* x x)))) 2)
value       4294967296
```

■ 関数の自己適用

最後にLispならではの計算を示しておこう。次の計算は100!を求めるものである。なぜ計算できるのか分かりますか?。

```
define F
  'lambda (f n) if = n 0 1 * n (f f - n 1)
expression  F
value       (lambda (f n) (if (= n 0) 1 (* n (f f (-
             n 1)))))
(F F 100)
expression  (F F 100)
value       933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```