ソフトウェア工学実験/計算機工学実験'94　課題2 (Lisp 言語) ＃3

久野　靖　(筑波大学大学院経営システム科学専攻)

1994.5.16,1994.6.15

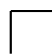
1　閑話休題 — cons の秘密について

さて、ここまで cons の第 2 引数は常にリストだという話で通してきた。しかし実はそうでなくてもよい。次の例を見て欲しい。

```
>(cons 'a 'b)
(A . B)
```

この . というのは何か? 実は、これまで Lisp に打ち込んできた式 (S 式、と呼ばれています) の内部構造は次の図のように「cons セル」とよばれる要素から成っている。各 cons セルは「下」と「右」の 2 方にたどることができる。実はこれが car と cdr である。リストというのは実はこれを n 個つないで、末尾に終りの

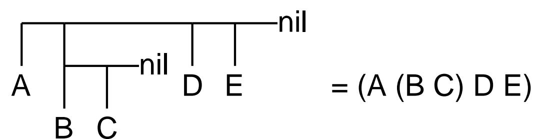
基本的なｃｏｎｓセル



リストの基本形

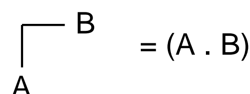
nil
A B C D
= (A B C D)

入れ子のリスト



末尾がｎｉｌでないリスト

E
A B C D
= (A B C D . E)



印 nil をつけたものだったわけである。そして、末尾が nil でないことも可能であり、その場合には . のあとにその末尾の要素を書くことで表す。より厳密に言えば次のどちらの書き方も正しい。

```
(A . nil)   ==  (A)
(A . (B . (C . nil)))  ==  (A B C D)
```

ただ、. を毎回つけていると煩わしいのも確かなので、普段はできるだけ . をつけない記法を使うのが一般的だということである。

2 命令型語としての Lisp

さて、ここまで Lisp をできるだけ関数型ないし適用型言語として使ってきた。でも Lisp は命令型言語としても使える。命令型言語に書かせない要素は (1) 変数と代入 (前回やった setq ですね!) と (2) ループである。ついでに、(3) 入力と出力もあった方がよい。

まず前回のやり方だとすべての変数はグローバル変数になってしまう。これがいやなら手続きに局所的なローカル変数を用意できる。また、変数を使うと「順番に実行」が欲しい。そのため defun の形式は実は次のようになっている。

```
(defun 関数名 (引数 ... &aux 局所変数名 ...)
   式 ...)
```

なお、本体の式はもちろん順番に実行される。関数の値は、最後の式の値である。

```
>(defun sisoku (x y &aux wa sa sho seki)
    (setq wa (+ x y)) (setq sa (- x y))
    (setq sho (/ x y)) (setq seki (* x y))
    (list wa sa sho seki))
SISOKU
>(sisoku 2 3)
(5 -1 2/3 6)
```

次に、ループを作るにはその名もズバリ

```
(loop 式 ...)
```

という式による。もちろん、式の並びが順番に繰り返し実行される。ループから出たければ

```
(return 式)
```

を実行する。この式がループ全体の値となる。(式を書かないと nil。) 例えばループを使ってべき乗を書くと次の通り。

```
>(defun power (x n &aux result)
    (setq result 1)
    (loop (if (< n 1) (return result))
          (setq result (* x result))
          (setq n (- n 1))))
POWER
>(power 2 3)
8
```

こっちの方がいいですか?

練習 1 リストの長さを数える関数 listlen をループを使って作れ。

さて、ループができると入力と出力がほしくなりますね? それは簡単で、

```
(read)     --- キーボードから S 式を読み込んで値として返す。
(print 式) --- 式の値を表示する。
```

なお、print の値は表示したのと同じ式です。だから

```
>(print '(a b c))
(A B C)  ← print が表示した
(A B C)  ← print の結果の表示
```

となります。

練習 2 数を入力するとその 2 乗を表示することを、0 が入力されるまで繰り返す関数 jijouloop を書け。

ぐっと Pascal ぽいでしょう? こっちの方がいいですか?

3 apply

さて、前回 listsum というのを作った時に何か疑問に思いませんでしたか? つまり、(+ 1 2 3 4) とかやれば合計が取れるのにそれをわざわざループで回りながら足し算するのは何かおかしい、というわけです。そんなことをしなくても「(1 2 3 4) というリストに+を適用して欲しい」と言えばすむはず。実はそのための関数 apply というのがある。その使い方は:

```
(apply 関数 引数のリスト)
```

これを使えば先の「合計」は次のような感じでできる。

```
>(apply #'+ '(1 2 3 4))
10
```

簡単でしょう? (関数の値を意味したい時には#' だったのは覚えていますね?) ところで、第 1 回に funcall というのをやりました。funcall の場合は引数を個別に書きますから

```
>(funcall '+ 1 2 3 4)
10
```

になるわけです。整理すると、引数が 1 つのリストになっている場合は apply、ばらばらで、個数が決まっているときには funcall を使うのが素直でしょう。

練習 2 数のリストを受けとり、その第 1 要素が負の数なら各要素の合計、そうでなければ全要素を掛け合わせた値を返す関数 sigmapi を書け。

4 eval

さて、様々なシステム関数のうちで一番不思議なやつをやりましょう。それは、与えられた式を評価する関数 eval である。一般に、単に Lisp に向かって

```
> x
```

と打ち込むとそれが評価されるのだった。また'x とするとそれはただの x に評価される。そして、eval はこの評価を強制的に行なわせる関数である。だから一般に

```
> (eval 'x)
```

と打ち込むと (x が何であろうと)

```
> x
```

と打ち込んだのと同じ効果がある。例えば次の練習問題を考えてみよ。

練習 3 変数名のリストを渡すと、それらの変数に入っている (setq されている) 値のリストを返す関数 valuelist を定義せよ。

5　Lispの処理系を自前で作る

5.1　トップレベルがやっていること

実は、Lispの処理系を自前で作るのはすごく簡単である。というのは、これまでやってきて分かるように、KCLがやっていることは>に向かってS式を打ち込むと、それを形式として評価し、結果を打ち出すことだけだからである。じゃあ早速自分で作ってみよう。

```
(defun xtoplevel (&aux e)
  (loop (princ "? ")
        (setq e (read))
        (cond ((eq e 'end) (return))
              (t (print (eval e)) (terpri)))))
```

見てのとおり、これは「?と打ち出し、S式を読み込み、それがendだったら終るが、そうでなければそれを評価して、結果を打ち出す(ついでに改行もする)」というだけである。動かしてみよう。

```
>(xtoplevel)
? (cons 'a 'b)
(A . B)

? (list 'a 'b 'c 'd 'e)
(A B C D E)

? end
NIL
>
```

ちゃんと、Lispの処理系みたいでしょう?

5.2　evalを自前で作るには

もうお分かりのように、「評価」の神秘はすべてevalの中にあるわけである。言い返れば、関数evalを自前で書ければそれでLispの処理系が書けたことにほとんどなるわけである。そこで以下では必要最小限の機能を持つ、しかしちゃんと動くevalを作ってみることにする。なお、本ものの evalと混同すると困るので、これから作る関数は全部xevalのように頭にxをつけておく。

5.2.1　記号の値

まず、記号を評価すると、その記号に束縛されている値が出てくる、というのを扱うことにしよう。そのためには、「どの記号には現在どんな値が束縛されているか」を覚えておく必要がある。そのため、

```
((変数名 . 値)(変数名 . 値) ...(変数名 . 値))
```

という形のリストを使うことにする。これをLispの世界では「連想リスト」と呼んでいる。例えば

```
((X . A)(Y . 1)(Z A B C))
```

だと、xはa、yは1そしてzは何でしょう?

で、次にやるべきことは、連想リストの中から求める変数の部分を見つけてくる関数を作ることである。実はこれは前回の演習問題で作ったxassocである。

```
(defun xassoc (x l)
  (cond ((null l) nil)
        ((eq (caar l) x) (car l))
        (t (xassoc x (cdr l)))))
```

動かしてみよう。

```
>(setq l '((x . a) (y . 1) (z a b c)))
((X . A) (Y . 1) (Z A B C))

>(xassoc 'x l)
(X . A)

>(xassoc 'z l)
(Z A B C)
```

では、これを使って xeval をちょっとだけ作って見よう。xeval が式を評価するには連想リストも渡してやらないといけないことに注意 (ほんものの eval はシステム内部の表を直接参照するので連想リストはいらない)。

```
(defun xeval (e a)
  (cond ((atom e) (cdr (xassoc e a)))
        (t 'error)))
```

ともかく、この xeval は今のところシンボルしか扱えないので、渡された S 式がアトムでなければ error というのを返す。さて、xtoplevel をこっちの eval を使うように直しておこう。

```
(defun xtoplevel (&aux e)
  (loop (princ "? ")
        (setq e (read))
        (cond ((eq e 'end) (return))
              (t (print (xeval e *env*))
                 (terpri)))))
```

なお、*env*というのは現在定義されている値を保持しておくために (勝手に) 作った変数である。広域変数を*...*と言う形にする、というのは規則ではないが、Lisp でよく使われる流儀である。さっそくこれで遊んでみよう。

```
>(setq *env* '((x . a) (y . 1) (z a b c)))
((X . A) (Y . 1) (Z A B C))

>(xtoplevel)
? x
A

? z
(A B C)
```

うんうん、よさそうだが...

```
? t
NIL

? (quote a)
ERROR
```

あれあれ、t が nil に評価されるのはちょっと困る。これは*env*に (t . t) を入れておくことで解決する。(nil . nil) も入れておこう。

5.2.2 quote

上で quote が使えないのは当然、だってまだ作っていないから。では作ろう。xeval を次のように直せばよい。

```
(defun xeval (e a)
  (cond ((atom e) (cdr (xassoc e a)))
        ((atom (car e))
         (cond ((eq (car e) 'quote) (cadr e))
               (t 'error)))
        (t 'error)))
```

つまり、e がアトムでなくてリストでも、その先頭がアトムであってなおかつ quote なら... e の cadr、というのは

```
(quote なんとか)
```

の「なんとか」の部分ですよね。を取り出せばよい。

```
>(setq *env* '((x . a) (y . 1) (z a b c)
               (t . t) (nil . nil)))
((X . A) (Y . 1) (Z A B C) (T . T) (NIL))

>(xtoplevel)
? t
T

? x
A

? (quote x)
X

? (quote (a b c))
(A B C)

? '(a b c)
(A B C)

? end
NIL
>
```

今度は大丈夫でしょう? え、「'」なんて実現していないって? それは、「'」は読み込む時に処理されて内部ではもう「(quote ...)」になってしまっているからなのでした。

5.2.3 apply

さて、いつまでも記号しか評価できないのではつまらない。関数が使えるようにしよう。関数は... apply で適用できるのでしたよね。そこで我々も自前の xapply を作り、xeval は次のように直す。

```
(defun xeval (e a)
  (cond ((atom e) (cdr (xassoc e a)))
        ((atom (car e))
         (cond ((eq (car e) 'quote) (cadr e))
               (t (xapply (car e)
                          (xevlis (cdr e) a) a))))
        (t (xapply (car e) (xevlis (cdr e) a) a))))
```

ずいぶん難しそうですって? 増えたのは最後の2行だが(つまりリストはquoteでない時はすべて関数呼び出しだから)、まずxevlisというのは式のリストをもらってその各要素を評価してまたリストにする。

```
(defun xevlis (l a)
  (cond ((null l) nil)
        (t (cons (xeval (car l) a)
                 (xevlis (cdr l) a)))))
```

ちょっと試してみましょうか。

```
> (xevlis '(x z y) *env*)
(A (A B C) 1)
```

いいでしょう? そして、xapplyは次の通り。

```
(defun xapply (f x a)
  (cond ((eq f 'car) (caar x))
        ((eq f 'cdr) (cdar x))
        ((eq f 'cons) (cons (car x) (cadr x)))
        ((eq f 'atom) (atom (car x)))
        ((eq f 'eq) (eq (car x) (cadr x)))
        (t 'error)))
```

例えば、適用したい関数がcarだったら、引数の並びxの第1要素((car x))のcarが結果なわけ。carを実現するのにcarを呼んだらずるいって? どこかから下はS式の操作を実現しないといけないので、これぱかりはしかたがない。その代わり、この「ずる」をやるのは5つの基本関数だけである。では実行。

```
>(xtoplevel)

? (car '(a b))
A

? (cons 'x '(y z))
(X Y Z)

? (cons 'x (cons 'y (cons 'z nil)))
(X Y Z)

? (eq 'x 'x)
T
```

5.2.4 lambda

ずいぶんLispらしくなってきたでしょう? しかしまだ、使える関数は基本関数だけである。それじゃいやだから、lambdaが使えるようにする。そのためにはまず、xpairlisという関数がいる。

```
(defun xpairlis (x y a)
  (cond ((null x) a)
        ((null y) a)
        (t (cons (cons (car x) (car y))
                 (xpairlis (cdr x) (cdr y) a)))))
```

これは連想リストの前に「a は 1 で b は 3 で...」というような情報を付け加えるために使う。次のような具合である。

```
>(xpairlis '(a b) '(1 3) *env*)
((A . 1) (B . 3) (X . A) (Y . 1) (Z A B C) (T . T) (NIL))
```

さて、これを利用して xapply を次のように直す。

```
(defun xapply (f x a)
  (cond ((eq f 'car) (caar x))
        ((eq f 'cdr) (cdar x))
        ((eq f 'cons) (cons (car x) (cadr x)))
        ((eq f 'atom) (atom (car x)))
        ((eq f 'eq) (eq (car x) (cadr x)))
        ((atom f) 'error)
        ((eq (car f) 'lambda)
         (xeval (caddr f) (xpairlis (cadr f) x a)))
        (t 'error)))
```

つまり、関数のところに (lambda... が来ていたら、その場合は仮引数部のリストと実引数部のリストを対応させて連想リストの頭に追加し、その状態で本体を評価すればよいわけである。もっと具体的な例で言おう。

```
(xapply '(lambda (x) (cons x x)) '(a))
```

であれば、これは連想リストの頭に「(x . a)」つまり a を評価したら値は y だよ、と書いた状態で「(cons x x)」を評価すればいいわけである。そうですね? では実行例。

```
>(xtoplevel)
? ((lambda (x) (cons (cons x nil) nil)) 'a)
((A))
```

確かにできている。

5.2.5 defun

ところで、やっぱりいきなり lambda と書くのではなく、名前をつけて呼びたいですよね? そこで、関数の名前とその本体も変数と同様に連想リストに登録することにして、xapply の「(atom f) 'error」の所を次のように直す。

```
((atom f) (xapply (cdr (xassoc f a)) x a))
```

つまり、連想リストから関数本体を探してきて、それを xapply するように直す。ついでに、xtoplevel も直して defun を入れてしまおう!

```
(defun xtoplevel (&aux e)
  (loop (princ "? ")
        (setq e (read))
        (cond ((eq e 'end) (return))
              ((and (listp e) (eq (car e) 'defun))
               (setq *env* (cons
                            (cons
                             (cadr e)
                             (cons 'lambda (cddr e)))
                           *env*)))
              (t (print (xeval e *env*)) (terpri)))))
```

つまり、S 式がリストでなおかつ先頭が defun だったら、

(関数名 lambda 引数部 本体)

というリストを連想リストに追加するわけである。では実行。

```
>(xtoplevel)
? (defun f (x y) (cons y x))
? (f 'aa 'bb)
(BB . AA)
```

確かにできている。

5.2.6 cond

これで完成? いや、1 つだけ忘れているものがある。まだ cond を作っていないから、条件判断が書けない! でもここまで来ればもう簡単。まず xeval を直して、quote に加えて cond も特別扱いにする。

```
(defun xeval (e a)
  (cond ((atom e) (cdr (xassoc e a)))
        ((atom (car e))
         (cond ((eq (car e) 'quote) (cadr e))
               ((eq (car e) 'cond)
                (xevcon (cdr e) a))
               (t (xapply (car e)
                          (xevlis (cdr e) a) a))))
        (t (xapply (car e)
                   (xevlis (cdr e) a) a))))
```

で、xevcon であるが。

```
(defun xevcon (l a)
  (cond ((null l) nil)
        ((xeval (caar l) a) (xeval (cadar l) a))
        (t (xevcon (cdr l) a))))
```

つまり、最初の条件を見て、それが真ならその本体が値。そうでなければ次の条件を見る... ために、自分自身を再帰的に呼ぶ。これで OK。いよいよ再帰関数でもなんでも書ける「ほんものの」Lisp ができた。

```
>(xtoplevel)
? (defun g (l)
      (cond (l (cons (cons (car l) (car l))
                     (g (cdr l))))
            (t nil)))
? (g '(a b c))
((A . A) (B . B) (C . C))

? end
NIL
```

なお、普通はリストの終りかどうかを調べるのに (null l) を使うのだけれど、null が定義されていないので判定を逆にしている。ともあれ、ちゃんと再帰関数が動くような xeval ができましたね!

いかがでしたか。Lisp の処理系というのはどうなっているか、結構自分で分かるようになったと思いません?

A 本日の練習問題兼出席

本日は、まず例題の追試と練習問題 1〜3 をやってください。(すぐ終ると思いますが。)

次に、「小さい Lisp」を打ち込みます[1]。ただし、一気に打ち込んでもつまらないので、資料に出てくる順にちょっとずつ増やしては試しに実行してみる。(最低限、資料に出ている例$+\alpha$くらいは試すこと。)

で、練習問題と打ち込めたところまでの実行例でいいですから「小さい Lisp」の実行例を打ち出し、裏に

- 学籍番号、氏名、所属、本日の日付。
- 以下のアンケートに対する答え (簡単でいいですよ)。

  Q1. Lisp のプログラムが書けそうになりましたか。ならないとすれば、どこに問題がありましたか。

  Q2. 本日やったことのうち面白かった/興味深かった部分はどこですか。また、難しいと思った部分はどこですか。

  Q3. 本日の感想、今後の要望などお書きください。

を記入したものを出席用に提出してください。レポート課題もありますから、まあ簡単に済ませてください。出席として認定されるためには、本日 5 時までに事務室のレポートボックスに提出のこと。

B Lisp 編の課題

さて、いよいよレポート課題です。自分の実力に応じて、以下の課題群から最低 3 つ以上選んでください。やさしいのも難しいのもありますが、自分の実力に応じているかどうかは過去 3 回の出席を見るとわかるので、不当に易しいものを選ばないように。(といって、無理に難しいのを選んで自爆するのも意味ありませんから。あくまで適切にね。)

まず、CommonLisp にはあるけど「小さい Lisp」に足りない機能を追加する、というのをいくつか考えてみました。

[1]打ち込ませる位ならコピーさせろ、という人もいると思うけれど、打ち込んでは動かしてみることで理解できるようになると思うのでよろしく。

課題 A: `if` が使えるようにする。できれば then のみのものと else まであるものの両方が可能だと嬉しい。

課題 B: `loop` が使えるようにする。なお、`return` が難しかったら、`loop` でなく `(while 条件 式)` などという構文をでっちあげて作ってもよい。

課題 C: `cond` の枝に複数の式が書けるようにする。

課題 D: 関数本体に複数の式が書けるようにする。

課題 E: `defun` の他に `setq` も使えるようにする。[2]

`if` や `loop` がちゃんと動いていることを「証明」するには、`print` が使えるようにしておくとよいと思います。ついでに次のはどうでしょう。

課題 F: 「小さい Lisp」には数値というものがない! 例えば「1」と打ち込むとエラーになるはず。それではつまらないので、数値をちゃんと扱い、四則演算と比較ができるようにする。[3]

課題 G: これら以外に、CommonLisp にもないようなオリジナルな (奇想天外な) 機能を考案して、使えるようする。

さて、ここから先は `lambda` 式の変形版です。じつはここに出てくるのはすべて様々な Lisp 処理系において採用されていたものばかりですので、せいぜい昔をしのんで (?) ください。

課題 H: `lambda` 式の不便なところは、与えられた引数を全部評価してしまうので、`if` のようなものが書けないという点である。そこで、引数を評価せずに受けとるというちょっと代わった版 (これを `nlambda` と名付ける) とこれを使った関数を定義できる `ndefun` を追加してみよ。

課題 I: `lambda` 式の不便なところは、引数の個数が固定していることである。だから`+`とか `list` のような関数が書けない。そこで、引数を評価はするけど個別に分解してしまわずに、1 個のリストとして受けとるというちょっと代わった版 (これを `llambda` と名付ける) とこれを使った関数を定義できる `ldefun` を追加してみよ。

課題 J: 上の 2 つをいっしょくたにして、引数を一切評価せず 1 個のリストとして受けとる版 (これを `flambda` と名付ける) とこれを使った関数を定義できる `fdefun` を追加してみよ。実はこれが歴史的にはいちばん古い。

課題 K: ちょっと別のアプローチとして、引数は評価せずに受け取り、形式を内部で組みたて、その組みたてた形式を評価する、というやり方 (これを `mlambda` と名付ける) とこれを使った関数を定義できる `defmacro` を追加してみよ。例えば次のようにするわけである。

```
(defmacro if (x y z)
   (list 'cond (list x y) (list 't z)))
```

これを

```
(if (null l) 'a 'b)
```

[2] 本ものの setq を実現するのは結構難しい (なぜか考えること!)。できる範囲でよい。

[3] これをやるためには、アトムが数値かどうか調べられないといけない。それは `(numberp アトム)` でできる。

のように呼び出すと、定義に従って一旦次のような形式が組み立てられる。

```
(cond ((null l) 'a) (t 'b))
```

そして、これが評価されると... 求めていた、if と同じ動作が実現できるわけである。

どの課題を選んだにしても、実現するだけでなく、それを使った例題を考えて実行例を掲載すること (そうしないとできてるかどうかわからない!)

なお、レポートの形式は次の通りとする。これを守らないと提出しても受理されないことがあるので、注意して欲しい。

- A4 版の用紙を縦に使うこと。
- ホチキス等でばらばらにならないよう閉じること。
- 必要にして十分な実行例を含めること。
- 内容構成は次の順とする。
  1. 表紙。レポートタイトル (ソフトウェア/計算機工学実験'94 Lisp 編レポート、ですね)、学籍番号、氏名、所属、提出日付のみを記述する。
  2. 課題内容。どんな課題をやったか。
  3. 方針。下敷きのプログラムをどう直そうと思ったか。また、自分固有の機能についてはなぜそういうのがいいと思ったか、どう実現しようと思ったか、など。
  4. 回答。実際にどういうプログラムにしたか。実行例。
  5. 考察。やってみて分かったこと。採点においては考察の内容が重視されます。手を抜かないように。また「感想」ばっかりでもだめよ!
  6. 付録。プログラムリストが長い場合には付録にしてください。

〆切は次回の実験の開始時刻まで。事務室のレポート箱に提出のこと。