プログラミング言語論2014 # 4 —
記号処理とLisp

久野　靖 *

2014.5.8

1　Lisp入門

1.1　Lisp言語の由来と特徴

Lisp言語はJohn McCarthyによって1958年に考案された、世界で2番目の(1番目はFORTRAN)高水準言語であり、ラムダ計算にヒントを得て「少数の基本構造だけでプログラムを組み立てる」ことが基盤にありました。また、リスト処理、記号処理などの考え方を取り入れることで(これらはそれ以前にあったIPLと呼ばれる言語から来ているらしいです)、柔軟なデータの扱いを得意とし、人工知能分野で広く使われるようになりました。

McCarthyのLisp (Lisp-1、Lisp-1.5)の成功により、多くのLisp処理系が作られるようになりましたが、それぞれに独自の拡張を行ったため、多様な言語仕様を持つ「Lisp属」が形成されました。これらの流派は最終的に統合されてCommonLispとなりましたが、CommonLispとは別の方向性をめざしたLisp属の言語は現在でも多数あります(Scheme、Clojureなど)。

Lispの特徴としては、次のものがあげられます。

- プログラムとデータをともに「S式」と呼ばれる共通の記法で表現している。これにより、プログラムを生成し実行するプログラムが容易に記述できる。
- S式は記号(symbol)と呼ばれるデータ要素とその並びを中心として構成されており、他の手続き型言語のような面倒なデータ構造定義によらずに柔軟かつ動的なデータを扱うことができる。
- ごみ集め(garbage collection)を標準として備えた最初の言語であり、これによりメモリ管理に煩わされることなしに動的なデータを扱うことができる。

ここでは、フリーのCommonLisp実装であるCLisp処理系を用いて簡単にLisp入門をしたあと、Lisp処理系の基本原理がどのようなものであるかを理解するための題材として、「Lispで書いたLisp」を取り上げて行きます。

1.2　CLispのREPL

Lisp処理系は伝統的に、「Read-Eval-Print Loop」(REPL)の形で実行環境が提供されています。これはつまり、「S式を読み込み、それを評価(計算)し、結果を印字し、また最初に戻る」ということです。Clispで見てみましょう。

```
% clisp
...
[1]> (+ 1 2 3)          ←S式を入力
```

*経営システム科学専攻

```
6
[2]> (* (+ 1 2 3) 5.5) ←別の S 式を入力
33.0
[3]> (setq x 100)       ←変数 x を設定
100
[4]> x                  ← x を参照
100
[5] (/ x 1000)          ← S 式の中で x を参照
;;;;; どうなると思います?
[6]> y                  ←未定義の変数参照
;;;;; y は未定義だというエラーと選択肢
ABORT :R3     ABORT
Break 1 [7]> :R3        ←中止してトップに戻るという選択
[8]> (bye)              ←終わるという関数
Bye.
%
```

先に書いたように、Lisp の入力は「S 式」ですが、これは次のように定義されると思ってください (とりあえず)。

S 式 ::= リスト | アトム
リスト ::= (S 式…)
アトム ::= 記号 | 数値 | 文字列 | …

S 式をプログラムとして実行 (評価) するときは、次のようになります。

- 数値は、その数値、文字列はその文字列に評価される。
- 記号は、その記号が表す変数の値に評価される。
- リストは、その先頭は記号であり、関数名として扱われる、残りの部分はそれぞれ評価された後、関数の引数として渡される。

実はリストにはこの規則に従わない別のものもあるのですが、それはおいおい説明します。とにかく大事なのは、C などの言語では「`func(1, 2)`」のように書いていたのが Lisp ではすべて「`(func 1 2)`」のようにかっこを外に書く、カンマは使わず空白で区切る、ということです。慣れるまでちょっととまどうかも知れません。

あと、記号のうちでも `nil` と `t` は特別な意味を持っていて、それぞれ「偽/リストの終わり/空リスト」「真」を意味しています。ですからこれらを勝手な変数名として使うべきでありません。[1]

1.3 関数定義

Lisp のプログラムを作るというのはだいたい、必要な関数を定義していくことに相当します。関数は次の形になっています。

`(defun` 関数名 (引数…) 本体…)

これも全部 S 式の形になっていることに注意。なので、引数もすべて空白で区切ります。本体の式を複数並べた場合は順番に実行して最後の値が関数値として返されます。が、だいたいは本体の式は 1 個だけです。なお、この defun というのは S 式ではありますが関数ではないですね。このような特別な機能を持つ S 式を「特殊形式 (special form)」と呼びます。

もう 1 つ、if 文に相当する特殊形式 cond を紹介してしまいましょう。

[1] ただし条件判定のときには `nil` 以外の値はすべて「真」として扱われます — C で 0 以外は真なのと同様。

```
(cond (式 1 本体…)
      (式 2 本体…)
      …
      (t 本体…))
```

これは、まず式 1 を評価し、それが真なら対応する本体を実行し、結果を返します。そうでなければ、式 2 を実行し、それが真なら対応する本体を実行し、結果を返します…のように続き、最後は t なので、どれでもなければ最後の本体を実行して結果を返します。つまり if-elsif の連鎖になっているわけです。[2]

では、絶対値と階乗の関数を書いてみましょう。直接打ち込むのだと、間違いがあったときに直せませんから、エディタで書いてファイルに入れるのがよいと思います。

```
(defun absolute (x)
  (cond ((< x 0) (- x))
        (t x)))

(defun factorial (n)
  (cond ((< n 1) 1)
        (t (* n (factorial (- n 1))))))
```

これらをたとえば test1.lisp に書いておいたとして、CLisp に読み込ませ実行するのは次のようにします。

```
[1]> (load "test1.lisp")
;; Loading...
;; Loaded...
T
[2]> (absolute -3)
3
[3]> (factorial 5)
120
```

以下の演習で必要な関数を整理しておきます。

- 四則演算と剰余 — +, -, *, /, mod (-は単項の符合反転も)
- 数値の比較 — >, <, >=, <=、=、/=

演習 1 CLisp を起動し、上の例題群をそのまま実行してみよ。エラーも出してみること。階乗でちょっと大きな値も試してみること (中断したければ Ctrl-C)。できたら、次のような関数も定義して実行してみよ。

a. 円の半径を与えると、面積を返す。
b. 2 次元の 2 点の座標 x1、y1、x2、y2 を与えると、距離を返す。
c. 整数を与えると、奇数か偶数かに応じて「”even”」「”odd”」を返す。

1.4 S 式と記号処理

さて、ここまでは例題として数値の計算ばかりやってきたが、いよいよ Lisp の特徴である記号処理に進むことにします。そのためには、S 式をデータとして扱う必要がありますが、そのために quote という特殊形式から始めます (’というのは打ち込むのを楽にするための略記法です)。

[2]なお、2 分岐の if「(if 式 式 式)」もありますが、今回は後で実装してみる都合上、cond を優先して使います。

(quote S 式)　　;;; または　'S 式

こう書くと、「そのままの S 式」を指定することができます (これがないと、S 式が関数として実行されてしまうのでしたね)。次に、S 式を操作するための「5 つの基本関数」を覚えて頂きます。

- (eq X Y) — X と Y がともに記号のとき、それらが同じ記号なら t、そうでなければ nil を返す。
- (null L) — リストが空 (=nil) のとき t、そうでなければ nil を返す。
- (car L) — リストが空でないとき、その先頭要素を返す。
- (cdr L) — リストが空でないとき、その先頭を除いた残りのリストを返す。
- (cons X L) — リスト L の頭に要素 X を追加したリストを返す。

実際に例を見てみましょう。

```
[1]> (eq 'a 'a)
T
[2]> (eq 'a 'b)
NIL
[3]> (null '())  ←空のリストは
T
[4]> (null nil)  ←nil と同じ
T
[5]> (null '(a b))
NIL
[6]> (car '(a b c))
A
[7]> (cdr '(a b c))
(B C)
[8]> (cons '(a b) '(c d e))
((A B) C D E)
```

最後のが意外に思えるかも知れませんが、(c d e) というリストの先頭に「(a b)」というリストを 1 要素として追加するので、こうなるのです。ここで基本 5 関数とは別なのですが、便利に使える関数を 2 つ紹介しておきます。

- (append L M) — 2 つのリスト L と M を連結したリストを返す。
- (list A B C …) — すべての引数を並べたリストを返す。

後者は quote でいいのではと思った人、演習をやってから言うように。

あと、car と cdr は連続して使うことが多いので、(car (cdr X)) は (cadr X)、(car (car (cdr X))) は (caadr X) のように「合成した」関数が「a」「d」の並び 4 個まですべて用意されています。

演習 2 まず、(setq w '(a b (c d e) f)) を実行してください。そのあと、quote は使わず、基本 5 関数だけで次のものを取り出してみてください。

a. A

b. (C D E)

c. NIL

d. (E F)

演習 3 まず、(setq x '(a b c))、(setq y '(d e)) を実行してください。そのあと、基本 5 関数と append と list だけで次のものを生成してみてください。

a. (A B C D E)

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

c. (A B C (D E))

d. (A B C A D E)

1.5 リスト処理の再帰関数

リスト処理に慣れたところで、こんどは再帰関数を使ってリストを処理する流儀を見て頂きます。その前に、条件判定に必要な述語をいくつか追加しておきます。

- (atom X) — X がアトム (数値か記号) なら t。
- (numberp X) — X が数値なら t。
- (symbolp X) — X が記号なら t。
- (listp X) — X がリスト (空リストを含む) なら t。
- (consp X) — X がリスト (空リスト以外) なら t。

では基本的な例として、リストの長さを求める関数を作ってみます。

```
(defun listlen (l)
  (cond ((null l) 0)
        (t (+ 1 (listlen (cdr l))))))
```

考え方は次のようになります。このようにリストを先頭からはがしていく再帰は基本なのでよく味わってください。

- リストが空リストなら、長さは 0。
- それ以外なら、長さはリストの先頭を除いた残りのリストの長さ+1。

実行例を示します。トレースしてみると、徐々にリストを短くしつつ計算していることがよく分かります。

```
[2]> (listlen '(a b c d))
4
[3]> (trace listlen) ← トレースモードにする
...
;; Tracing function LISTLEN.
(LISTLEN)
[4]> (listlen '(a b c d))
1. Trace: (LISTLEN '(A B C D))
2. Trace: (LISTLEN '(B C D))
3. Trace: (LISTLEN '(C D))
4. Trace: (LISTLEN '(D))
5. Trace: (LISTLEN 'NIL)
5. Trace: LISTLEN ==> 0
4. Trace: LISTLEN ==> 1
3. Trace: LISTLEN ==> 2
```

```
2. Trace: LISTLEN ==> 3
1. Trace: LISTLEN ==> 4
4
[5]> (untrace listlen) ← トレース解除
(LISTLEN)
```

演習 4 次のような処理をする再帰関数を書け。既に定義した関数を下請けにしてもよいことに注意。

a. 数値のリストを渡すとその数値の合計を返す関数 listsum。例: `(listsum '(1 2 3 4))` → 10。

b. リスト中に含まれているすべてのアトムの数を返す関数 countatoms。例: `(countatoms '(a (b c) d (e)))` → 5。

c. 記号とリストを受け取り、リスト中からその記号をすべて除去したリストを返す関数 remove。例:`(remove 'e '(e v e n t))` → `(V N T)`。

d. リストとリストを受け取り、2 番目のリストの中から 1 番目のリストに含まれている記号をすべて除去したリストを返す関数 remset。例: `(remset '(a e i o u) '(n a g a s a k i))` → `(N G S K)`。

e. 2 つの長さが同じリストを受け取り、その各要素の対を長さ 2 のリストとして並べたリストを返す関数 pair。例: `(pair '(a b c) '(x y z))` → `((A X) (B Y) (C Z))`。

f. 2 つのリストを受け取り、両者を連結したリストを返す関数 concat。実はこれは append と同じもの (当然、append を利用してはいけない)。

2 Lisp の秘密 (?)

2.1 cons の内幕

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ました。しかし実はそうでなくてもよいのでして、次の例を見てください。

```
[1]> (cons 'a 'b)
(A . B)
```

この . というのは何でしょう? 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 1 のように「cons セル」とよばれる要素から成っています。各 cons セルは「下」と「右」の 2 方にたどることができます。実はこれが car と cdr なのです。

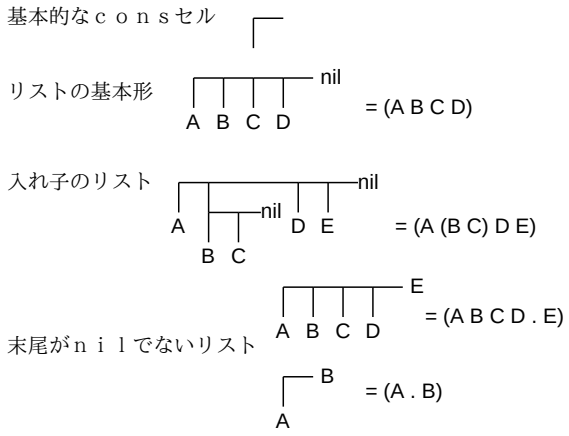

図 1: cons セルの概念図

リストというのは実はこれを n 個つないで、末尾に終りの印 nil をつけたものだったわけです。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します。より厳密に言えば、次のどちらの書き方も正しいわけです。

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

ただし、. 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す。

2.2 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ました。でも Lisp は命令型言語としても使えます。命令型言語に書かせない要素は (1) 変数と代入 (`setq`) と (2) ループで、ついでに、(3) 入力と出力もあった方がよいですね。

まず、いきなり `setq` を使うとすべての変数はグローバル変数になってしまいます。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まする。また、変数を使うと「順番に実行」が欲しいわけですが。そのため `defun` の形式は実は次のようになっています。

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

ここで「局所変数」のところは単に使いたい変数名を書いてもよいですし、「(変数名 式)」という形を書くことで初期値を与えることもできます (初期値を与えなければ `nil` が初期値)。

本体の式ももちろん順番に実行されます。そして関数の値は、最後の式の値です。

```
[1]> (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
[2]> (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 という特殊形式を使います。

```
(loop 式 ...)
```

もちろん、式の並びが順番に繰り返し実行されます。ループから出たければ「`(return 式)`」を実行するとループを脱出し、この式がループ全体の値となります (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ようになります。。

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

演習 5 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。

演習 6 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 C や Java ぽいでしょう? こっちの方がいいですか?

2.3 関数適用

さて、先に 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」と言えばすむはずで、実はそのための関数 apply というのがあります。その使い方は次の通り。

```
(apply 関数 引数のリスト)
```

ここで、CommonLisp では関数を値として書いたり渡したりするときは次の形式を使うことになっています。

```
(function 関数名)     ;;; または   #' 関数名
(function (lambda (引数…) 本体…)     ;;; または   #'(lambda ...)
```

これを使えば「合計」は次のような感じでできます。

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

なお、関数だけは値を指定したいけれど、引数はその場に普通に書きたい、という場合には類似品の funcall を使います。

```
(funcall 関数 引数 引数 …)
```

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

2.4 局所変数と let

ところで、先に関数定義時に引数部分の末尾に&aux で区切って局所変数を指定できる、という話をしましたが、局所変数は setq を使わないで関数型で書く場合も有用です。ただし、その場合はどちらかというと関数の中である程度計算したところで初期値つきで局所変数を導入することが多いでしょう。その場合には let という特殊形式が使われます (むしろこちらの方が歴史的に古く、多く使われる)。

```
(let ((変数名1 式1) (変数名2 式2) …) 式…)
```

ですが、この let は実は lambda を使って作られています。つまり、上の式は次のものと同等です。ただ、式の順番がこれだとごちゃごちゃで分かりにくいので、普通は let が使われるわけです。

```
(funcall #'(lambda (変数名1 変数名2 …) 式…) 式1 式2 …)
```

2.5 mapcar によるループ

ここまで、繰り返しはひたすら再帰によるループを使っていましたが、もうちょっと高水準な繰り返しがあって、多くつかわれています。それは、関数 f とリスト $(x_1 x_2 \cdots x_n)$ を与えられて、f を各要素に適用する関数 mapcar です。

```
(mapcar 関数 リスト)
```

実際にやってみましょう。

```
[1]>  (mapcar #'(lambda (x) (cons x x)) '(a b c d))
((A . A) (B . B) (C . C) (D . D))
```

演習 10 mapcar と同じものを (名前が衝突するとよくないので)xmapcar という名前で作れ。

演習 11 mapcar を利用して次の関数を作れ。

a. 数値と数値から成るリストとを受け取り、リストの各要素を最初の数値だけ増やす関数 addlist。例: (addlist 3 '(1 2 3)) → (4 5 6)。
b. 数値のリストを受け取り、その合計を返す関数 listsum2。もちろん mapcar を使うのだから再帰は使わないこと。(ヒント: setq は使う必要があるでしょう。)

2.6 記号処理の例: 数式の微分

ところで、記号処理って何のこと、と思われているかも知れませんね。その例として、「数式 (といっても Lisp なので S 式で表した奴) と変数の記号を与え、数式を与えた変数について微分する」という例題を考えてみます。式としては変数、数値、+、*だけが使えるものとし、また乗算の場合はカンタンのため被乗数の個数は 2 と決めます。

```
(defun diff (exp var)
  (cond ((null exp) nil)
 ((numberp exp) 0)
 ((symbolp exp) (if (eq exp var) 1 0))
 ((eq (car exp) '+)
  (cons '+ (mapcar #'(lambda (x) (diff x var)) (cdr exp))))
 ((eq (car exp) '*)
  (list '+
        (list '* (diff (cadr exp) var) (caddr exp))
        (list '* (cadr exp) (diff (caddr exp) var))))))
```

考え方は次の通りです。

- 空リストは微分しても空リスト。数値は微分したら 0。記号は指定した変数と同じなら 1、そうでなければ 0。
- 足し算の場合は、各項を微分して足せばよい。
- 乗算の場合は、2 項と決めたので、「$(fg)' = f'g + fg'$」による。

動かしてみましょう。

```
[1]>  (diff '(+ (* x x) (* 2 x) 1) 'x)
(+ (+ (* 1 X) (* X 1)) (+ (* 0 X) (* 2 1)) 0)
[2]> (diff '(* (* x x) (* x x)) 'x)
(+ (* (+ (* 1 X) (* X 1)) (* X X)) (* (* X X) (+ (* 1 X) (* X 1))))
```

確かにできていますが、無駄な項や未整理の項が沢山あります。まあ、微分はすると言ったけど綺麗にするとは言っていないので。

2.7 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 です。一般に、単に Lisp に向かって

```
> x
```

と打ち込むとそれが「評価」されるのでしたね。また'x とするとそれはただの x に「評価」されます。そして、eval はこの「評価」を強制的に行なわせる関数です。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ります。例えば次の練習問題を考えてみてください。

演習 7 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。

3 Lisp の処理系を自前で作る

3.1 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す。というのは、これまでやってきて分かるように、CLisp 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す。じゃあ早速自分で作ってみましょう。

```
(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)」というだけです。動かしてみます。

```
>(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?

3.2 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 の中にあるわけです。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す。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します。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 をつけています。

3.2.1 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ます。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ります。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います。これを Lisp の世界では「連想リスト」と呼びます。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?

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す。実はこれは先の演習問題で作った xassoc です。

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

動かしてみます。

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

[2]> (xassoc 'x l)
(X . A)

[3]> (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みます。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不要)。

```
(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 というのを返します。さて、xtoplevel をこっちの eval を使うように直しておきます。

```
(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) 作った変数です。広域変数を*...*と言う形にする、というのは規則ではありませんが、Lisp でよく使われる流儀です。さっそくこれで遊んでみましょう。

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

[2]> (xtoplevel)
? x
A

? z
(A B C)
```

うんうん、よさそうですが...

```
? 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りますね。これは*env*に (t . t) を入れておくことで解決します。(nil . nil) も入れておきましょう。

3.2.2 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。では作りましょう。xeval を次のように直せばよいのです。

```
(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。を取り出せばよいわけです。

```
[1]> (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))

[2]> (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。

3.2.3 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ので。関数が使えるようにしましょう。関数は... apply で適用できるのでしたね。そこで我々も自前の xapply を作り、xeval は次のように直します。

```
(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か? 増えたのは最後の 2 行ですが (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します。

```
(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 が結果なわけです。car を実現するのに car を呼んだらずるい、ですか? 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、こればかりはしかたがありません。その代わり、この「ずる」をやるのは 5 つの基本関数だけです。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
```

3.2.4 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す。それじゃいやだから、lambda が使えるようにします。そのためにはまず、xpairlis という関数が必要です。

```
(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います。次のような具合です。

```
[1]> (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 を次のように直します。

```
(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す。もっと具体的な例で言いましょう。

```
(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 x を評価したら値は a だよ、と書いた状態で「(cons x x)」を評価すればいいわけです。いいですね? では実行例。

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

確かにできています。

3.2.5 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します。

```
((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 を入れてしまいましょう!

```
(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す。では実行。

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

確かにできています。

3.2.6 cond

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

```
(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 は次のようになります。

```
(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 です。いよいよ再帰関数でもなんでも書ける「ほんものの」Lisp ができました。

```
>(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ます。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か?

演習 8 上の「小さい Lisp」処理系 (全部で 50 行くらい) を打ち込んで動かせ。少しずつ動作確認しながら動かさないと間違いがあったときにどこが間違っているか分からないので注意。

演習 9 CommonLisp にはあるけど「小さい Lisp」に足りない次の機能からいくつか選んで追加せよ。

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。

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。

C: cond の枝に複数の式が書けるようにする。

D: 関数本体に複数の式が書けるようにする。

E: defun の他に setq も使えるようにする。[3]

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。

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。[4]

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。

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。

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。

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。

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)
```

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!)

[3] 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。

[4] 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。