

Emacs Lisp てほどき

— .emacs を書くための —

目次

1	はじめに	1
2	*scratch*バッファで	2
3	基本的な S 式	3
4	シンボル	7
5	バッファローカルな変数と暗黙値	9
6	関数とコマンド	11
7	ラムダ式とフック	13
8	おわりに	15
A	Emacs Lisp のコーディング書式	16

1 はじめに

Emacs は、起動時、(典型的には) ホームディレクトリに `.emacs` という名前のファイルがあるか調べ、あれば、それにもとづいて様々な初期設定を行う。ファイル `.emacs` のテキストは、一見したところ括弧が目立つ独特のプログラミング言語で書かれているが、実は、この言語こそが Emacs を操る手段にほかならない。

そもそも、Emacs にある様々な編集機能は、ユーザがかなりの程度まで自由に制御できるように作られている。ほとんどの機能はユーザが Emacs それ自身の中でプログラムでき、その潜在能力はテキストエディタとして最低限求められるものをはるかに越えて強力である。M-x hanoi で起動される河内の塔のデモンストレーションはその一端に過ぎない。このような一般的プログラミング能力を備えるため、Emacs は、Emacs Lisp と称する一つのプログラミング言語処理系を内蔵している。そして、普段使われる編集機能のほとんど大部分は、この Emacs Lisp で記述されている。

実際のところ、Emacs とは、Emacs Lisp のコンパイラとインタープリタを備え Emacs Lisp を仮想的な機械語とする仮想的な操作システムであって、たまたまその機能がテキストエディタとしての利用に便利であるようにできている、と見ることも可能である。初期設定ファイル `.emacs` がこの Emacs Lisp で書かれていることは言うまでもない。

Emacs がどのように Emacs Lisp に基づいて働いているかについて、たとえば、a キーを押すと編集バッファ上に a の文字が書き込まれ、C-b (Control を押しながらの b キー) を押すと編集ポイントが左に 1 文字戻ることを、例として取り上げてみよう。これを Emacs は大略、以下のような仕組みで実行する。

1. 押されたキーを受けとる
2. そのキーと組になっている Emacs Lisp 関数を探す
3. その Emacs Lisp 関数を呼び出す
4. Emacs Lisp 関数がひと働きする

通常、a キーと組になっている Emacs Lisp 関数は `self-insert-command` である。この関数はその名のとおり、(最後に押された) キーの文字それ自身を (バッファ上の編集ポイントの箇所に) 書き入れる働きをする。一方、C-b と組になっている Emacs Lisp 関数は `backward-char` である。これもまたその名のとおり、(編集ポイントを) 逆向きに文字単位で移動させる働きをする。

ここで、前述の M-x hanoi をはじめとして Emacs には、M-x (Alt を押しながら、または Escape を押しからの x キーの押下) に引き続きコマンド名を入力することによって、そのコマンドを実行する方法があることを思い出そう。実は、そこで入力するコマンド名とは、Emacs Lisp 関数の名前にほかならない。言い換えれば M-x は Emacs Lisp 関数を名前呼び出す方法の一つにほかならない。したがって、C-b を押すかわりに、関数名を使って

```
M-x backward-char RET
```

と入力しても、編集ポイントが逆向きに移動する。

このように Emacs というシステムは Emacs Lisp を基礎としている。以下の説明が目指すところは、Emacs Lisp で書かれた `.emacs` などの Emacs 用ファイルの内容の基本的な意味を理解し、それによって、Emacs をより便利に使うための糸口をつかむとともに、プログラミングの本質を高次に理解するために Lisp の世界に一步だけ踏み込むことである。

2 *scratch*バッファで

ファイルを指定せずに Emacs を起動した場合、*scratch* という名前のバッファが画面に現れる。(C-x C-w, つまり Control を押しながら x キーを押してから, Control を押しながら w キーを押す, などとしない限り) このバッファには, 対応するファイルというものはなく, 単なる落書きに使うこともできる。しかしここでは, そのバッファのモードに注目しよう。Lisp Interaction, つまり Lisp 対話モードになっているはずである。普段, Emacs Lisp は Emacs の中で黒子として働いており, 人間の目の前には, あまり表だって現れてはこない。しかし, このモードでは Emacs Lisp に対し, 人間が直接, 次々と命令を下すことができる。したがって, Emacs Lisp がどのようなものかについて大体の感じをつかむために, このモードを利用することができる。

このモードで LFD(Line Feed, つまり C-j) を打つと, カーソルの前にある, Emacs Lisp で書かれた式が, 一つ計算され結果が表示される。以下に示すのは, 二つの数, 1 と 2 の和を計算する例である。

```
(+ 1 2) LFD
3
```

その名から推察できるとおり, Emacs Lisp は, Lisp というプログラミング言語の方言の一つである。そして Lisp では, 上記の (+ 1 2) をはじめ, 計算の対象となる式を, S 式 (S-Expression) と呼ぶ。‘S’ は Symbol (記号) の略である。その語源として, かつて Lisp プログラムをまず手書き用の比較的分かりやすいメタ式 (Meta 式) で書き, それから括弧にあふれた記号式 (Symbol 式) に直して計算機に読み込ませていた, という故事が伝えられている。また, 式を計算することは「式を評価する」(evaluate the expression) と言い表す方が, より Lisp の伝統にかなっている。

これ以降, 場所を節約するため, S 式 (+ 1 2) を評価すると 3 になることを

```
(+ 1 2) 3
```

と書く。

3 基本的な S 式

Emacs Lisp では数や文字列も一つの S 式である。数や文字列はそれ自身へと評価される。つまり、数や文字列を評価しても、結果はやはり同じ数や文字列になる。

```
7          7
"Copland"  "Copland"
```

文字列内部において、制御文字などを表現するには、\ で始まる拡張列を用いる。

```
C-h, BS    \b
C-i, TAB    \t
C-j, LFD    \n
C-k         \v
C-l         \f
C-m, RET    \r
C-[, ESC    \e
"          \"
\          \\  

```

制御文字を一般的に表現する拡張列も下記のとおり用意されている。下記において m に替えて @ から _ まで可能である。

```
C-m        \C-m または \^m
M-m        \M-m
C-M-m      \C-\M-m
```

疑問符 ? ではじまる文字コードは整数として取り扱われる。制御文字などの表記法は文字列の中の拡張列に準ずる。

```
?A          65
?a          97
?\r         13
?\C-m       13
?\^m        13
```

真理値も自分自身へと評価される。真は t、偽は nil である (英語では、nil は無あるいは零を意味する名詞である。null はその形容詞形である)。

```
t           t
nil         nil
```

任意個の S 式を空白やタブ、改行などで区切り、小括弧、つまり (と) で囲んだものも S 式の一つであり、リスト (list) と呼ぶ。Lisp の語源が LISt Processing にあると伝えられていることから分かる通り、リストの処理は Lisp が最も得意とする分野である。

一般に Lisp では、リストを使って関数呼び出しを表現する。例えば

```
(+ 1 2) 3
```

がその例である。+ は引数の和を結果とする関数であり、1 と 2 は引数である。そして

```
(+ 1 2)
```

は、それら三つの要素を含んだ一つの「リスト」である。Lisp はリストに出会うと、その第 1 要素 (この場合 +) を関数名、第 2 要素以降 (この場合 1 と 2) をその引数として関数を呼び出す。つまり、第 1 要素の関数を、第 2 要素以降の引数へ適用 (apply) する。Lisp では、このようなリストによる関数呼び出しを組み合わせて、まとまったプログラムを作り上げる。そして、そのことが emacs で見られるような括弧だらけの特異な様相をつくり出しているわけである。

リストによる関数適用の唯一の例外は、要素が一つもない空のリストである。この場合、呼び出すべき関数がないため、エラー扱いにすることもできたはずであった。しかし、伝統的な Lisp は、そう設計されていない。伝統的な Lisp と同様、Emacs Lisp は 空リスト () を nil と全く同じものとして取り扱う。したがって、S 式として () と nil を区別することは全く不可能である (単語本来の意味から言えば、空リストを nil とするのは素直な考え方である。むしろ、nil を真理値の偽として扱うことの方が派生的な用法と言える。この原理主義に基づく比較的若い有力な Lisp 方言も存在する)。

下記で +, -, *, /, % は、それぞれ数の加減乗除および剰余の関数を表している。最後の例は、リストの要素にまたリストを使った組合せの例である。

```
(+ 1 1) 2
(- 1 2) -1
(* 2 3) 6
(/ 10 2) 5
(% 10 3) 1
(+ (* 2 3) (/ 10 2)) 11
```

数の大小を比較する関数には =, /=, <, >=, <=, > などがあり、真ならば t, 偽ならば nil を結果の値とする (/= は を意味する)。

```
(/= 2 2) nil
(< 2 3) t
```

文字列を扱う関数として特に有用なものとしては、format がある。これは、C 言語ライブラリ関数の sprintf とほぼ同等の働きをする関数である。

```
(format "%d は 16 進数で %x" 20 20) "20 は 16 進数で 14"
(format "%c のコードは %d" ?A ?A) "A のコードは 65"
(format "%s 県" "愛知") "愛知県"
```

C 言語の sprintf と異なり、結果の文字列を入れておく場所をあらかじめ確保しなくてもよい点に注目しよう。Lisp では、一般に、記憶場所が必要なときには、それが自動的に用意され、不要になれば自動的に回収される。この特徴により、テキストエディタなど可変長データを取り扱うプログラムの記述が容易になっているわけである。

リストを扱う関数として代表的なものには、list, cons がある。list は各引数 (を評価した結果) を各要素とするリストを作り出す関数である。

(list 1 2 3)	(1 2 3)
(list "御河国" "穂国" "遠江国")	("御河国" "穂国" "遠江国")
(list (- 1 2) 3 (= 4 4))	(-1 3 t)
(list (list 1 2) (list 3 4))	((1 2) (3 4))
(list)	nil

上記の最後の例については、空リスト () と nil が全く同じものであることを思い出そう。

cons (コンス) は、CONSTRUCT(組み立てる、構築する) の略といわれており、第1引数 (の評価結果) を第2引数 (の評価結果) のリストの先頭に追加する関数である。

(cons 1 (list 2 3))	(1 2 3)
---------------------	---------

下記のように考えれば、list を使わなくても、cons だけでどんなリストも得られることに注意しよう (実を言えば、Emacs など Lisp 処理系内部で新しくリストを組み立てるときは、実際に cons ないしそれに相当するプログラムを使って下記のように順々にリストを「組み立てて」いるのである)。

(cons 3 nil)	(3)
(cons 2 (cons 3 nil))	(2 3)
(cons 1 (cons 2 (cons 3 nil)))	(1 2 3)

伝統的 Lisp には cons のほかに四つの基本関数 car, cdr, eq, atom がある。

(car (list 1 2 3))	1
(cdr (list 1 2 3))	(2 3)
(eq nil nil)	t
(eq 1 2)	nil
(atom 1)	t
(atom nil)	t
(atom (list 1 2 3))	nil

car (カー) と cdr (クダー) は、cons の逆の働きをする。cons は二つの引数をペアにするが、car と cdr はそのペアを分解する。名前は、Lisp が最初に作られたマシンの機械語で、たまたまペアを表すのに使われたレジスタの第1の部分を CAR、第2の部分を CDR と呼んでいたことに由来すると言われている。car と cdr は、それぞれそのペアの CAR と CDR を取り出す関数というわけである。エラーを起こさない任意の S 式 x, y について、(car (cons x y)) = x と (cdr (cons x y)) = y が成り立つ。

eq は二つの引数が同じ実体を表しているとき、そのときに限り真である。cons で別々に作られたリストは、たとえ同じ要素から出来ていても、ペアとしては別々だから、eq では偽 (nil) になることに気をつけよう。

atom は、引数が cons で作られたペアでないとき、そのときに限り真である。これ以上 car や cdr で分解できない、という意味での「原子」かどうか判定するわけである。

list と同じく組合せで説明できるが、よく使われるものに null がある。(null x) は (eq x nil) と定義できる。

(null nil)	t
(null 1)	nil
(null (list 1 2 3))	nil

Emacs Lisp には C 言語や Java の条件式 ? : にあたる関数も用意されている。if である。第 1 引数を評価した結果が非 nil ならば第 2 引数を、nil ならば第 3 引数 (およびそれ以降) を評価する。

```
(if t 1 2) 1
(if nil 1 2) 2
(if (< 2 3) "2 は 3 より小さい" "2 < 3 は嘘")
      "2 は 3 より小さい"
```

if のような関数は、必ずしも引数を評価するとは限らない特別な組込み関数であり、特殊形式 (special form) として区別されている。それに対し、普通の関数では、C 言語や Java の関数呼び出しと同様、引数をすべて評価してから、その関数を呼び出す。

あまりにも何気なく使われるため、レッキとした特殊形式の一つであるとさえ意識されないほど、よく使われる特殊形式として、quote が挙げられる。その働きは (驚くべきことに) 全く評価を働かないという働きである (このような働きは普通の関数には全く不可能であることに注意しよう)。

```
(quote 1) 1
(quote (+ 1 2)) (+ 1 2)
(quote (4 4 4)) (4 4 4)
```

数や文字列に quote を使うことには、ほとんど意味がない。リスト (+ 1 2) など、そのまま書いたのでは評価がひき起こされる S 式を、評価せずにもとのかたちのまま取り扱うために使ってこそ意味がある。一般に Lisp は、その使用頻度に鑑み、quote を単なる単一引用符で代用できるように設計されている。

```
'1 1
'+ 1 2 (+ 1 2)
'(4 4 4) (4 4 4)
```

ところで、ここで述べた + や quote は、あの hanoi などと同様、Emacs Lisp 関数ではあるが、

```
M-x + RET
```

として呼び出すことはできない。これは、+ や quote が「コマンド」として定義されておらず、コマンドとして定義されていない関数は、M-x で呼び出すことも、キーと組にしてキー押下で呼び出すこともできないからである。Emacs でこれらの関数は、間接的にコマンドの中で使われたり、.emacs など Emacs Lisp プログラムで使われる。それとは逆に、コマンドとして定義されている関数を *scratch* バッファから

```
(hanoi 5) LFD
```

のように打ち込んで呼び出すことは可能である (ただし、backward-char など編集ポイントの移動を伴うコマンドについては、評価すべき S 式を読み取り、結果として出力すべき S 式を書き込むそのバッファの上で、同時に編集ポイントの移動をすることになるため、一見不可解なエラーが発生することがある)。

4 シンボル

S 式には、数や文字列やリストなどのほか、記号、つまりシンボル (symbol) と呼ばれるものがある。Emacs Lisp にはそのほかにもベクトル、バッファ、プロセスなどがあるが、シンボルは、Lisp がプログラミング言語として体をなす上で基本的な役割を演じている点で、別格の存在と言える (そもそも S 式の S とは symbol の s であり、最初期の、Lisp が数も文字列も備えていなかった時代にすら、リストとシンボルだけはあったといわれている)。

シンボルは、ほかの様々な言語での変数名、関数名に相当する存在である。今まで見てきた関数名 + や if は、正確には、関数名として使われているシンボルである。英字で始まる英数字列でなければならない等の、よくある制限は、伝統的な Lisp にも Emacs Lisp にもない。空白文字や改行、括弧や引用符などを含まない限り、そして他に解釈のしようがない限り、ほとんどどのような名前でもシンボルとして使うことができる。以下は皆、シンボルである。

```
a b1 Emacs this-is-a-symbol + 012H
```

シンボルは S 式の一つであり、したがって、ほかの S 式と同様、どのようなシンボル についても

が成り立つことに注意しよう。

シンボルの役割は、第一に、値を格納しておくいれもの、つまり変数 (variable) として働くことである (関数としての働きについては後の節で述べる)。実際、シンボルを評価すると、その変数としての値が結果として得られる。変数値をセットするには特殊形式 setq を用いる。setq の第 1 引数はシンボル、第 2 引数はそれに代入すべき変数値である。

```
'a          a
a          [変数値未定義エラー]
(setq a 1)  1 [ただしこのとき a の変数値も 1]
a          1
(+ a 2)    3
(setq b (+ a 2)) 3 [ただしこのとき b の変数値も 3]
b          3
(setq a "Gershwin") "Gershwin" [ただしこのとき a の変数値も "Gershwin"]
a          "Gershwin"
b          3
```

ここで setq の第 1 引数のシンボル名は、評価されずにシンボルのまま処理されていることに注目しよう (もし評価されたとしたら、例えば最初の (setq a 1) に対して、a の変数値が未定義である、というエラーが起きたであろう)。これからも分かるとおり、set する際に (第 1 引数を) quote しておくということが、setq という風変わりな名前の語源である (容易に推理できるとおり、quote をしないただの set という関数も存在する)。

setq は、その評価結果の値よりも、そのときの一種の副作用として、変数値がセットされるという効果が重要である。このように副作用をもたらすことを主な仕事とする関数は、コマンドはもちろんのこととして、ほかにも Emacs Lisp には数多くある。

例えば、画面最下行のエコー領域にメッセージを、C 言語標準ライブラリ関数の printf に準じた書式で表示する関数 message がそうである。その結果の値は文字列であり、その限りでは関数 format と同様だが、ここでは、むしろ、エコー領域にそれが表示されることが重要である。

```
a "Gershwin"
```

```
(format "Good afternoon, Mr. %s" a)
      "Good afternoon, Mr. Gershwin"
```

```
(message "Good afternoon, Mr. %s" a)
      "Good afternoon, Mr. Gershwin"
      [ただしこのとき Good afternoon, Mr. Gershwin とエコーされる]
```

さてここではシンボル a と b を変数として使ったが、そのことによって Emacs の振舞いが変化した、ということは特になかったはずである。しかし、Emacs には、その振舞いを決める様々な変数があり、そういった変数をセットし直すと Emacs の振舞いもそれに応じて変わっていく。以下に挙げるのは、そのような変数の一例である。

scroll-step

ポイントがバッファの表示されていない部分に動いた時に自動的にスクロールされる行数を、指定する。ただし値が 0 のときはポイントのある行が中央になるようにスクロールされる。暗黙値は 0 である。

load-path

Emacs Lisp のライブラリをロードするときに検索されるパス名のリスト。

ファイル .emacs を使えば、これらの変数を Emacs 起動時に自動的に設定できる。Emacs は起動されると、まず .emacs を読み込み、その内容を、Emacs Lisp の S 式が列をなしているものと解釈して、先頭から次々に評価していく。したがって、もしその中に

```
(setq scroll-step 1)
```

があれば、それも S 式として評価されて、変数 scroll-step の値が 1 になる。

ところで、一つの setq で複数の変数をセットすることも実は可能である。したがって、たとえば

```
(setq a "Ferde")
      b "Grofe")
```

の代わりに

```
(setq a "Ferde"
      b "Grofe")
```

としてもよい。

5 バッファローカルな変数と暗黙値

ある変数の値は、Emacs のどのバッファから見ても同じなのが普通である。しかし例外も存在する。バッファローカル (buffer-local) な変数には、それが通用しない。バッファローカル変数の代表例としては下記の変数がある。これらの変数を `setq` で変更しても、その効果はそのバッファだけにしか及ばない。

`indent-tabs-mode`

`nil` 以外なら、行の字下げに空白とタブを使う。`nil` なら、空白だけを使う。暗黙値は `t` である。

`fill-column`

詰め込む (各行の右の桁をそろえる) ときの、行の最大幅を指定する。暗黙値は 70 である。詰め込みコマンドには、段落詰め込みの `M-q` (`M-x fill-paragraph`)、自動詰め込みモード入/切の `M-x auto-fill-mode` などがある。

`case-fold-search`

`nil` 以外なら、探索するとき、アルファベットの大文字と小文字を区別しない。`nil` なら区別する。暗黙値は `t` である。探索コマンドには `C-s` (`M-x isearch-forward`) `C-r` (`M-x isearch-backward`) などがある。

変数 `fill-column` を例にとって説明しよう。`fill-column` をセットするには、普通、セットしたい桁まで編集ポイントを移動させてから、

```
C-u C-x f (または C-u M-x set-fill-column)
```

を打つ。するとコマンドの関数 `set-fill-column` の中で、おおよそ以下のような `S` 式が評価される (実際はもう少しだけ複雑である。標準でインストールされる Emacs ライブラリの `lisp/simple.el` を参照)。

```
(setq arg (current-column))  
(message "Fill-column set to %d (was %d)" arg fill-column)  
(setq fill-column arg)
```

ここで関数 `current-column` は、その時の編集ポイントの桁位置を結果とする関数である。その値が `setq` によって `arg` にセットされ、従来の `fill-column` とともに `message` によってエコー領域に表示される。それから、`arg` が `fill-column` にセットされる。

ところで、これでセットされた値はそのバッファでしか有効ではない。つまり、ほかのバッファの `fill-column` は変更されない。しかし、この仕組みのおかげで、複数のファイル (のそれぞれに対応する複数のバッファ) の同時編集が、便利になっていることに気を付けよう。もしこういった仕組みがなければ、それぞれのファイルのテキストの桁幅が別々のとき、あるファイル (に対応するあるバッファ) から別のファイル (に対応する別のバッファ) へ移るたびに `fill-column` をセットし直さなければならなくなっただろう。

ここで、fill-column のようなバッファローカル変数の値をファイル .emacs で setq を使ってセットしても、ただか、Emacs 起動時のバッファ *scratch* にしか効果がないことに注意しよう。なぜなら最初に選択されるバッファは*scratch* であり、ほかのバッファでの値までは変更できないからである。そのため、.emacs の中でバッファローカル変数の値をセットするには、特殊形式 setq-default を用いる必要がある。この特殊形式はバッファローカル変数の暗黙値をセットする働きをする。setq-default でセットした値は、(個々のバッファでセットされ直さない限り) 全バッファの変数に対する値となる。

```
(setq-default fill-column 72)      fill-column  
〔ただしこのとき fill-column の暗黙値も 72〕
```

バッファローカルでない変数に対しては、setq-default の効果は setq と同じである。したがって、ある変数がバッファローカルであるかどうか分からないとき、.emacs の中で値を設定するには、setq-default を使えば十分である。

なお、付け加えて言えば、ある変数を fill-column のようなバッファローカル変数として印付けたいときは、関数 make-variable-buffer-local を用いる。この関数を下記のように適用すると、引数の評価値 (ただしシンボルであること) が、バッファローカル変数として取り扱われるようになる。

```
(make-variable-buffer-local 's)    s  
〔ただしこれ以降 s はバッファローカル変数になる〕
```

6 関数とコマンド

シンボルは、変数を表す名前としてだけでなく、関数を表す名前としても使われる。関数を表すものとして使われているシンボルとしては、

```
hanoi self-insert-command backward-char + - * / % =  
/= < >= > <= format list cons if quote setq
```

などがあることを学習した。極めて単純化して言えば、Emacs Lisp のシンボルは、quote されていない限り、

```
リストの第 1 要素ならば 関数 [例: (a 1 2)]  
それ以外ならば 変数 [例: (+ a 3)]
```

として扱われる。伝統に従い、変数としての値と、関数としての値は、Emacs Lisp では全く別であり、シンボルを、変数と関数の二つの値を持つ二面相にすることも可能である (そうでない Lisp もある)。たとえば

```
(setq + 3)
```

として + の (関数値ではなく) 変数値として 3 を与えることも Emacs Lisp では可能である (しかし、紛らわしさを避けるため、このようなことは十分な理由がない限り、あえてすべきではない)。ちなみにこのとき

```
(+ + +)
```

6

Emacs の中で働いている Emacs Lisp の関数は、その本体が C 言語で書かれていて機械語になっているプリミティブ (primitive) と呼ばれる関数と、Emacs Lisp 関数をいくつか組み合わせて定義された関数とに大別される。特殊形式は一般に前者に属する。後者の関数の定義が書かれたファイルは、lisp ディレクトリの下に収められている。接尾辞が .el となっているファイルが Emacs Lisp で書かれたソースファイルであり、.elc となっているファイルは、Emacs 内蔵のバイト・コンパイラを使って Emacs Lisp 専用の中間コードにコンパイルされたファイルである。このディレクトリの下で定義されている関数には、Emacs を作成するときに Emacs 内部に取り込まれ、既に Emacs から使用可能になっている関数と、あらかじめ予約だけされていて、実際に使われるときになってからロードされる関数がある。河内の塔 hanoi や Conway のライフゲーム life は後者である。

関数を定義する、つまり、シンボルに関数としての値をセットする代表的な方法は、特殊形式 defun (DEFine a FUNction) を用いる方法である。下記の例は引数に 2 を掛ける関数 2* を定義する。ここでシンボル NUM は仮引数として使われている。

```
(defun 2* (NUM) (* 2 NUM)) 2* [ただしこのとき 2*が定義される]
```

defun を評価して得られる値は、定義の対象となったシンボルそれ自身であるが、それはこの場合、重要ではない。むしろ、defun の評価における一種の副作用として、シンボル 2* に関数としての値が定義されたことに主要な意味がある。defun で定義された関数は、ほかの関数と全く同様に呼び出すことができる。

```
(2* 6)           12  
(+ (2* 6) 3)    15  
(2* (2* -10))  -40
```

次の例は、最初 2 段、次に 3 段の河内の塔を実行し、最後にその (河内の塔の) バッファを破棄する関数である。

```
(defun hanoi-hanoi ()
  (hanoi 2)
  (hanoi 3)
  (kill-buffer nil))
hanoi-hanoi
〔ただしこのとき hanoi-hanoi が定義される〕

(hanoi-hanoi) t 〔ただし結果の前に 2 段と 3 段の河内の塔が順に実行される〕
```

ここで定義した関数 hanoi-hanoi は、コマンドとして定義されていないので M-x で呼び出すことはできない。コマンドとして定義するには、定義の本体の最初で特殊形式 interactive を使う。すると Emacs はこれをコマンドとして認識する。

```
(defun hanoi-hanoi ()
  (interactive)
  (hanoi 2)
  (hanoi 3)
  (kill-buffer nil))
hanoi-hanoi
〔ただしこのとき hanoi-hanoi が (再) 定義される〕
```

このようにコマンドとして定義した関数は、以下のように M-x で呼び出すことができる。

```
M-x hanoi-hanoi RET
```

コマンドは、キーと組にすることもできる。コマンドをキーと組にする代表的な方法は、関数 global-set-key を使う方法である。第 1 引数でキーの列、第 2 引数でコマンドを指定する。たとえば、C-c / と hanoi-hanoi とを組にするには S 式

```
(global-set-key "\C-c/" 'hanoi-hanoi)
```

を評価すればよい。もしここで関数名を表すシンボル hanoi-hanoi が上記のように quote されなかったなら、シンボル hanoi-hanoi それ自体ではなく、その変数としての値がとられ、変数値未定義エラーが発生しただろう。

もちろん、global-set-key は、Emacs の標準的なキーとコマンドとの組合せの設定を変更したいときにも利用できる。下記は、C-h に DEL と同じ文字後退削除コマンドを割り当て、その代わりとして C-c h に ヘルプ・メニュー・コマンドを割り当てる例である (文字後退削除のコマンドは delete-backward-char であり、ヘルプ・メニューのコマンドは help-for-help である)。

```
(global-set-key "\C-h" 'delete-backward-char)
(global-set-key "\C-ch" 'help-for-help)
```

この二つの S 式をファイル .emacs に書いておけば、Emacs を起動するたびに、それが評価され、結果としてキー設定が変更されることになる。

なお、簡単に推察できるとおり、global-set-key に対し、local-set-key という関数も存在する。その引数は global-set-key と同様である。ただし、local-set-key でキーをセットした場合、それはそのバッファのもとでキーを押したときだけに有効になる点が異なっている (バッファローカル変数との類似性に注意しよう)。

7 ラムダ式とフック

シンボルの関数としての値は、defun で定義されるが、「関数としての値」とは具体的には何であろうか？シンボルの関数としての値を結果とする関数 symbol-function を使って調べてみよう。

```
(defun 2+ (NUM) (+ 2 NUM))    2+ [ただしこのとき 2+が定義される]
(symbol-function '2+)        (lambda (NUM) (+ 2 NUM))
```

このように「関数としての値」とは、lambda つまりギリシャ文字の λ (ラムダ) を先頭にして、仮引数リスト、関数本体の順に並んでいるリストであることが分かる。このような式を Lisp では、ラムダ式と呼びならわしている (ラムダ式、つまり expression は、元来、Church らが 1930 年代ごろ、基礎数学のための関数の表現方法として考案し、大戦後の 1950 年代、McCarthy ら Lisp の発明者が、Lisp で「関数としての値」を表す方便として採用したものである)。

極めて単純化して言えば、Emacs Lisp において関数とは、機械語になっているプリミティブと、ラムダ式の 2 種類しかない (特殊形式の実体はプリミティブである)。defun は、(変数値とは別の面で) シンボルにラムダ式をセットする。Emacs Lisp が S 式を評価するとき、関数のあるべきところにシンボルがあったなら、まずそのシンボルにラムダ式がセットされているかどうかを調べる。そしてもしセットされていれば、シンボルをそのラムダ式に置き換えてから評価を続行する (実際はもう少し複雑であるが、その問題はここで取り扱うべき範囲を越える)。

たとえば、いましがた定義した 2+ に引数 3 を

```
(2+ 3)
```

のように適用すると、内部的には

```
((lambda (NUM) (+ 2 NUM)) 3)
```

が評価されることになる。すると、ラムダ式の仮引数 NUM に実引数 3 がその時だけ一時的に代入されて

```
(+ 2 NUM) [ただしこのときに限り NUM の変数値は 3]
```

が評価され、

```
(+ 2 3)    5
```

となるのである。

もちろん、以下のようにラムダ式を、名前のない関数 (anonymous function) として直接使ってもよい。

```
((lambda (NUM) (+ 2 NUM)) 3)    5
```

ラムダ式は、つまるところリストであり、quote することもできる。しかし、Emacs Lisp では、数や文字列と同じく、ラムダ式はラムダ式自身へと評価されるように定義されている。したがって quote しなくてもよい。むしろ、コンパイルのときにリストではなく関数本体と認識されてバイトコード化の対象になれるようにするため、決して quote してはならない。(Emacs Lisp はおおむね伝統的 Lisp の仕様に沿っているが、この点では、より近代的な Lisp 方言に接近している。ラムダ式の quote には意味論的に興味深い問題があるが、それはここで取り扱うべき範囲を越える)。

```
(lambda (NUM) (+ 2 NUM))    (lambda (NUM) (+ 2 NUM))
```

名前のない関数で十分であり、defun で定義するまでもない、と判断したときには、関数名を書くべきところに、直接、ラムダ式を書いてもよい。下記は、ラムダ式を使ってキーをセットする例である（ここでは quote をしていないことに注意されたい）。この S 式を評価すると、C-c t のキー押下で現在の日付と時刻がエコー領域に表示されるようになる。

```
(global-set-key "\C-ct"  
  (lambda ()  
    (interactive)  
    (message "今は \%s です" (current-time-string))))
```

なお、ここで使われている関数 current-time-string は、下記のように、現在の日付と時刻を表す文字列を結果とする関数である。

```
(current-time-string) "Wed May 11 19:10:53 2005"
```

ただし、このようにラムダ式を直接、キーにセットすることには、(名前がないので M-x で呼び出せないなど) 多少の不都合がある点にも注意されたい。

.emacs には、このラムダ式がしばしば使われるところがある。それはフック (hook) である。フックは俗にホックとも呼ばれ、日常生活上、何かをヒョイと引っ掛ける器具を意味しているが、計算機の用語では、何かの動作をする前 (あるいは後) に、何かの処理を追加したいとき、本体を変更することなく、その追加処理を引っ掛ける場所や機構をあらわす。Emacs では、たとえば、Java 言語プログラム編集用の Java モードに切り替わる時、その関数 java-mode は、java-mode-hook の変数値 (がもし定義されていればそれ) を関数として (引数なしで) 呼び出す。したがって、java-mode 自体を変更しなくても、java-mode-hook に対し関数を

```
(setq java-mode-hook 関数)
```

のように代入しておけば、Java モードに切り替わる際の処理を追加できる。このとき、この関数に名前を付けずに、匿名のラムダ式を与えてもよい。

```
(setq java-mode-hook (lambda ()  
  (setq tab-width 4  
        indent-tabs-mode nil  
        case-fold-search nil))))
```

ただし、実際には setq ではなく add-hook を使うのが普通である。add-hook を使えば、複数箇所で別々に java-mode-hook を設定した時、以前の処理を上書きするのではなく、その処理の後に追加するように設定してくれるからである。

```
(add-hook 'java-mode-hook (lambda ()  
  (setq tab-width 4  
        indent-tabs-mode nil  
        case-fold-search nil))))
```

8 おわりに

多くの重要な点について触れていないとはいえ、これまでで Emacs Lisp の基本的な事柄についてひととおり説明した。今や、ほかの人の書いた `.emacs` やその他の Emacs Lisp プログラムを見ても、およその意味は理解できるはずである。

なお、Emacs Lisp についてさらによく調べたいときには、Emacs のオンライン・ヘルプを利用することができる。(ここでの設定によれば C-c h で現れるところの) ヘルプ・メニューで b を押せば、そのバッファでのキーと関数の組合せ、つまり、結び付き (Binding) が一覧できる。ヘルプ・メニューで c や k を押してキーを入力すれば、そのキー (Key) に結び付いている関数を知ることができる。ヘルプ・メニューで f や v を押して名前を入力すれば、それぞれその名前の関数 (Function) や変数 (Variable) についての説明を (それがもし用意されていれば) 見ることができる。系統的に調べたいときは、GNU Emacs Lisp Reference Manual が便利である。

A Emacs Lisp のコーディング書式

Emacs で `.emacs` や `*.el` を編集しようとする時、(変数 `auto-mode-alist` が変更されていない限り) 自動的にそのバッファは Emacs Lisp 編集専用モードになる。このモードでは、Java モードと同様、TAB や LFD で自動的な字下げが行われる。一般的には、その字下げに従うようにすればよい。

Emacs Lisp のコメントはセミコロンからその行の終りまでであるが、それについては、以下の取り決めに従って書くようにすると、後々便利である。M-; や TAB でも、この取り決めに従ってコメントの字下げが行われる。

- ';' セミコロン一つで始まるコメントは、右の端を同じ桁にそろえて書く。このコメントでは、その行の左に書かれた Lisp プログラムの働きを説明する (コメントが存在しない行では M-; は、自動的にこのセミコロンを書き込む)。
- ';;' セミコロン二つで始まるコメントは、Lisp プログラムと同じ字下げにそろえて書く。このコメントでは、以下の行に書かれたプログラムが何を目的に働くのか、あるいは、現在の状態はどうなっているはずかを説明する。あるいは、関数定義の直前の行に書いて、その関数がどんな関数か、どう呼び出せばよいかを説明する。
- ';;;' セミコロン三つで始まるコメントは、左端から書く。関数定義の中には書かない。このコメントでは、全体的、一般的な説明を述べる。
- ';;;;' セミコロン四つで始まるコメントは、左端から書く。プログラムの大きな区切りの先頭で題名を書くのに使う。