Emacs 解剖学

Lecture 完

ソースで遊ぼう

井田昌之

一番気になっていること

そもそもは，筆者が手元にもっている Emacs の入門テキストを紹介するということで引き受けた連載なのですが，いろいろないきさつで Emacs や Richard Stallman に関するほかでは聞かれない情報（？）を紹介するという趣旨に変わってきて，さまざまな手持ちのデータを引っ張り出して書いてきました．それも，とうとう与えられた回数の最後になりました．

それで，最近，一番気になっていることを書いてみようと思います．

「プログラムを作る」

このことがコンピュータサイエンスの根本にある共通基盤だと考えたとすると，「プログラムを作る（プログラミング）」そのものをどうやって伝えていくかということが，その重要課題の一つになります．

そうなってくると，やはり，技術の伝承とか共有とかいった側面を無視することはできなくなります（もちろん，科学としての側面が大きいことは言うまでもありません）．使う人，ころがす人ばかりになってしまってはなりません．

商品としてのソフトウェア，特にバイナリだけを供給されるソフトウェアの場合には，いろいろなやりにくさがでてきます．そのソフトウェアに組み込まれた「機能」については共有できるけれども，その機能はどうやって作られているのか，どうしたらそのような機能を作ることができるのかを知りたい人にとっては，それを知るすべがなくなってしまいます．

「設計目標としての参照」という価値以外の価値はないことになります．要するに，ブラックボックスとして機能だけを利用するということは，多数の人に簡単な操作でできる良い技術を供給する，という技術文明の根本的な目的には合致しているのですが，それを作り出す側にいたいという人，そういう人を育てたいという場所では役に立たないわけです．

「工房的プログラミング」

こんなことが，古くさい職人的なプログラミングの時代を終えたはずの現代でも重要な位置をもっていると考えています．多くの大学の多くのソフトウェアの研究室は，そんなことも教えているはずだと考えています．

先日，アメリカのあるワークステーションメーカの最大手の副社長と面談をしました．彼女は OS などのソースリストの公開を積極的にしたいし，それが大学とメーカとの接点だと言っていました．こういう視点を多くのメーカがもってくれるといいですね．同時に，当然，それを受け入れる側の態勢というのも問題になります．山のようなステップで作られ，いろいろな要素が複雑にからみあって作られている「ほんもののソフト」をいきなり目の前にしても，どこから手をつけていいのか困るということになります．そこに至るような適当な規模の小さなモデルやひな型を用意して，つながりをよくするような道具立てをすることが，大学側の準備として必要になります．それにはフリーソフトは適切なものだと言えます．

よく，フリーソフトは品質が劣ると言われます．筆者の観察では，それは多くの場合，「商品のソフトウェアを購入するのが一番いいのだろうが，お金をかけたくない．代わりにフリーソフトを使おう」という，「代用品としてのフリーソフト」が文脈の中で述べられています．

ソースを読もう

GPL（GNU Public License）の核にあるアイディアは，すべてのプログラマに自分が使っているソフトのソースコードを渡して，それによってみんなでソフトウェアを育てていこうという点にありました．ただでソフトをもらってこれるというのは一方向の流通であって，その意味で，価格がないというだけでソフトの販売と同じモデルです．これは FSF（Free Software Foundation）で考えている「互いに協力して育てていく」という双方向モデルとは異なります．

そうなると，GNU の利用者たるもの，その成果を享受するという段階の次に，ソフトを育てていくことに参加するということが重要になってきます．三つのファクタがあるでしょう．

- 第一に，ソフトウェアの品質の向上のために，使っているときに気がついたおかしな動作や改良のヒントなどを積極的に教えてあげること．いわば品質検査に協力すること．
- 第二に，こうしたソフトのバイナリをもらってきて使うだけでなくて，自分でソースから configure（make）し，ソフト製作の最終段階を追体験をすること．
- 第三に，ソースを読んで，そのソフトがどう作られているか理解すること．

この三つのファクタの上に，次の段階として，

「機能の拡張・改良をしてそれを配る，FSF に戻す」

ということがあります．

それぞれどうやってやるかということがありますが，ここではタイトルにあるように「ソースと遊んでみよう」ということを取り上げてみます．これはソースを読むということの一部あるいは前段階だと思います．プログラムを読んで理解して，そしてその元々の製作者のレベルまで達するというのは誰でもできるというものではありません．けれども，ちょっと寄り道をして，ソースを見ながら遊んでみることは誰にでもできることです，もし手元にソースプログラム（リスト）があれば…．そして，それがソースを読むことの出発点だと思っています．

Emacs にはソースがついている

Emacs では，その tar ファイルを解凍すると，さまざまなディレクトリがあります．それらにはすべての情報が入っています．etc というディレクトリもあって，その下には，履歴などのさまざまな補助情報などがあったり，info にはヘルプ情報があったりします．lib の下に，src と lisp というディレクトリがあります．src には C 言語によるソースが，lisp には Lisp 言語によるソースが入っています．Lisp 言語でのソースの部分は，実行時に，利用者が同じ名前の関数を定義することで，簡単に上書きして機能を変更することができます．インタプリタで動作しているので，コンパイルされたバイナリコードとソースコードを混在させて実行できる特徴がこれを支えています．

このために eval-last-sexp などという機能が用意されています．たとえば，どんな編集の中でも，そのバッファに（+ 1 2 3）などという式を入れてそこで（閉じ括弧のすぐ右にカーソルを置いて），M-x eval-last-sexp を実行させてみてください（そのまま入力するか，バインドされているキー，デフォルトでは c-x c-e，をたたく）．エコーエリア（Emacs 画面の最下行）に，実行結果 6 が表示されます．（setq XXX YYY）などという形で与えて実行させれば，XXX がシステムの変数であったとしてもその内容が YYY に変えられます．（defun XXX (...) ...）であれば，XXX という関数をそこで定義します．もし手元に Emacs があるなら，ぜひやってみてください．

Emacs 仮想マシン?

ソースと遊んでみる例として，「Emacs 仮想マシン」と筆者が勝手に呼んでいる部分とたわむれてみます．「Emacs 仮想マシン」とは，バイトコードコンパイラとその実行機能のことです．Emacs の基本機能

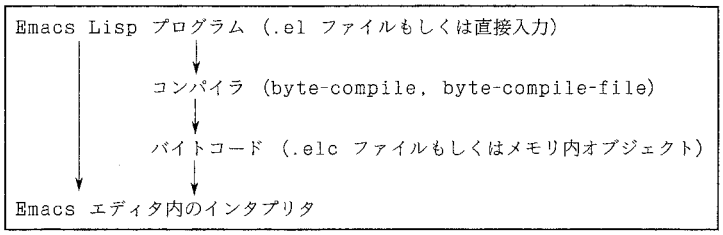


図1 Emacs Lisp コードの実行

はCとLispで書かれていますが，Emacs Lispで書かれた部分はバイトコードと呼ぶコードにコンパイルすることができ，Emacs Lispインタプリタは，Lispのソースと，それがコンパイルされたバイトコードの両方を解釈実行します（図1）．ファイルからロードする場合，ソースは.elというファイルタイプ，バイトコードは.elcというファイルタイプをもちます．

Emacs仮想マシンはスタックマシンで，演算対象をスタックに積んでおいて，それで演算命令を実行するような形をしています．たとえば1+2は，push 1, push 2, addというようなシークエンスで実行をします．結果はスタックに残ります．

この部分は，bytecomp.el, byte-opt.el, byte-run.elのLispコードとbytecode.cのCコードからなります．それは，grepしていったりファイル名を眺めていったりすることで，わかります．

バイトコードを取り上げるのは，本当は次の理由もあります．

Lispで書かれた部分はバイトコードコンパイラでコンパイルされますが，このコンパイラのことを知るのは，過去のこうした技術の歴史をひもとく鍵だともいえるし，将来のGUILE化への重要な橋渡しにもなっていることがあります．

また，最近注目されているJavaは，ある意味で「カッコのないLisp属のプログラムがその環境ごと別のマシンに飛んでいく」という性質をもっていて，しかも，JavaはJava仮想機械というマシンアーキテクチャの定義がベースになっています．このJava仮想機械と「Emacs仮想マシン」の距離はそれほど遠いものではありません．こうした比較も新しい発展をもたらしてくれるのではないかと勝手に考えています．

それで，本気を出して見ていくということも必要なのですが，ここではその入り口ということで．

Lap

Version 18のバイトコードコンパイラとVersion 19のそれとはまったく異なるソースが使われています．Version 19のコンパイラはJamie Zawinski（当時存在していたlucid社所属）が作り，彼と，Hallvard Furusethという人が作者になっています．なお，ZawinskiはJavaの仮想マシンの設計にもかかわっています．この点も，Java仮想機械との類似性の遠因を与えています．

Version 18のコンパイラは，関数名の属性にその機能のコンパイルを受け持つコードの名称をいれておき，それを利用してevalに準じた仕組みのままに直接的にコードを生成していきます．Lispとしてはよく見られるようなコンパイラでした．

これに対してVersion 19のコンパイラは，最適化フェーズをもっていて，ソースをlapコードに変換し，そのレベルで最適化などの処理をし，それをバイトコードバイナリに変換します．lapというのは，Lisp Assembly Programの頭文字から来ていて，古くからある呼び方です．リストの形で各命令を保持するものです．これも以前からあるLispコンパイラのひとつの姿を受け継いでいます．

たとえば，

```
((varref.a)(varref.b)(plus.0)
(return.0))
```

というような形でプログラムを表現するものです．

そしてlapは中間コードの役割を果たしていて，その段階で最適化が行なわれます．

ソースを見ると，「lapコードは途中で作られるが，捨てられて残らない」というような記述があります．当然なんとか見てみたいという気がおきます．

それをするのは普通のシステムなら大変ですが，

Emacs なら簡単です．そこの部分のプログラムを別のダミーを定義して置き換えてしまうといった方法があるのです．bytecomp.el のリストをコメントを手がかりにして見ていくと，byte-compile-lapcode という関数が引っかかります．そのコメントには，

"Turns lapcode into bytecode. The lapcode is destroyed."

などと書かれています．それは，

```
(defun byte-compile-lapcode(lap)...
```

で始まります．この引き数にコンパイル中の lap コードが来ているのでしょう，きっと．それで，この部分を置き換えてコンパイル中に，引数として来たものは何だったのか調べてみることにします．

```
(defun byte-compile-lapcode(lap)
(setq**lap**lap))
```

などとしてみます．そうすると，ここを通ったときの lap 引数の値が**lap**という変数に代入されるので，後から**lap**の値をつつきにいけばいいわけです．**lap**というのはシステムのもつ変数にはなさそうなもので適当に付けた名前です．もっと変な名前のほうがいいかもしれません（それから，本格的にこれをやるなら，もともとのコードは別の名前でとっておいて，(setq**lap**lap) をしたら，もともとのコードを素知らぬ顔で実行させる，などということをします．さらに，「とっておくという処理は，1回だけの実行にする」という処理が必要です．そうでないと，もし2度以上実行させると元のものがいなくなるなどということが生じたりしますが，その辺はいたずらを何回かやってみるとなるほどと気がつく範囲のものです）．

それで，試しとして，

```
(defun test(a b c)(+(- a b)c))
```

くらいのコードを M-x eval-last-sexp で定義し，それを，

```
(defun byte-compile-lapcode(lap)
(setq**lap**lap))
```

を実行させておいた上で (byte-compile 'test) させてみます．終わったら**lap**を M-x eval-last-sexp させます．

エコーエリアに test の lap コードが表示されます．見えましたか？

この lap コードから最終的なバイトコードが作られます．

どんなコードがでるのか？

それぞれのコードに対してどんな命令が生成されるかを知るには，disassemble という機能を利用します．これを対話型で使うと，*Disassemble*というバッファにその出力を表示します．

当然，Emacs システムの中身も Lisp で定義されているものであれば，disassemble で見ることができます．

ユーザが定義した関数もまったく同様にできます．それをしてみましょう．まず，Emacs Lisp プログラムを入力します．そして，それを M-x eval-last-sexp などで評価させます．

```
(defun test(a b c)(+(- a b)c))
```

そこで，M-x disassemble と入れると，

```
Disassemble function:
```

と聞いてくるので，

```
Disassemble function:test
```

と入れます．

すると，*Disassemble*バッファが現われ，図2のようなものがその中に表示されます．varref というのは変数の値を push する命令だと気がつくでしょう．ついでにいくつかの似たようなプログラムを与え，それを disassemble してみます（図3，図4，図5）．

図3のコードは図2をちょっと変えただけです．

図4は乗算を入れています．図3は図2とほとんど同じことですが，図4の (defun test(a b c)(+(- b c)(* d e))) になると，少し違った仕組みが入ってきます．(* d e) の部分は，push d, push e, multiply のようなシークエンスではなく constant*,varref d, varref e, call 2 となっています．call という関数呼出し命令が使われているようです．その引数には呼出して実行する関数の引き数の個数を与えるように見えます．さらに，一番底に関数のシンボルを push してあるように見えます．3引数の関数の呼出しであれば，call 2 ではなく call 3 が出て，さらに一番底に実行する関数が入ってくるはずです．test がちょうど3引数なのでそれを呼び出す関数を定義して disassemble してみま

```
byte code for test:
  args: (a b c)
0   varref   a
1   varref   b
2   diff
3   varref   c
4   plus
5   return
```

図2 (defun test(a b c)(+(- a b)c)) のディスアセンブル出力

```
byte code for test:
  args: (a b c)
0   varref   a
1   varref   b
2   varref   c
3   diff
4   plus
5   return
```

図3 (defun test(a b c)(+a(- b c)) のディスアセンブル出力

す．図5のようになります．そして，予想は当たっていたことがわかります．

elc ファイルの中味は?

lap のレベルで中味を見る方法，そして，出力のバイトコードを見る方法を説明しました．でも探求心の旺盛な（疑り深い？）読者の人は，本当のところ，バイナリファイルにはどんなものが入っているのか見たくなるでしょう．バイナリファイルを手術する，あるいは覗いてみる，最も簡単な方法は，やはり，Emacs に読ませることです（でした）．8進表示で出てくるので，それを見てなおしていくことができます．これで随分いろいろなバイナリの手術をしたことがあります．

けれども，Nemacs/Mule の場合には注意がいります．バイナリの中に日本語テキスト的シークエンスに似た部分が入っているとその辺をなんとか表示しようとしてしまい，そのままのバイナリが単純には見えないからです．多言語環境への対応という方向性は，おそらく Emacs でバイナリオブジェクトを変更することを邪道の極みに位置づけていくことになっていくのでしょう．

```
(defun test(a b c)(+(- a b)c))
```

をファイルとしてコンパイルすると，Version 18 ではおよそ以下のような内容のものが elc ファイルとして生成されます．

```
(defun test(a b c)
(byte-code "..." [a b c]2))
```

これは元のソースと同等の機能で扱いも同じでよく，かつ，実態は仮想マシンのバイトコードなので実行が速いということになります．

```
(defun test (a b c) (+(- b c) (* d e)))
byte code for test:
 args: (a b c)
0   varref   b
1   varref   c
2   diff
3   constant *
4   varref   d
5   varref   e
6   call     2
7   plus
8   return
```

図4 (defun test(a b c)(+(- b c)(* d e))) のディスアセンブル出力

```
(defun call-test (x) (test x(1+ x) 2))
byte code for call-test:
args: (x)
0   constant test
1   varref   x
2   varref   x
3   add1
4   constant 2
5   call     3
6   return
```

図5 (defun call-test(x)(test x(1+ x) 2)) のディスアセンブル出力

この byte-code 関数ですが，この引数は三つで，第一は実行するバイトコードからなる文字列，第二はデータベクタ，第三は max-stack となっています．データベクタにはその関数で使われているすべての定数，変数名，関数名が含まれます．

Version 18 では以上のような byte-code 関数が作られ，それが実行されたわけですが，Version 19 ではバイトコードオブジェクトというデータ型ができ，それが実行されます．したがって，Version 19（正確にはテストをしている 19.29 以降）での elc ファイルに入る中味は，オマケの部分を取ると，

```
(defalias 'test #[(a b c) "..." [a b c]2])
```

のようになります．defalias は，まぁ defun と同じようなものだと思ってください．test の関数オブジェクトにバイトコードオブジェクトをセットします．このバイトコードオブジェクトは，以下のような六つの要素をもっています．

1) 引数リスト（引数シンボルのリスト）
2) バイトコード（実行対象となる命令列をもつ文字列）
3) データベクタ（バイトコードが参照する Lisp オブジェクトのベクタ．定数，変数名，関数名）
4) スタックの大きさ（この関数が必要とするスタックの最大の大きさ）
5) 説明文字列（説明文字列．もしなければ nil）
6) 対話引数（対話引数の指定．なければ nil）

elc ファイルの先頭は Version 19 では ";ELC" で始まります．そしてその次にバージョン番号と 2 バイトの 00 が続きます．そして，Version 18 とは互換性がないので，それをはじくコードが挿入されています．先頭の行の仕掛けは，diff にかけても

「バイナリファイルの diff をしようとしている」

というメッセージだけを出し，比較をしにいかないようにするためだとソースプログラムの中に書かれています．また，これは UNIX の場合，/etc/magic ファイルに登録することで便利にもなれると書かれています．これを見て，はたと思い当たったのは，Java のクラスファイルのフォーマットです．Java のクラスファイルというのは Java 仮想マシンのコードで Java コンパイラの出力です．いわば elc ファイルの兄弟のようなものです．Java クラスファイルでは，16 進で CAFEBABE という 4 バイトが先頭にあって，その次にバージョン番号が続いています．

バイトコードにはどんな命令があるか

Emacs Lisp のプログラムが変換されるバイトコードの見方がわかると，次にはその中の命令がどんな機能になっているのか知りたくなるはずです．生成されたコードを見て，およその検討をつけるというのは，いい方法です．それぞれの命令にはオペランドがついているのもあり，それらをどうやって切り分けているのか，などということも気になってくるはずです．

そうすると，ソースプログラムをちゃんと読まないとだめだ，ということになってきます．また，スタックマシンでの動作の基本はなにか適当な参考書を読むのもいいかもしれません．

バイトコードの形式とその機能は bytecomp.el と bytecode.c にあります．

命令表やその機能などを作ってみましたが，そうとうなページ数になってしまうので，思い切ってすべてこの記事からは削除しました．興味のある人は命令表を作ってみると面白いでしょう．

東京セミナーの話

この 3 月に Richard Stallman 氏が来日し，第 4 回のセミナーを開きます（その詳細については「お知らせ」を参照ください）．久しぶりというだけでなく，特に今回は，Emacs と Mule が統合されて最初のセミナーなので，彼もはりきっています．文字の扱いの問題が大きいと同時に，この間，DOS 対応，Windows NT 対応のコードが組み込まれ，改良され，実行できるプラットフォームも UNIX だけでないと言えるようになってきました．これを受けて，半田さんや宮下さんの講演も予定しています．

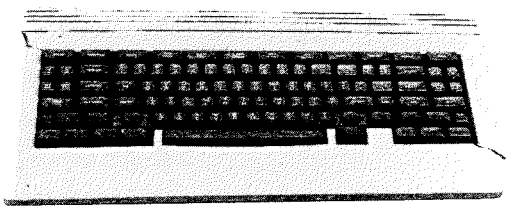
GNU のプロジェクトとしてはこれらに加えて，さまざまなものがあるわけですが，いろいろと議論した結果，Hurd，GNUstep，GUILE の三つがテーマとして取り上げられます．

Hurd はマイクロカーネルによる OS で，数年前からその出現が待たれていたものです．Richard は「Linux を GNU/Linux と呼ぶべきだ」と主張しています．その理由は Linux として独自に開発されたものは OS のカーネルだけであって，その上のツールは

LMI キーボードその後…

本 Lecture 第2回「Emacs の成立とキーボード」(Vol. 28, No. 7) で，LMI 社の Lisp マシンキーボードを取り上げましたが，資料が不鮮明だったために正確な文字配列を記した図を掲載することができませんでした．誌上で情報を募ったところ，山口智浩氏より現物をお借りできました．また，エピソードもお寄せいただきましたので，以下にご紹介いたします．

山口氏には，この場を借りて深謝申し上げます．



LMI 社の Lisp machine は，1984 年に辻研究室で購入し，5～6 年くらい現役で活躍しておりました．その後は，一気に UNIX ワークステーションの時代になり，AI マシンは使われなくなったのですが，愛着があったため（当時学生だった私の所属した研究グループの名称が Lisp チーム），研究室の職員の方にお願いして，キーボードとディスプレイ（縦長）を記念に残していました．

残念ながら，阪神・淡路大震災の際に両方とも破損し，ディスプレイのほうはゴミとして捨てられ，キーボードのほうも一部破損しております．

現物を見ていただくとわかるように，右上の手の形の方向キーやギリシア文字など，標準でないキーがずいぶん多いのですが，それらの使用法はアプリケーション依存だったようです．

キーボードを残していた理由の一つに，GNU Emacs での Meta キーの存在の歴史的証拠があります．さらに，Zmacs エディタ上で Hyper キーは，次のように Lisp 固有の機能のショートカット[1]として使われていました（下二つは Emacs Lisp にもある）．

キー操作	意　味
Ctrl-Hyper-C	Compile Region (Lisp ソースのコンパイル)
Ctrl-Hyper-E	Evaluate Region (Lisp ソースの実行)
Ctrl-Hyper-V	Describe Variable (変数情報を返す)

また，Ctrl，Meta キーなどは，Shift キーと同様に左右両方にあり，通常はどちらでも同じですが，キーボードからのコールドリブート[2]は，

Ctrl-Meta-Ctrl-Meta-Rubout

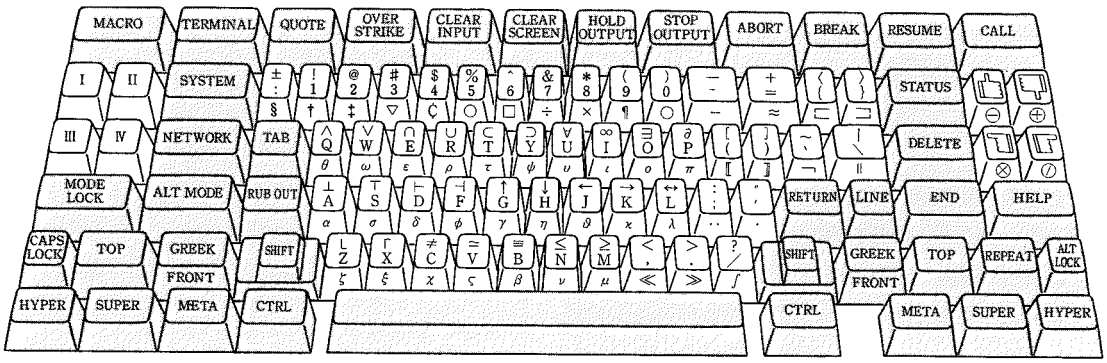
と青色の五つのキーを，両手で同時に押さえ続ける操作でした．

以上，ご参考になりましたら幸いです．

参考資料

1) ZMACS Introductory Manual, LMI, 1983.
2) Lisp Machine Manual, 1983.

(山口智浩　大阪大学 基礎工学部 システム工学科)



Lisp マシンキーボード（網のかかったところが，青色になっています）

すべて GNU Project の結果を利用しているにすぎないというのです．だから，全体としては Linux としてのオリジナル部分だけでなく GNU で作られているものすべてを含めているのだから，GNU/Linux と呼ぶべきだというわけです．ここでの対象からはずれるので，このことの本質的な議論はしませんが，Linux に対する彼の視点の中には，Hurd の遅れ（といっても諸般の状況を考えれば納得できる範囲だと私は考えるのですが）に対する彼のあせりというかいらだちを感じたことがあります．Hurd は，まだ 1.0 版までこないわけですが，ともかくもようやく公開され出しています．

GNUstep は，GPL 準拠の OpenStep 相当のツールです．GUILE は，GNU での機能拡張用言語の今後の標準言語にしようとしているものです．Cygnus (GNU ソフト利用のサポートを中心にしている会社) での開発の流れと，それと独立して，カリフォルニア大学バークレイ校に行った Fredman がやっているものがあるようです．これらの開発に関する最近のいきさつはよく知りませんので，私自身もその話が聞けるのが楽しみです．私の観察では，Richard がどの程度どういう形で，どういう時期に直接手を出すかによって，そのプロジェクトが成功するか失敗するかはかなり違いがでるので，これらの三つについてもその点からも眺めてみたいと思っています．GUILE については相当自分でやろうとしていましたが，必ずしもそれがプラスの方向で他の人とベクタがあっているようには思えませんでした．

ともかく，3 月 11 日にフリーソフトウェア財団の主催で開かれるセミナーに期待しています．申込方法などは，http://csrl.ge.aoyama.ac.jp/FSFseminar/ にあります．

おわりに

Emacs は Richard Stallman 氏の作品ではありますが，同時に多くの人の共有財産の性格があると思っています．この連載を読んでそういう性質について多少でも理解してもらえたら幸いです．また，Emacs は Version 20 のベースとなるとされる Version 19.34 から Mule との統合が予定されていて，すでにそのように進んでいるようです．これは半田剣一氏（電総研）の努力が大きいものと言えます．

また，最近，Windows NT 用の Mule では TCP/IP もつながり，メールや W3 などもそのままの感覚でできるようになったことを開発をしている宮下氏（京大）からのメールで知りました．そうなると，NT でも Emacs に入ったままで多くの仕事ができるはずで楽しみです．きっとまだまだ Emacs はネット上のコミュニティで改良/進歩がされていくことでしょう．

（いだ まさゆき　青山学院大学 国際政治経済学部）