インターフェイスの街角 (77) — 矢印キーの活用

増井俊之

Emacs と矢印キー

現在、UNIX の多くのユーザーに使われている Emacs は、エディタとしての機能は最高ではないかもしれませんが、簡単なプログラムを書いて自由に機能拡張ができるという大きな利点があります。

ところで、Emacs では、エディタのもっとも重要な機能の 1 つであるカーソル移動に Ctrl-P や Ctrl-N などのキーが割り当てられています。慣れてしまえば、これらのキーでカーソルを移動させる操作は苦になりません。しかし、矢印キーでカーソル移動が可能なエディタも多い現実をみると、初めて使う人にとってこのようなキーボード割当ては違和感があるのではないでしょうか。

矢印キーなどは初心者用のものであり、ハッカーともあろうものがそんな軟弱なキーを利用するのは言語道断、と思われているのかは定かでありませんが、ハッカーとして名高い東大名誉教授の和田英一先生が中心になって開発された「Happy Hacking Keyboard」には矢印キーは付いていませんでした。しかし、現在のほとんどの PC のキーボードには矢印キーがありますし、Happy Hacking Keybaord の後継モデル「Happy Hacking Keyboard Lite」にも付いています。ハッカーからみると、このようなキーは邪道なのかもしれませんが、いろいろなアプリケーションで有効に使えることも多いので、活用法をもっと考えてもよいのではないでしょうか。

上に述べたように、矢印キーはほとんどの計算機のキーボードに付いていますが、熟練した Emacs ユーザーが矢印キーを活用しているかどうかは疑問です。慣れている人は、両手をホームポジションからなるべく動かさずにあらゆる処理を実行しようとするので、カーソル移動も矢印キーではなく、Ctrl-N などのキーを使っている人がほとんどではないかと思います。たしかに、カーソルの移動だけが目的なら、矢印キーよりも Ctrl-N などのほうがメリットがありそうです。しかし、矢印キーをそれ以外の操作でも使えるようにすれば、もっと使い途がひろがるのではないでしょうか。

たとえば、Windows の MS Word やメモ帳では、Ctrl キーや Fn（ファンクション）キーと矢印キーを組み合わせて次のような操作ができます。

■ メモ帳でも可能ですか?
■ Fn キーは通常の PC のキーボードにはないのでは?

- Ctrl-矢印キー：単語単位の移動
- Shift-矢印キー：領域選択
- Fn-矢印キー：ページ移動

標準の Emacs ではこのような組合せは用意されていないようですが、Emacs でも工夫すれば矢印キーをもっと有効に活用できそうです。

コンソールで動く Emacs では、たとえば〝↑〞（上向き矢印キー）と Ctrl キー/Meta キー[1]を組み合わせれば、4 種類のキー（↑、Ctrl-↑、Meta-↑、Ctrl-Meta-↑）をコマンドに割り当てることができます。また、カーソル移動に関連するキーについても、Ctrl-F や Meta-F、Ctrl-Meta-F などに異なるコマンドを割り当てられますが、このような組合せはあまり活用されていないようです。これらを使えば、カーソル移動だけでなく、方向が関係するさまざまな編集作業に矢印キーが使えるでしょう。

今回は、領域の選択/移動や罫線の描画などに矢印キー

1 一般的な PC のキーボードには Meta キーがないので、Alt キーで代用するか、あるいは、ESC キーを押してから該当するキーを押す操作で代えることができます。

著者校正 [2004 年 6 月 5 日]

を活用する方法を考えてみます。

領域選択と領域移動

編集中のテキストの一部を選択して移動/複製するのはエディタの基本的な機能ですが、Emacs ではこのような基本的な操作がなぜかあまり簡単ではありません。こうしたコピーや移動をおこなう場合、Emacs では Ctrl-Space (Ctrl-@) に割り当てられた set-mark コマンドで〝マーク (mark)〟位置を指定してからカーソルを移動させ、設定したマークの位置と現在のカーソル位置 (point) とのあいだの領域 (region) に対してさまざまな操作を実行するのが一般的です。

たとえば、通常、領域の移動は以下のような手順でおこないます。

1. Ctrl-Space でマーク位置を指定する。
2. カーソルを動かして領域を指定する。
3. 領域の内容をバッファにコピー (kill) する。
4. 移動先にカーソルを動かす。
5. バッファの内容をその位置にペーストする。

マークやバッファといった概念は Emacs の世界以外では馴染みのあるものではないので、初心者に説明するのが難しいばかりか、熟練ユーザーにとっても、バッファの内容が目に見えないためちょっとしたミスを犯しがちです。たとえば、なんらかの理由で上記の 3 番目の操作 (バッファへのコピー) に失敗しても、明白には分からないので、5 のペースト操作のあとでやっと気付き、最初から全部やりなおしということがよくあります。

■ 移動 (kill → yank) の操作では、退避させた部分が目の前の画面から消えるのですぐに分かるのでは?

一方、グラフィカルなユーザー・インターフェイスをもつテキストエディタの多くでは、マウスで領域を選択してから Drag&Drop で領域を移動する操作ができます。この場合は、バッファのような概念を導入する必要はなく、コピーの失敗に気づかないこともほとんどありません。このような GUI ベースでの操作とくらべると、Emacs の標準的なコピー/ペースト方式は問題が多いように思われます。

マウスが使えなくても、領域の選択や移動を矢印キーで実行ができれば、この問題はかなり改善されるのではないでしょうか。たとえば、

- 領域の選択

図 1 最初の状態

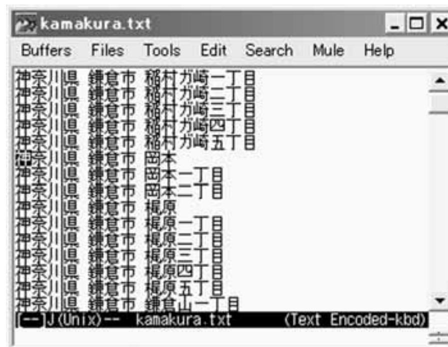


図 2 カーソル移動で領域を選択

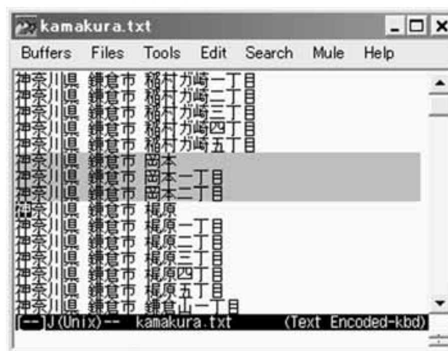
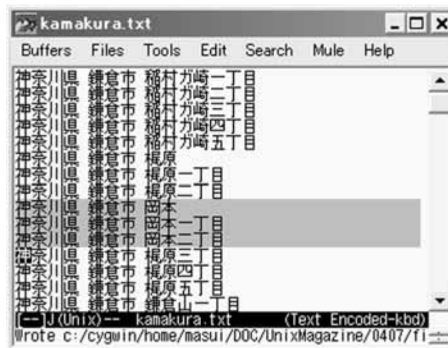


図 3 選択領域を移動した状態



- 領域の移動

に矢印キーを使えば、さきほど示した方法よりはるかに簡単に領域の指定と移動が実行できそうです。キー割当ては、たとえば次のようにすればよいでしょう。

- Ctrl キーを押しながら矢印キーを押すと領域が選択される (マウスによる領域の選択と同様な動作をする)。
- Meta キーを押しながら矢印キーを押すと領域が移動す

著者校正 [2004 年 6 月 5 日]

リスト 1 cursor.el

```
(defun move-region (func)
  (if mark-active
      (let (m)
        (kill-region (mark) (point))
        (funcall func 1)
        (setq m (point))
        (yank)
        (set-mark m)
        (setq deactivate-mark nil)
        )
    (funcall func 1))
  )

(defun move-region-right ()
  (interactive)
  (move-region 'forward-char)
  )

(defun move-region-left ()
  (interactive)
  (move-region 'backward-char)
  )

(defun move-region-up ()
  (interactive)
  (move-region 'previous-line)
  )

(defun move-region-down ()
  (interactive)
  (move-region 'next-line)
  )

(defun dup-region ()
  (interactive)
  (if mark-active
      (let (m)
        (kill-region (mark) (point))
        (yank)
        (setq m (point))
        (yank)
        (set-mark m)
        (setq deactivate-mark nil)
        )
    (forward-char 1))
  )

(defun select-region (func)                    ↗
↘(if (not mark-active) (set-mark (point)))
  (funcall func 1)
  )

(defun select-region-right ()
  (interactive)
  (select-region 'forward-char)
  )

(defun select-region-left ()
  (interactive)
  (select-region 'backward-char)
  )

(defun select-region-up ()
  (interactive)
  (select-region 'previous-line)
  )

(defun select-region-down ()
  (interactive)
  (select-region 'next-line)
  )

(defun forward-char-mark-inactive ()
  (interactive)
  (setq mark-active nil)
  (forward-char 1)
  )

(defun backward-char-mark-inactive ()
  (interactive)
  (setq mark-active nil)
  (backward-char 1)
  )

(defun next-line-mark-inactive ()
  (interactive)
  (setq mark-active nil)
  (next-line 1)
  )

(defun previous-line-mark-inactive ()
  (interactive)
  (setq mark-active nil)
  (previous-line 1)
  )
```

る（マウスをドラッグして選択領域を移動するのと同様の動作をする）。

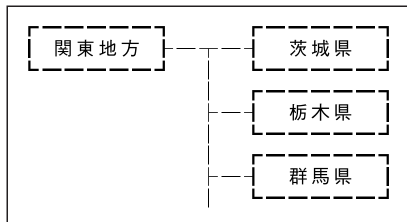
- 矢印キーだけを押すと普通にカーソルが移動し、領域選択は解除される。

これを利用してテキストを編集している例を示します。図 1 は編集前のテキストの状態です。ここで、Ctrl キーを押しながら矢印キー（↓）でカーソルを下に移動すると、図 2 のように領域が選択されます。続いて、Meta キーを押しながら矢印キー（↓）を押すと、図 3 のように領域が移動します。

このようにすれば、領域の選択や移動にマークを設定したりバッファに一時コピーする必要はなく、マウスによる

著者校正 [2004 年 6 月 5 日]

図 4　罫線素片を使って描いた図



選択/コピー操作と同じような感覚で使えることが分かります。

■ keisen.el に関する設定はコメントアウトしました

この操作は、リスト 1 の Emacs Lisp プログラム cursor.el を読み込み、以下のようなキー割当てをすることで実現できます。

```
(global-set-key [right]
                 'forward-char-mark-inactive)
(global-set-key [left]
                'backward-char-mark-inactive)
(global-set-key [up]
                 'previous-line-mark-inactive)
(global-set-key [down]
                'next-line-mark-inactive)
(global-set-key [C-right] 'select-region-right)
(global-set-key [C-left]  'select-region-left)
(global-set-key [C-up]    'select-region-up)
(global-set-key [C-down]  'select-region-down)
(global-set-key [M-right] 'move-region-right)
(global-set-key [M-left]  'move-region-left)
(global-set-key [M-up]    'move-region-up)
(global-set-key [M-down]  'move-region-down)
(global-set-key "\M-f"    'move-region-right)
(global-set-key "\M-b"    'move-region-left)
(global-set-key "\M-p"    'move-region-up)
(global-set-key "\M-n"    'move-region-down)
```

罫線の描画

いわゆる罫線素片を使うと、図 4 のようなちょっとした図をメモやメールなどの文中に含めることができます。

Emacs で〝┣〟や〝┛〟などの罫線素片を並べてこのような図を作ろうとするとけっこうな手間がかかりますが、1 文字ずつ入力するのではなく矢印キーの移動で線を引くようにすれば、比較的簡単に描けるはずです。末尾のリスト 8 に示した keisen.el[2] を使えば、罫線による図を矢印キーで簡単に描けるようになります。

2 http://www.pitecan.com/Keisen/keisen.el
keisen.el は 1990 年に Nemacs 用に開発したものですが、今回、Emacs 20.x で動くように若干修正しました。Emacs のバージョンが変わるたびに手作業で修正しなければならいのは、どうにも面倒です。

図 5　左方向から罫線を描画

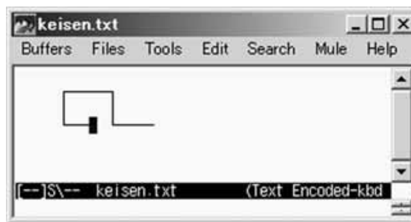


図 6　右に移動して「└」に到達

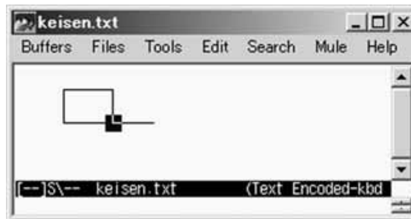
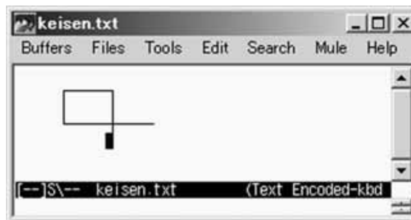


図 7　下向きに方向変換した結果



以下のように、keisen.el の罫線移動コマンドを Ctrl-Meta-矢印キーに割り当てておくと、Ctrl キーと Meta キーを同時に押しながら矢印キーを押して罫線を描いていくことができます。

```
(global-set-key [C-M-right] 'keisen-right-move)
(global-set-key [C-M-left]  'keisen-left-move)
(global-set-key [C-M-up]    'keisen-up-move)
(global-set-key [C-M-down]  'keisen-down-move)
```

keisen.el では、現在のカーソル位置にある文字の種類（└や┤など)、カーソル移動方向、直前のカーソル移動方向、罫線の太さをもとにして描く文字を選択します。たとえば、現在のカーソル位置にある文字が〝└〟でカーソル移動方向が下方向、直前のカーソル移動方向が右方向である場合は、カーソル位置の文字を〝┼〟に置き換えてから下方向にカーソルを移動します (図 5〜7)。

例に挙げたような簡単な図ならワープロソフトなどで描けばいいと思われるようになってきたせいか、最近は罫線素片による図をあまり見かけなくなりました。しかし、見方を変えれば、普通のテキストエディタには簡便な描画に

著者校正 [2004 年 6 月 5 日]

適した機能がないためとも考えられます。Emacs などのエディタで罫線が簡単に扱えるのなら、テキスト中に絵を描くことがもっと一般化するかもしれません。とくに、地図などをメールの本文に含めたい場合に重宝するのではないでしょうか。

plain2 のようなソフトウェアで処理すれば、罫線で描いた図を TEX や HTML の表形式に変換することができます。これなら、最終的に TEX や HTML のテキストが必要な場合にも使えるので、罫線による図を活用する場面も増えるでしょう。

キー入力速度の利用

領域処理と罫線処理にいろいろなキーを割り当ててしまったので、Ctrl キー、Meta キーと矢印キーの組合せをすべて使いきってしまいました。〝次単語への移動〞や〝行末への移動〞など、カーソル移動に関連したコマンドはほかにもあるので、できればこれらの機能にも矢印キーを活用したいところです。

東京工業大学の小松浩幸氏が公開している accel-key.el[3] を使うと、キーを入力する速度によってカーソル移動の動作を変えることができます。これを利用すれば、さらに多くの機能を矢印キーに割り当てられるでしょう。

Emacs で accel-key.el を読み込んだ場合、普通に矢印キーを押すとカーソルが移動しますが、キーを押し続けるとカーソルの移動速度が速くなったり、マウスのダブルクリックのように同じキーを連続的に押すと次単語にジャンプしたりします。このプログラムと同様な手法を使ってキー入力のパターンを変更すれば、矢印キーにさらに多くの機能を割り当てられるようになります。

その他のキー割当ての改善

矢印キーの活用方法を紹介してきましたが、まだまだキーの利用法を工夫してエディタを使いやすくする余地は残されているように思います。

選択領域と削除キー

多くの GUI アプリケーションでは、領域が選択されているときに削除キーを押すと選択領域がすべて消え、文字入力をおこなうと選択領域が新たに入力された文字列に置き換わります。一方、Emacs では領域選択が無効になるだけです。Emacs しか使わないのならそれでもかまいませんが、ほかのアプリケーションとの整合性を考えると、領域を選択している状態で削除キーを押したり文字入力をおこなった場合、その領域を削除することにしてもよい気がします。

この機能の実現はごく簡単で、考え方としては、領域選択の有無に応じて削除キーの動作を変えればよいことになります。

余ったキーの活用

上記のように仕様を変更すると、削除操作に割り当てたキーで選択領域を削除できます。結果として、Ctrl-W キーで kill-region コマンドを実行しなくてもよくなります。また、矢印キーで簡単に領域選択ができるのなら、Ctrl-K に割り当てられている kill-line コマンドの出番も減るかもしれません。そうすれば、Ctrl-W や Ctrl-K などのキーをほかの機能に割り当てることができるようになります。また、Ctrl-F や Ctrl-B のようなキーをカーソル移動に使うのはやめ、カーソル操作はすべて矢印キーで実行することにすれば、これらのキーも別の用途で使えます。

このように、矢印キーの活用によって〝余った〞キーにも別の有用な機能を割り当てることができます。たとえば、これらのキーはすべて日本語入力の効率化に使うといった工夫をするといいかもしれません。

; キー

QWERTY キーボードのホームポジションでは、右手の小指が〝;〞（セミコロン）の位置にあります。セミコロンを入力することはあまりないのに、このような大事な場所に割り当てられているのはなぜなのでしょうか。とくに、日本語ではセミコロンを入力する機会はめったにありません。このような重要な位置には、もっとよく使うキーを割り当てたほうが便利でしょう。思いきってセミコロンの位置に改行キー (newline) を割り当ててみたところ、右手の指の移動量がかなり減って快適になることが分かりました。

これは Emacs にかぎったことではありませんが、このようなちょっとした工夫をすれば、キー入力がさらに効率よくおこなえるようになるはずです。

3 http://www.taiyaki.org/elisp/accel-key/

著者校正 [2004 年 6 月 5 日]

おわりに

矢印キーを活用するには、押しやすい位置にあることが重要です。ところが、普通の QWERTY キーボードの場合、ホームポジションに指を置いたまま矢印キーを操作するのはそう簡単ではありません。

ホームポジションから手を動かさないことを重視するのなら、さきほどとは逆の発想で、Ctrl-F、Ctrl-B などを矢印キーと同じ扱いにすればいいかもしれません。このあたりは、いろいろなキー割当てを試してみて、ある程度の期間、使い続けないと効果のほどは実感できないでしょう。

キーボードやテキストエディタは毎日のように使う基本的なシステムですが、意外に使いにくいところがたくさん残っているように思います。一般的な PC のキーボードではそれほど気になりませんが、 sigmarion III の小さなキーボードを使ったりしていると、キーの打ちにくさという問題が増幅され、効率の悪さに気づくことが多いようです。ちょっとカスタマイズに凝って、キーボードを使いやすくする技術をきわめてみようかなどと思っています。

（ますい・としゆき　産業技術総合研究所）

図 8　keisen.el

```
;; keisen.el -- provide facility for drawing ruled-line

(require 'picture)

(defconst keisen-right 1)
(defconst keisen-up 2)
(defconst keisen-left 4)
(defconst keisen-down 8)

(defconst keisen-table "\
***└*─┘┴*┌│├┐┬┤┼**********┝*****\
*****┸**********┗***************\
**********┥*****━*┷*****┯*┿*****\
┛***************┻***************\
*****┰**********┏***************\
┃┠**┨╂**********┣***************\
┓***************┳***************\
┫***************╋***************"
  "罫線キャラクタの各方向の枝の有無を8ビットで表現する。
インデックスの上位4ビットは太い線の有無を示し、下位4ビットが
細い線の有無を示す。")

(defvar keisen-width 1
  "罫線の太さ。1のとき細く、2以上のとき太い。")

(defun keisen-toggle-width ()
  "罫線の太さを切り換える"
  (interactive)
  (cond
   ((> keisen-width 1) (message "細い罫線を使用します") (setq keisen-width 1))
   (t (message  "太い罫線を使用します") (setq keisen-width 2))
   ))

(defun keisen-opposite-direction (dir)
  (cond
   ((= dir keisen-right) keisen-left)
   ((= dir keisen-left) keisen-right)
   ((= dir keisen-up) keisen-down)
   ((= dir keisen-down) keisen-up)
   (t 0)
   ))
```


著者校正 [2004 年 6 月 5 日]

```
(defun keisen-direction (command)
  (cond
   ((eq command 'keisen-right-move) keisen-right)
   ((eq command 'keisen-left-move) keisen-left)
   ((eq command 'keisen-up-move) keisen-up)
   ((eq command 'keisen-down-move) keisen-down)
   ((eq command t) keisen-last-direction)
   (t 0)))

(defun keisen-new-string ()
  (let (pos factor str old-direction new-direction)
    (setq old-direction (keisen-direction last-command))
    (setq new-direction (keisen-direction this-command))
    (setq keisen-last-direction new-direction)
    (setq factor (if (> keisen-width 1) 16 1))
    (setq str (if (eobp) " "
                (buffer-substring (point) (+ (point) 1))))
    (setq pos (string-match str keisen-table))
    (if (null pos)
        (progn
          (setq pos 0)
          (if (= old-direction (keisen-opposite-direction new-direction))
              (setq old-direction new-direction))
          (if (= old-direction 0) (setq old-direction new-direction))
      ))
    (setq pos (logior pos
                      (* (keisen-opposite-direction old-direction) factor)
                      (* new-direction factor)))
    (substring keisen-table pos (+ pos 1))
    ))

(defun keisen-move (v h)
  (setq picture-vertical-step v)
  (setq picture-horizontal-step h)
  (setq picture-desired-column (current-column))
  (picture-insert (string-to-char (keisen-new-string)) 1)
  )

(defun keisen-right-move ()
  "罫線を引きながら右方向に移動する"
  (interactive)
  (keisen-move 0 1))

(defun keisen-left-move ()
  "罫線を引きながら左方向に移動する"
  (interactive)
  (keisen-move 0 -1))

(defun keisen-up-move ()
  "罫線を引きながら上方向に移動する"
  (interactive)
  (keisen-move -1 0))

(defun keisen-down-move ()
  "罫線を引きながら下方向に移動する"
  (interactive)
  (keisen-move 1 0))
```


著者校正 [2004 年 6 月 5 日]