taiyaki.org/elisp programming techniques

小松 弘幸 <komatsu@taiyaki.org>
2002-09-20: Linux Conference 2002

1. はじめに

Emacs のもっとも強力な利点は Emacs Lisp (elisp) による拡張性の高さである. Emacs Lisp とは Emacs が内蔵する拡張言語であり, 利用者は Emacs Lisp を用いて Emacs を自由に拡張できる. Emacs Lisp は Emacs の機能拡張に特化した言語であるため, C 言語で書かれた Emacs のソースコードを変更する手段に比べ, 手軽に拡張できる.

筆者は Emacs の拡張性の高さと Emacs Lisp に魅了され, さまざまな Emacs Lisp のソフトウェアを作成し, 筆者のウェブサイト [1] で公開している.

現在は以下が主なソフトウェアである.

physical-line	カーソルの物理行移動を実現するマイナーモード
pobox-el	日本語予測入力システム POBox [7] 用 Emacs クライアント
sense-region	箱型すなわち矩形領域の編集支援
ac-mode	ファイルパス・URL・文字列の補完およびインデントを TAB キーのみで実現するマイナーモード
word-count-mode	指定範囲内の文字数・単語数・行数を動的に計測して表示する マイナーモード
text-adjust	文章の形式を整形する.「全角・半角文字の変換」「句読点の置換」「全角文字と半角文字の間への空白の挿入」などを行う
tspeech	キー入力に対する音声フィードバック
kakasi	わかち書きソフトウェア KAKASI を Emacs で使用するためのライブラリ
table	プレーンテキストから, csv・HTML・LaTex 形式などの表を作成する
accel-key	マウスのように, キーボードのダブルクリックによって 実行するコマンドを変化させる
urlencode	文字列に対して RFC1738 にもとづいた URL エンコードおよび デコードをする
visible-mark	改行記号と EOF 記号を表示する. (XEmacs 専用)

本論文では, 筆者による physical-line・pobox-el・sense-region の 3 つの Emacs Lisp を紹介すると共に, それぞれのソフトウェアを制作する上で 得られたプログラミング技術を紹介する.

physical-line は既存の関数をアドバイス (advice) によって拡張している. 2 節では, physical-line の紹介とアドバイスの説明と 利用方法の議論を行う.

pobox-el は POBox サーバとネットワーク通信を行うほか, いくつかの外部ソフトウェアとデータをやりとりする. 3 節では, pobox-el の紹介と外部ソフトウェアの利用方法を 議論する.

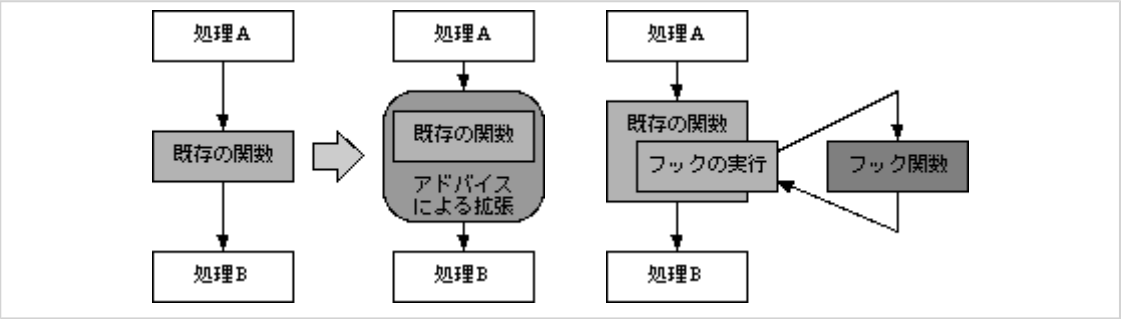
sense-region の実現には, リージョン情報の取得が不可欠である. 4 節では, sense-region の紹介とリージョン情報の取得方法の 議論を行う.

なお, Emacs Lisp に関する基本的な内容は本論文では扱わない. 必要に応じて参考文献 [2-6] などを参照して欲しい.

2. アドバイスによる機能拡張

アドバイス (advice) は, 既存の関数を拡張する手法である. アドバイスを用いると, 既存の関数を実行する前後に独自の処理を挟むことができる (下図左). そのため, 利用者は既存の機能を再実装する必要はなく, 拡張機能のみを実装すればよい.

アドバイスに似た機能にフック (hook) がある. フックを用いると, 利用者は既存の関数の実行途中で, 利用者が作成したフック関数を実行できる (下図右). アドバイスは拡張対象となる関数の外側に位置するのに対して, フックは内側に位置するとも理解できる.



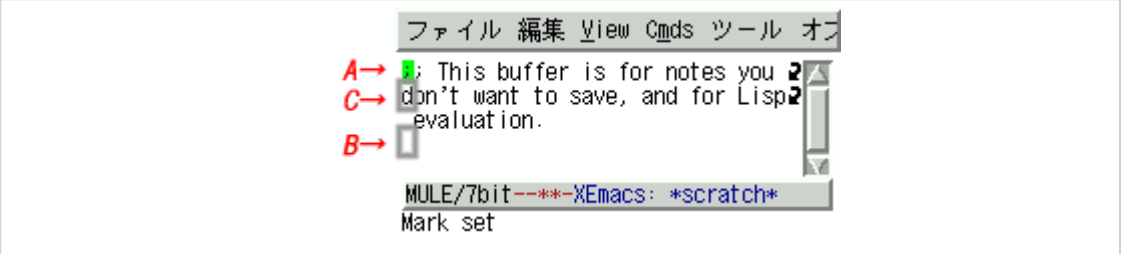
アドバイスとフックの概念図

フックを利用するには, 拡張対象となる既存の関数が あらかじめフックを利用するように作成されていなければならない. 加えて, フック関数を実行するタイミングは変更できない. そのため, フックが利用できない関数や, フックの実行タイミングが適当でない関数の拡張には, アドバイスが必要である.

アドバイスを用いて実装された taiyaki.org/elisp に physical-line がある. 本節では physical-line を題材に, アドバイスについて議論する.

2.1. physical-line: 物理行によるカーソル移動

physical-line はカーソルの物理行移動を実現するマイナーモードである. 下図のように, 文章の一行の長さが画面の横幅よりも長い場合を考える. 図中 (A) の位置にあるカーソルを下に移動させると, 従来の動作では図中 (B) に移動する. なぜなら Emacs での従来の上下の移動は, 改行コードから改行コードまでの 論理的な一行の移動だからである. しかし状況に応じては, 画面に表示されている物理的な行に基づいて カーソルを移動したい場合がある. つまり, 図中 (A) から (C) への移動である. physical-line はこの物理行に基づくカーソル移動を実現する.



physical-line でのカーソルの動き:
A→B: 従来の動作, A→C: physical-line での動作

physical-line では, 既存の関数の line-move をアドバイスによって拡張している. line-move はカーソルを縦に移動させる関数である. 関数 previous-line (上カーソルキー) や 関数 next-line (下カーソルキー) なども,

line-move を内部で実行している. カーソルの物理行移動は previous-line や next-line の拡張を行ったり, 新たに作成したコマンドにキーバインドを割り当てることによっても実現できる. しかし, その場合はカーソルの移動を行うすべてのコマンドを 拡張しなければならず, 実装が複雑になる. そのため, line-move をアドバイスによって拡張することは合理的である.

line-move の拡張は defadvice を用いるのではなく関数の再定義によっても 可能である. しかし, これも非常に複雑な作業となる. 関数 line-move を定義しているソースコードの長さは, Emacs21 の場合, 100 行を越える. このように長いソースコードの一部を変更して再定義するのは, 拡張箇所が分かりづらくなるうえ, 保守も難しい. しかも Emacs と XEmacs で line-move のソースコードは異なるため, 関数の再定義による機能拡張の方法は, さらに複雑になる. そのため, アドバイスを用いる方法がよい.

2.2. defadvice

アドバイスの使用は関数 defadvice を用いて行う. 下記のコードは physical-line での defadvice の使用例である.

```
(defadvice line-move (around physical-line-move disable)
  "Move cursor to same column of frame line down ARG lines."
  (if (not physical-line-mode)
      ;; physical-line-mode 以外のときは何もしない
      ad-do-it
...
    (let* ((arg (ad-get-arg 0))
           (moved (physical-line-the-vertical-motion arg)))
...
```

defadvice のコード例 (一部)

defadvice のとる引数を下図に示す. POSITION などの [ ] 内の引数は省略可能である. [ ] 内の引数を省略する場合, 代わりに nil を置く必要はなく, 記述しないだけでよい. 上のコード例では, POSITION と ARGLIST が省略され, FLAG である disable を 指定している.

(defadvice FUNCTION (CLASS NAME [POSITION] [ARGLIST] [FLAG...]) ...)

FUNCTION	拡張したい関数名
CLASS	既存機能との実行順序
NAME	拡張機能の名前
POSITION	拡張機能間の実行順序
ARGLIST	拡張機能用の引数のリスト
FLAG	その他のフラグ

CLASS には, defadvice で定義したコードが, 既存のコードに対していつ実行されるかを指定する. CLASS のとる値には before・after・around などがある. before は拡張機能を既存のコードの前に, after は拡張機能を既存のコードの後に, それぞれ実行することを意味する. around は既存のコードを拡張機能のコードから任意のタイミングで呼び出すことを 意味する (後述の **ad-do-it の説明**を参照). around では既存のコードを実行しないようにもできる. 上の例でも, physical-line-mode の値によっては既存のコードは実行されない.

NAME は, 拡張機能を識別するための名前である. コード例では physical-line-move となっている. この名前を用いて, 定義した拡張機能を有効化・無効化する.

FLAG には protect・disable・activate などが指定可能である. activate フラグを指定すると, 後述する **アドバイスの有効化・活性化**がすぐさま行われる. disable フラグを指定すると, アドバイスは無効化された状態になる. disable を伴うアドバイスを実行させるためには, アドバイスの有効化・活性化を 行う必要がある. 詳しくは後述する. protect フラグを指定すると, エラーなどによる処理の中断が起こった場合でも, アドバイスで定義したコードは中略されずに実行されるようになる. 定義したアドバイスを常に有効にしたい場合は activate フラグを, マイナーモードでの利用など状況に応じて切り換えたい場合は disable フラグを 用いるとよい.

2.3. アドバイスの有効化および活性化

defadvice によって定義したアドバイスを実行するには, アドバイスを有効化 (enable-advice) し, その後活性化 (activate) させる必要がある.

アドバイスの有効化および無効化を行う基本的な関数は ad-enable-advice と ad-disable-advice である. また, アドバイスの活性化および非活性化を行う基本的な関数は ad-activate と ad-deactivate である.

```
(if physical-line-mode
    (ad-enable-advice 'line-move 'around 'physical-line-move)
  (ad-disable-advice 'line-move 'around 'physical-line-move))
(ad-activate 'line-move))
```

アドバイスの有効化および活性化

defadvice のフラグで activate を設定していた場合, defadvice を実行した時点で有効化および活性化は既に行われているので, ad-enable-advice・ad-activate を実行する必要はない. 逆に disable フラグを設定していた場合, 有効化も活性化も行われていないため ad-enable-advice・ad-activate の両方を実行する必要がある.

フラグをなにも設定していないアドバイスの状態は, 有効化はされているが活性化はされていない状態である. また, 一度定義したアドバイスの機能を変更した場合, その変更を反映させるには defadvice を再度実行するだけでなく, 再定義したアドバイスを もう一度活性化させる必要がある.

2.4. アドバイスで使用する機能

ad-do-it

ad-do-it はアドバイスの実行位置が around の場合に使用する. ad-do-it が書かれた場所に既存のコードが展開される. ad-do-it は関数ではないので (ad-do-it) と括弧をつけてはいけない.

ad-get-arg と ad-set-arg

ad-get-arg は既存の関数に与えられた引数を取得する機能である. 例えば第 1 引数を取得したい場合, (ad-get-arg 0) と実行すればよい. ただし ad-get-arg を使用せずに, 引数が代入される変数名を直接使用しても 引数を取得することは可能である.

引数の内容を変更するには ad-set-arg を用いる. 第 1 引数に変更したい引数の場所, 第 2 引数に値を与えて変更する. 例えば (ad-set-arg 1 "Taiyaki") である.

ad-return-value

既存の関数の実行後, 返り値は ad-return-value に代入される. 利用者は ad-return-value の値を取得・変更することで, 返り値を操作できる. アドバイスでの返り値が, そのまま関数の返り値にならないことに 注意が必要である.

```
(defun ad-return-value-test ()
  "Original")

(defadvice ad-return-value-test
  (around ad-return-value-test-advice activate)
  "An advice to test ad-return-value"
  ad-do-it
  (let ((ret ad-return-value))   ;; 既存関数からの返り値を ret に代入
    (setq ad-return-value        ;; アドバイスによる返り値の変更
          (concat ret "modified by advice")))
  "Adviced")                     ;; アドバイスでの返り値は無視される


(ad-return-value-test)
=> Original modified by advice
```

ad-return-value コード例:
最終行の "Original modified by advice" が返り値である

interactive-p

アドバイスの内容によっては, 利用者がコマンドとして実行した時のみ 有効化したいものもある. 例えば, カーソルを行頭に移動させる beginning-of-line コマンドの拡張を考える. beginning-of-line は利用者がコマンドとして実行するだけでなく, 他の機能から関数として実行される場合もある. アドバイスによる拡張は, コマンドとして利用者が実行した時のみ有効にした方が, 他の機能に悪影響を及ぼす心配が少なくなるので望ましい.

interactive-p を用いると, その関数が 利用者によってコマンドとして実行されたのか 他の関数から実行されたのか判別できる (下図). interactive-p はアドバイスでの使用に限定した関数ではないが アドバイスで用いると非常に有効である.

```
(defadvice beginning-of-line
  (around outline-beginning-of-line-advice disable)
  (if (and (interactive-p)
           (eq last-command this-command))
      (call-interactively 'backward-paragraph)
    ad-do-it))
```

interactive-p により, コマンドとして実行時のみ アドバイスを有効化

3. 外部ソフトウェアの利用

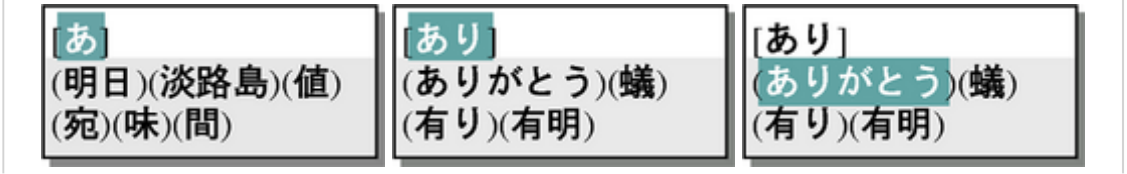
Emacs は外部のソフトウェアとデータ通信を行うことができる. その方法は, 別のソフトウェアを Emacs から新たに起動したり, あるいはサーバプロセスとのネットワーク通信により行う. 外部ソフトウェアを利用すれば, Emacs Lisp での再実装を避け, 既存のソフトウェア資産を有効活用できるだけでなく, 実行速度の向上も期待できる. また, あるソフトウェアの Emacs フロントエンドを作成する場合には 外部プロセスとのやりとりは当然必要である.

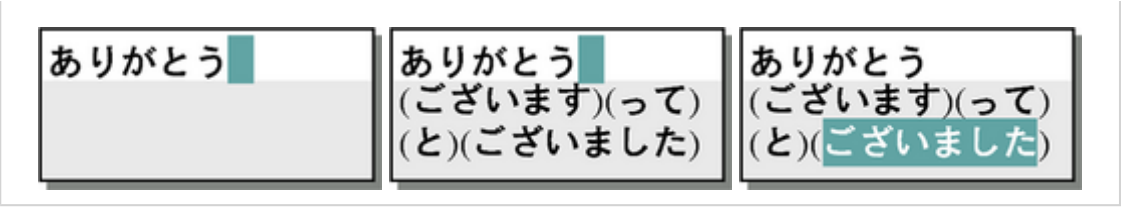
3.1. pobox-el: 日本語予測入力システム

pobox-el は, 増井が開発した予測入力方式である POBox [7,8] の Emacs クライアントである (下図). pobox-el は POBox による予測入力のほか, いくつかの入力方法を提供する.

pobox-el

POBox は変換候補を予測してインクリメンタルに提示する. そのため, 利用者はすべての文字を入力する前に, 目的の単語を入力できる. 下図は「ありがとうございました」と入力する例である. 利用者が「あり」と入力した時点で, POBox は「ありがとう」を第一候補として 提示している. そして利用者が「ありがとう」を選択すると, POBox は利用者の入力を待たずに, 単語間の頻度情報を基に次の単語を予測し, 候補を提示する.





POBox の動作例:
「ありがとうございました」と入力

pobox-el は, POBox サーバとのやりとりにネットワーク通信を行う. また KAKASI などの外部ソフトウェアを起動しての通信も行う. 次節から, 外部ソフトウェアの利用方法を議論する.

3.2. 外部ソフトウェアの利用

外部プロセスとのやりとりには 3 種類の方法がある.

- 同期プロセス通信
- 非同期プロセス通信
- ネットワーク通信

同期プロセス通信と非同期プロセス通信は, 外部のソフトウェアを実行し 標準入出力でデータのやりとりをする方法である. 同期と非同期の違いは, 同期は外部ソフトウェアの終了を待つが, 非同期では待たない点である. ネットワーク通信は, サーバマシンとポート番号を指定してサーバソフトウェアとの やりとりを行う方法である. ネットワーク通信は常に非同期で行われる.

3.3. 同期プロセス通信

同期プロセス通信は, 関数 call-process などにより行う.

call-process

以下の call-process のコード例は, "/etc/passwd" を標準入力とした grep コマンドの例である. 動作の内容は, シェル上での "grep -n zsh < /etc/passwd" と同様で, ファイル passwd の中から "zsh" という文字列を検索している.

```
(get-buffer-create " *result*")
(call-process "grep" "/etc/passwd" '(" *result*" nil) nil "-n" "zsh")
```

call-process のコード例

call-process は以下の引数をとる.

(call-process PROGRAM &optional INFILE DESTINATION DISPLAY &rest ARGS)

PROGRAM	"ls", "pwd" などの実行したいプログラム
INFILE	標準入力となるファイル名, nil の場合は /dev/null となる
DESTINATION	結果の出力先を示す. バッファやバッファを表す文字列などをとる. 値 が 0 の場合, 出力は無視し, 外部プロセスの終了を待たない.
DISPLAY	nil 以外の場合, 外部プロセスからの出力があるたびに結果を更新する.
ARGS	プログラムがとる引数

その他の同期プロセス通信

標準入力をファイルではなくバッファの内容にしたい場合, 関数 call-process-region を用いる. また, シェルを経由して実行したい場合, 関数 shell-command-to-string を用いる.

3.4. 非同期プロセス通信

非同期プロセス通信を用いると, 標準入力のデータを加工して出力する プログラムとのやりとりに便利である. 実際 pobox-el では, わかち書きソフトウェアである KAKASI とのやりとりに 非同期プロセス通信を行っている.

非同期プロセス通信を行うには, いくつかの処理が必要である. 第一に通信を行うプログラムを実行しなければならない. 次に, そのプログラムへデータを入力し, プログラムからの出力を取得する 必要がある. また, プログラムの状態を把握し状況に応じて操作する必要もある.

プロセスの開始

まずはじめに, 非同期プロセス通信を行うプログラムを実行する必要がある. プログラムの実行には, 関数 start-process を用いる.

```
(setq kakasi-process
      (start-process "KAKASI" " *kakasi*" "kakasi" "-w"))
```

start-process の例

関数 start-process は以下の引数をとる. また, start-process の返り値はプロセスオブジェクトである.

(start-process NAME BUFFER PROGRAM &rest ARGS)

NAME	プロセスの名前
BUFFER	プロセスに対応付けられるバッファ. バッファは文字列でもよい. また, 存在しないバッファを指定した場合, そのバッファは新たに作成される.
PROGRAM	実行したいプログラム
ARGS	実行したいプログラムの引数

コーディングシステムの設定

プロセスへの入出力を行う際に, EUC-JP・Shift_JIS などの 文字のコーディングシステムを設定する必要がある. 適切なコーディングシステムが設定されていないと, プロセス処理結果は文字化けしてしまう.

コーディングシステムの設定には set-process-coding-system を用いる.

```
(set-process-coding-system kakasi-process 'euc-japan 'euc-japan)
```

set-process-coding-system の例

set-process-coding-system の引数は順に「プロセス」, 「プロセスからの出力に対するコーディングシステム (decode)」「プロセスへの入力に対するコーディングシステム (encode)」である.

プロセスへのデータ送信

プロセスへ標準入力としてデータを送信するには, 関数 process-send-string と process-send-eof を用いる. 関数名が示すとおり process-send-string は通常の文字列を送信し, process-send-eof は終端記号である EOF (^D) を送信する.

```
(process-send-string kakasi-process "私の名前は小松です。\n")
(process-send-eof kakasi-process)
```

プロセスへのデータ送信例

process-send-string で送信する文字列に改行が付いていない場合, 恐らく process-send-eof を余分に実行する必要がある. 余分に実行しなければならない回数は, プロセスの内容に依存する.

プロセスからのデータ受信

プロセスからの処理結果は, start-process で指定したバッファに出力される. プロセスと対応付けされたバッファの取得は, 関数 process-buffer を用いても可能である.

処理結果の取得には, バッファを用いる以外にも プロセスフィルタを用いる方法がある. プロセスフィルタを用いると, プロセスの出力は指定された関数に渡され, バッファには出力されない. プロセスフィルタとなる関数は, プロセスからの出力があるたびに実行される.

プロセスフィルタの指定には, 関数 set-process-filter を用いる. set-process-filter は, プロセスとプロセスフィルタとなる関数の 2 つを引数に取る. プロセスフィルタとなる関数は, プロセスとそのプロセスからの出力文字列の 2 つの引数を取る. 下記のコードにプロセスフィルタの例を示す.

```
(defun kakasi-process-filter (process string)
  (message (format "Kakasi: %s" string)))

(set-process-filter kakasi-process 'kakasi-process-filter)
```

set-process-filter の例

プロセスフィルタを用いると start-process で指定したバッファには 出力されなくなる.

プロセスからのデータの明示的受信

プロセスからのデータは随時受信されるわけではない. プロセスからのデータは Emacs が利用者入力待ち状態の時など, 処理をしていないときに行われる.

プロセスからの出力データを特定のタイミングで受信したい場合, 関数 accept-process-output を用いる. accept-process-output を用いると, バッファやプロセスフィルタへの 出力待ちになっていた文字列をただちに出力する. また, プロセスからの出力がない場合は引数で指定した時間内で出力を待つ.

accept-process-output の取る引数は「プロセス」, プロセスからの出力を待つ「最大秒」と「最大ミリ秒」である.

```
(accept-process-output kakasi-process 1 0)
```

accept-process-output の例

プロセスの操作

プロセスの状態の把握や, 変更を行うための関数もいくつか用意されている.

(process-status PROCESS)

process-status は引数に与えたプロセスの状態を取得する関数である. 返り値は表に示すいずれかのシンボルとなる.

run	実行中
stop	一時停止中
exit	実行完了
signal	致命的なシグナルを受信
open	ネットワーク接続をオープン
closed	ネットワーク接続をクローズ
nil	プロセスが存在しない

(delete-process PROCESS)

delete-process は引数に与えたプロセスを強制終了させる.

(set-process-sentinel PROCESS SENTINEL)

set-process-sentinel はプロセスと監視関数を対応付ける. プロセスの状態が変わるたびに, 対応付けられた監視関数が実行される.

```
(defun kakasi-process-sentinel (process status)
  (message (format "%s: %S" (process-name process) status)))
(set-process-sentinel kakasi-process 'kakasi-process-sentinel)
```

set-process-sentinel の例

3.5. ネットワーク経由でのやりとり

ネットワークを通じてのやりとりの方法は, 非同期プロセス通信とほぼ同じである. ネットワーク通信では, 最初にプロセスを実行する代わりにネットワーク接続を行う. ネットワーク接続には, 関数 open-network-stream を用いる.

```
(setq pobox-network-process
      (open-network-stream "pobox" " *pobox*" "localhost" 1179))
```

open-network-stream の例

(open-network-stream NAME BUFFER HOST SERVICE &optional PROTOCOL)

NAME	ネットワークの名前
BUFFER	ネットワークに対応付けられるバッファ. バッファは文字列でもよい. また, 存在しないバッファを指定した場合, そのバッファは新たに作成される.
HOST	接続ホスト
SERVICE	ネットワークのサービス名もしくはポート番号
PROTOCOL	'tcp もしくは 'udp を指定可能. 省略した場合 'tcp となる.

3.6. 実際の非同期プロセスとのやりとり

非同期プロセスと通信を行う場合, 外部プロセスからのデータを適切に受信することが重要である. データ受信の失敗が思わぬ不具合を誘発することがよくある. ここでは例を挙げながら適切なデータの受信方法を議論する.

テストコード 1

テストコード 1 は "ls /" の実行結果を "*test*" バッファに出力し, "*test*" バッファのサイズを表示するプログラムである. 筆者の環境での結果は下図であり, バッファのサイズは "139" となる.

```
(defun process-test ()
  (let* ((buffer  "*test*")
         (process (start-process "ls-process" buffer "ls" "/")))
    (set-buffer buffer)
    (erase-buffer)
    (message (format "Size: %d" (buffer-size)))))
```

process-test テストコード 1

```
bin   dev  home        lib         mnt      root  tmp  var
boot  etc  initrd.img  lost+found  proc  sbin  usr  vmlinuz
```

```
Process ls-process finished
```

"ls /" の実行結果

テストコード 1 で表示されるバッファのサイズは "0" である. これは期待される結果ではない. バッファのサイズが "0" になる理由は, テストコード 1 では accept-process-output がないため, 実際のバッファへの出力は process-test 関数が終了した後になるからである.

テストコード 2

次に accept-process-output を追加した, テストコード 2 を検証する. テストコード 1 との違いは accept-process-output の有無だけである.

```
(defun process-test ()
  (let* ((buffer  "*test*")
         (process (start-process "ls-process" buffer "ls" "/")))
    (set-buffer buffer)
    (erase-buffer)
    ;; プロセスからの出力を明示的に受けはじめる
    (accept-process-output process 2 0)
    (message (format "Size: %d" (buffer-size)))))
```

process-test テストコード 2

テストコード 2 で表示されるバッファのサイズはまちまちである. 期待どおり "139" と表示される場合もあれば, "50" と表示される場合もある. この現象は「プロセスの出力」と「バッファサイズの計測」のタイミングに 依存する. accept-process-output の機能は, プロセスからの出力の明示的な受信を 開始することであり, 出力の終了は考慮しない. そこで, 出力の終了を待つようにする改善が必要である.

テストコード 3

テストコード 3 は, accept-process-output の後に 0.3 秒の待ち時間を追加した. バッファへの出力は 0.3 秒以内で終了するため, 表示されるバッファのサイズは 常に "139" になる.

```
(defun process-test ()
  (let* ((buffer  "*test*")
         (process (start-process "ls-process" buffer "ls" "/")))
    (set-buffer buffer)
    (erase-buffer)
    ;; プロセスからの出力を明示的に受けはじめる
    (accept-process-output process 2 0)
    ;; プロセスの終了を待つ (0.3 秒で終らない場合バグになる)
    (sleep-for 0.3)
    (message (format "Size: %d" (buffer-size)))))
```

process-test テストコード 3

バッファへの出力が 0.3 秒以内で終らない場合, 当然期待される結果には ならないので, 一定時間待つという方法はあまり良い方法ではない. そのため, バッファへの出力が完了したことを判断する別の方法が必要である.

プロセスの終了を待つ方法

テストコードのようにデータの出力が完了するとプロセスが終了する場合には, プロセスが終了したタイミングで, 出力結果を取得するのが確実である. 前述の set-process-sentinel によりプロセスの状態を監視する関数を 設定すれば, プロセスの終了は感知できる.

受信データに終端文字列を付ける方法

通常, 非同期プロセス通信をする場合は, Emacs と外部プロセスでデータ入出力が繰り返され, 外部プロセス

は終了しない場合が多い. そこで, データの受信の完了を判断するには多少工夫が必要になる.

例えば外部プロセスとして chasen を使用した場合, chasen は出力データの最後に "EOS" という文字列を付与する. このため, 受信完了の判別は最後の文字列が "EOS" であるか調べればよい.

しかし, すべての外部プロセスが明示的な終端文字列を付与している訳ではない. 明示的な終端文字列を付与しない外部プロセスを使用する場合, そのプロセスに応じた処置をしなければならない. 例えば kakasi や grep の場合であれば, Emacs から送信するデータの終端に 特殊な文字列を追加することで, 外部プロセスの付与する終端文字列の代わりになる. もちろん, 外部プロセスを修正して終端文字列を付与できれば, その方がよい.

4. リージョン情報の取得

Emacs での編集領域の指定方法には, リージョン範囲 (region: 下図左) と 矩形範囲 (rectangle: 下図右) の 2 つがある. さらにリージョン範囲にはいくつかのモードと状態が存在する. これらの範囲指定の状態を考慮することで, より使いやすいソフトウェアを作成できる.

リージョン範囲と矩形範囲の切り換え

4.1. sense-region: 矩形編集支援

sense-region はリージョンによる通常の範囲指定と, 箱型すなわち矩形による 範囲指定の統一的な操作を実現するソフトウェアである. 従来の矩形編集は「通常のリージョンとキー操作が違う」, 「選択された矩形範囲がハイライトされない」など, 直感的ではなかった. sense-region は矩形範囲とリージョン範囲の統一的な操作方法を提供する.

例えば, Emacs でのテキストの切り取り操作はリージョンの場合 "C-w", 矩形の場合 "C-x r k" である. どちらの場合も「選択範囲を切り取る」という 同じ編集機能であるが, コマンドは別になっている. そのため利用者は両方のコマンドを学習し, 状況に応じて使い分けなければ ならない.

sense-region はリージョン範囲と矩形範囲を "C-[space]" という操作により 交互に切り換える機能を提供する. sense-region により, 利用者は同じ編集機能を同じキー操作で行えるようになる. つまり, 上図の右の状態で "C-w" を実行すれば, 矩形を切り取ることができる. また, 選択範囲を矩形に切り換えた場合, sense-region ではハイライトも矩形に 切り換える. 上図に示した矩形のハイライトは sense-region によって実現されている.

4.2. リージョン状態の取得

sense-region を実現するには, リージョン範囲の情報を取得する必要がある. Emacs のリージョンを指定する方法は 2 つある. 1 つは常にリージョンが指定される方法であり, もう 1 つはリージョンの指定が オン・オフされる方法である. XEmacs にも同様の 2 つの方法がある, しかし取得する方法は Emacs と XEmacs で異なる.

Emacs のリージョンには 2 つのモードがある. ひとつは常にリージョンが有効になっているモードである. もうひとつはリージョンがハイライトされ, リージョンのオン・オフがはっきりしているモードである. このモードは, Emacs ではトランジエント (一時的) リージョンモードと呼ばれ, XEmacs では zmacs リージョンモードと呼ばれる. 本文では, この 2 つめのモードを XEmacs の場合も含め, トランジエント (リージョン) モードと呼ぶ.

リージョンの状態の取得

コマンドの動作を, リージョンの状態に応じて変化させれば コマンドの使い勝手は更に向上する.

例として, 小文字を大文字に変換するコマンド upcase-word (M-u) を挙げる. upcase-word はリージョンのオン・オフに関らず, カーソル位置から単語末までの 小文字を大文字に変換する. しかし, 利用者がリージョンをオンにして upcase-word を実行した場合, 利用者の意図した upcase-word の適用範囲はおそらくリージョン内であろう. upcase-region という, リージョン内を大文字にするコマンドも用意されているが 利用者が明示的に使い分ける必要があり, 非効率である. そこでリージョンの状態を取得し, 自動的に使い分けられるように改善したい. (ただし, XEmacs には幸いにして upcase-region-or-word というコマンドが 提供されている.)

リージョン状態の取得用コードを下に示す. Emacs と XEmacs ではリージョン状態の取得方法はまったく違う. トランジエントモードの判定に XEmacs では変数 zmacs-regions を使い, Emacs では変数 transient-mark-mode を使う. 更にトランジエントモードでリージョンがオンになっていることを確認するには XEmacs では関数 region-active-p を使い, Emacs では mark-active を使う.

```
(defun check-value (value)
  (and (boundp value)
       (symbol-value value)))

(defun transient-region-active-p ()
  (if (check-value 'running-xemacs)
;;;; Check for XEmacs
      (and (check-value 'zmacs-regions)
           (region-active-p)))
;;;; Check for Emacs
  (and (check-value 'transient-mark-mode)
       (check-value 'mark-active)))
```

リージョン状態の取得用コード

リージョン状態の継続

トランジエントモードのリージョンをオンにした状態でコマンドを実行した後, 状況に応じてリージョンをオフにしたり, オンのままにしたい場合がある. 例えば, テキストを処理するコマンドではリージョンをオフにしたいし, カーソルを移動させるコマンドではリージョンはオンのままにしたいだろう.

リージョン状態の継続方法も Emacs と XEmacs でまったく異なる. Emacs ではコマンドを実行した後, リージョンはオンのままである. Emacs でリージョンをオフにするためには, 変数 deactivate-mark の値を t にする必要がある. 逆に XEmacs ではコマンドを実行した後, リージョンはオフになる. XEmacs のリージョンをオンのままにするには, 変数 zmacs-region-stays の値を t にする必要がある.

リージョン状態を考慮する upcase

下図に, リージョンの状態を考慮した upcase のサンプルコードを示す.

```
(defun my-upcase ()
  (interactive)
  (if (transient-region-active-p)
      ;; トランジエントリージョンがオンの場合
      (call-interactively 'upcase-region)
    ;; それ以外の場合
    (call-interactively 'upcase-word))
  (or (check-value 'running-xemacs)
      (setq deactivate-mark t)))
```

リージョン状態を考慮する upcase

5. おわりに

筆者は, プログラミングからメールの読み書きにまで コンピュータによる生産作業のほとんどを, Emacs で行っている. そのため Emacs が便利になることは, すなわち生産効率の向上である. この信念のもと Emacs Lisp に傾倒し, 結果, 本業をおろそかにしてしまっている.

本論文で紹介したプログラミング技術は, そのような本末転倒の結晶といえる. この内容が Emacs の機能拡張の手助けになれば, 筆者の現実逃避もうかばれるというものである.

参考文献

1. **taiyaki.org/elisp**, *小松 弘幸*, http://taiyaki.org/elisp/
2. **GNU Emacs Lisp リファレンスマニュアル**, *Bil Lewis, Dan LaLiberte, Richard M. Stallman, the GNU Manual Group, 大木 敦雄 (訳)*, アスキー (ISBN: 4756134149), Apr 2000,
3. **GNU Emacsマニュアル20.6**, *Richard M. Stallman, 赤池 英夫 (訳), 大木 敦雄 (訳), 粕川 正充 (訳), 久野 靖 (訳), 鈴木 悦子 (訳), 高汐 一紀 (訳), 田中 聡 (訳)* , アスキー (ISBN: 4756134130), Apr 2000
4. **やさしい Emacs-Lisp 講座**, *広瀬 雄二*, カットシステム (ISBN: 4-906391-70-2), Jan 1999
5. **GNU Emacs拡張ガイド—Emacs Lispプログラミング**, *Bob Glickstein, 榎並 嗣智 (訳)*, オライリー・ジャパン (ISBN: 4-900900-19-2), Dec 1998
6. **Emacs Lisp UNIX短編シリーズ**, *青柳 龍也*, クオリティ (ISBN: 4769203918), Oct 1997
7. **POBox – 予測と曖昧検索にもとづく入力手法**, *増井 俊之*, http://www.csl.sony.co.jp/person/masui/OpenPOBox/
8. **An Efficient Text Input Method for Pen-based Computers.**, *Toshiyuki Masui*, In Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'98), Addison-Wesley, April 1998., pp.328-335.