Emacs 解剖学

Lecture 9

入力の補完

井田昌之、亀井信義

キー入力をさぼりたい

補完（completion）は，長いキー入力を助けてくれる機能です．マウスなどを使用して指示をするGUIの場合と違って，多くのコマンドシステムやプログラムの記述などの場合には，させたい操作を文字列的にタイプして指示をします．指示しようとする機能の名前を与えて，それを実行/記述させるわけです．この場合，もし名前が長かったり，あるいは複雑な構文をしている場合，それらをすべて覚えておいてそれを正確にタイプするのは至難の技になります．そして，それを利用者にいつも強いるのは良いシステムとは言えません．

そうすると選択は二つになります．短い名前にしてタイプしやすくするか，長い名前の入力を助ける機能を用意し，ユーザが困らないようにするか，です．前者の例には，たとえば，UNIXのコマンドなどをあげることができるでしょう．たとえば，moveのかわりにmv，remove-directoryのかわりにrmdirといったものです．これは，大変便利なことが多い半面，「しろうと」にはわかりにくいものになります．けれども頻繁に使う機能の場合には，簡単に入力できるので，大変使いやすいものでもあります．後者は，あまり頻繁には使わない機能を用意するような場合，あるいは初心者の利用が多い場合などに便利でしょう．名前を説明的にしたりできるからです．TSSのコマンドシステムでは入力している途中でエスケープキーをたたくと，次には何を入れればよいのか補助する情報を出してくれるようなものもありました．現在のUNIXのtcshなどのシェルでも補完機能があります．

Emacsには補完機能があります．これがあるので，Emacsがもっているさまざまな機能を呼び出すのに，完全にそのつづりを覚えていなくてもかまいません．また，ファイルを指す際のパス名も「決まり字」まで入力すればあとは補ってくれます．あるいは候補を出してくれてその中から選択すればいいようになっています．エディタでは，この連載にでてきたTECO，そしてそのOSであったITSなどはもともと名前が長くなるような概念がなく，短い入力ですみました．

今回は，この入力補完機能について取り上げます．

補完機能はどこから来たのか？

「補完機能はどこから始まったんだろう？」

どうしてもこういうことが気になって，多少調べてみました．たとえばVMSには，コマンド入力中にESCキーをたたくとその先には何を入れればよいかを助ける情報が出ます．VMSができたのは，1970年代後半です．ITSでは，どうもそうした機能はなかったようです．前々回のRichardの話のところでも述べましたが，ITSはハッカーマシンであって，初心者への補助機能などという概念はなかったようです．もちろん，TECOにもITS Emacsにも入力補完の概念はありません．

それで，ひとまず MIT コミュニティに聞いてみようと思い立ちました（それには Guy Steele 博士の助けを借りました．また，思いもかけずいろいろな人からの反応がありました．紙面を借りて謝意を表します）．

その結果，以下のようなことがわかりました．

1) Tenex OS には ESC によるコマンド行の補完機能はあった．(GLS より)
2) 20年前にバークレイで開発されていた SDS 940 の OS に，その原型があったのだと思う．また，Jed Donnelley の RATS system for RISOS には predictive command completion という機構があった．これは ESC キーをたたかず最少のキー入力だけで指示に対応しようとするものだ．70年代の初期のことだと思う．(Mike McMahon 氏)
3) CDC 6000 シリーズは NOS と KRONOS という OS があったけれど，それでもコンソール操作にはコマンド補完機能があった．70年代初期のことだが，Tenex より前とも言えないだろう．(Marty Connor 氏)
4) ITS の DDT にはパス名の補完機能があったと思う．けれども6文字ですんでしまうものの補完だから使うほどのことはなく，実際には使った覚えはない．(Carl Hoffman 氏)
5) XDS 940 にコマンド補完機能があって，Tenex はそれに強く影響を受けた．Dan Murphy に聞けばもっとよくわかるだろう．(Mark Crispin 氏)
6) Tenex のかなりのアイディアはバークレイタイムシェアリングシステムから来ている．(David A. Moon 氏)

これらを総合すると，コマンド行の補完機能は70年代初頭にはいくつかのシステムで存在していたようです．CDC では，コマンドを入力していくとその先の可能性を薄い色で提示してくれる，というような指摘もありました．

以上のことだけではなんとも言えませんが，このあたりのコマンドシステムでの補完は，XDS 940 に対するバークレイでの OS 開発に端を発しているように見えます．

エディタでの補完は?

結論から言って，肝心のこのことについては何もわかりませんでした．(多少言い訳になりますが，引越し引越しで資料がダンボールの中に埋もれてしまってお手上げということもあります．したがって，この原稿をボツにすることも考えたのですが，いろいろなことを知っているシニアの方々が，ひょっとして，何か話してくれるのではないか？ などとも考え，私の今のところ調べてわかったことを書いているわけです．ごめんなさい．)

もっとも確実に真相にせまるには，当然のことながら Richard Stallman に聞くのが一番です．「Tell me the origin of the GNU Emacs completion feature.」という電子メールを書きました．そうしたら早速返事が来ました．

短い返事でした．

「I don't remember. Some of the design may have come from Gosling Emacs.」(rms)

Gosling Emacs に起源があるだろう（かもしれない）と言っています．一般に彼とのやりとりのなかで，こういう表現をする場合は，本当に忘れてしまっているというより，若干整理をして説明するのが面倒だと思っている場合が多かったので，今回もそのように感じます．これ以上聞くのは，面と向かってゆっくりと聞けるときでないと無理です．Gosling Emacs は第3回に取り上げましたが，その資料もダンボールの中です．また，彼は Java でおおいそがしで，返事はもらえそうにありません．Java の件で先日会ったときも大変なスケジュールの中でした．それで，もし，私自身がフォローするなら，後日改めて取り上げるしかないと思っています．

以上のようなことが補完についてのイントロです．Emacs での補完機能の説明というのは，結局 Emacs Lisp でどうやってその機能を利用できるかということにつきます．

Emacs の補完機能については，そのことを詳しく調べている亀井さんに登場してもらい，彼にバトンタッチします．

(いだ まさゆき 青山学院大学 国際政治経済学部)

```
---------- Buffer: *Completions* ----------
Click mouse-2 on a completion to select it.
In this buffer, type RET to select the completion near point.

Possible completions are:
find-alternate-file                find-dired
find-file                          find-file-other-frame
find-file-other-window             find-file-read-only
find-file-read-only-other-frame    find-file-read-only-other-window
find-grep-dired                    find-name-dired
find-tag                           find-tag-noselect
find-tag-other-frame               find-tag-other-window
find-tag-regexp                    finder-by-keyword
---------- Buffer: *Completions* ----------
```

図1 'find' で始まるコマンド補完候補一覧表示

Emacs での補完機能

Emacs の特徴の一つに補完機能があります．補完は，ある名前の省略形を与えたときに，名前の残りの部分を埋めてくれる機能です．Emacs では，随所にこの補完機能を使っているため，長ったらしい名前（ファイル名，バッファ名，関数名，変数名など）をいちいち全文字タイプしないですむようになっています．数文字タイプしては，残りを補完することで目的の文字列を得る，という使い方ができるわけです．

たとえば Mule-2.3 において，バッファ '*scratch*' で，'find-read-file-coding-system-from-file-variables' などという，恐ろしく長い名前の関数（48文字！）を使いたいとします．これを1文字も間違えずに入力するのは大変なことです．しかし，こんなときに補完が使えれば，'f i n d - r M-TAB' と操作するだけで，'find-read-file-coding-system-from-file-variables' と入力したことになります．

また，候補が複数ある場合は，それらの一覧を表示させることもできます．たとえば，ミニバッファで 'M-x f i n d ?' と入力すると，'*Completions*' という名前のバッファが自動的に表示され，そこに 'find' で始まるコマンドの一覧が表示されていると思います（図1）．

補完を対話入力以外で使用することはありませんから，必然的にユーザのミニバッファ入力と深い関連をもつことになります．

説明の方法には，

- 補完機能を構成するプリミティブから高度な関数へと説明していく方法
- ユーザから見たミニバッファでの補完機能からプリミティブへ向けて説明していく方法

という二つの方法があります．どちらの切り口から説明していこうかと考えたのですが，「羽か鉛か？」の質問に，「羽」が選ばれたので（謎），今回はプリミティブの側から説明をしていきたいと思います．

なおここでは，

- Mule Version 2.3 (SUETSUMUHANA) of 1995.7.24
- GNU Emacs 19.28.1 (i486-JE-linux, X toolkit) of Mon Sep 4 1995 on Roy

を使用しました．

補完プリミティブ

Emacs が提供する補完プリミティブは，次の二つです．

- Function : try-completion STRING COLLECTION &optional PREDICATE
- Function : all-completions STRING COLLECTION &optional PREDICATE NOSPACE

'try-completion' は一つの補完結果を得るために使用し，'all-completions' はすべての一致（match）する補完候補（possible completion）を得るために使用します†．

† 'all-completions' の第4引数 NOSPACE は，Emacs 19.29 で追加された．

表1　補完プリミティブに与える要素

<table>
<tr><th colspan="2">要　素</th><th>意　味</th></tr>
<tr><td>入力文字列</td><td>引数 STRING</td><td>補完する文字列.</td></tr>
<tr><td>補完候補の集合</td><td>引数 COLLECTION</td><td>入力文字列の一致対象．この集合の各要素に，入力文字列の一致を試みる．
集合の与え方には，次の3通りがある．
<table>
<tr><th>タイプ</th><th>働　き</th></tr>
<tr><td>obarray</td><td>各要素の印字名（printed name）が補完候補.</td></tr>
<tr><td>alist</td><td>各要素の CAR 部が補完候補.
CDR 部は各要素の付加情報として利用可能.</td></tr>
<tr><td>関数であるシンボル</td><td>補完自体を行なう関数．入力文字列に応じて，動的に補完候補を生成/決定するために使用する.
これによって，プログラム補完（programmed completion）と呼ぶ独自の補完方法を定義することもできる.</td></tr>
</table></td></tr>
<tr><td>正規表現フィルタ</td><td>変数'completion-regexp-list'</td><td>補完候補を絞り込むフィルタ．補完候補が満たすべき正規表現のリスト.
'nil' ならば，絞込みはしない.</td></tr>
<tr><td>述語フィルタ</td><td>引数 PREDICATE</td><td>補完候補を絞り込むフィルタ.
補完候補はこの述語の検査を通過しなければならない.
'nil' ならば，絞込みはしない.
述語は補完候補の集合が obarray ならば，その要素のシンボルを，alist ならば，その要素の cons セルを一つの引数として受け取る.</td></tr>
</table>

表2　'try-completion' の戻り値

戻り値	意　味	実行例
't'	完全一致（exact match）の場合. 入力文字列が，その唯一の一致補完候補と等しかった場合.	図5
最長共通部分文字列 (the longest common substring)	一致補完候補ありの場合. すべての一致補完候補に共通な冒頭部分の文字列.	図6
'nil'	一致補完候補なしの場合.	図7

これら補完プリミティブの処理の流れを表わしたのが，図2です†.

ここで，補完プリミティブに与える要素は，表1のようになります.

いずれのプリミティブも，補完候補の集合 COLLECTION から入力文字列 STRING と一致する要素を選び出します．これを一致した補完候補と言います．各要素の冒頭部分が入力文字列に等しければ一致とみなすのです．ここで，変数 'completion-ignore-case' が非 'nil' ならば，文字列比較において大文字/小文字の区別をしません.

'try-completion' の戻り値は，表2のようになりま

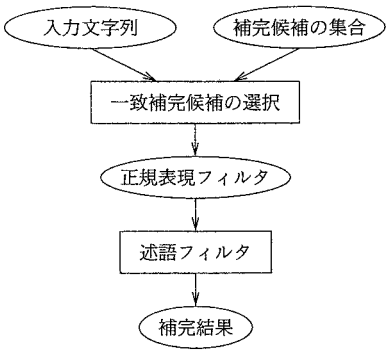

図2　補完プリミティブ処理の流れ

† このあたりの処理の流れについては，'src/minibuf.c' の 'try-completion' や 'all-completions' の定義を見ればわかる.

```
(setq   pico-table
        (list
         (cons  "PI-CO"     (cons   "桜文鳥"          "F"))
         (cons  "POPI-"     (cons   "桜文鳥"          "F"))
         (cons  "FU-CHAN"   (cons   "セキセイインコ"  "M"))
         (cons  "CHIBI"     (cons   "セキセイインコ"  "M"))
         (cons  "MA-CHAN"   (cons   "セキセイインコ"  "F"))
         (cons  "CHI-CHAN"  (cons   "セキセイインコ"  "F"))
         (cons  "RO-RI-"    (cons   "カナリア"        "M"))
         (cons  "RO-RA"     (cons   "カナリア"        "F"))
         (cons  "PICO"      (cons   "ツキノワインコ"  "M"))
         (cons  "PIPPI"     (cons   "コザクラインコ"  "F"))
         (cons  "PICOPON"   (cons   "?"               "F"))))
```

図3　補完候補の集合

```
(defun  ftest   (cell)
  (string=  (cddr   cell)   "F"))
```

図4　述語フィルタ用関数

```
;;  "PIPPI"の一致補完候補は"PIPPI"ただ一つであり，
;;  それは入力文字列自身と等しいので，完全一致．
(try-completion "PIPPI" pico-table)
⇒   t
```

図5　完全一致の例

```
;;  "PICO"の一致補完候補には，"PICO"と"PICOPON"の
;;  二つがあるので，両者の共通部分である"PICO"に
;;  なる．完全一致ではない．
(try-completion "PICO" pico-table)
⇒   "PICO"

;;  今度は述語フィルタによって，一致補完候補が
;;  "PICOPON"ただ一つになったが，入力文字列と
;;  その一致補完候補とは等しくないので，
;;  やはり完全一致ではない．
(try-completion "PICO" pico-table 'ftest)
⇒   "PICOPON"
```

図6　一致補完候補ありの例

```
;;  {"PIYO"で始まる補完候補は存在しない．}
(try-completion "PIYO" pico-table)
⇒   nil

;;  "FU-CHAN"は唯一の一致補完候補であったが，
;;  述語フィルタによって除去された．
(try-completion "FU-CHAN" pico-table 'ftest)
⇒   nil
```

図7　一致補完候補なしの例

表3　'all-completions' の戻り値

戻り値	意　味	実行例
文字列のリスト	一致補完候補ありの場合	図8
'nil'	一致補完候補なしの場合	図9

```
;;  複数の補完候補が一致する場合．
(all-completions "PICO" pico-table)
⇒   ("PICO" "PICOPON")

;;  上の例と同じだが，述語フィルタによって
;;  "PICO"は除去された．
(all-completions "PICO" pico-table 'ftest)
⇒   ("PICOPON")
```

図8　一致補完候補ありの例

```
;;  "PIYO"に一致する補完候補はない．
(all-completions "PIYO" pico-table)
⇒   nil
```

図9　一致補完候補なしの例

す．'all-completions' の戻り値は，表3のようになります．

図表中の実行例において，補完候補の集合は，図3のように alist で定義し，述語フィルタ用関数は，図4のように定義しています．

表4　プログラム補完のフラグに対応する，期待される振舞いと戻り値

フラグ	期待される振舞い	戻り値	
'nil'	'try-completion'	't'	完全一致
		文字列	最長共通部分文字列
		'nil'	一致補完候補なし
'lambda'	完全一致の検査 ('assq', 'assoc' など)	't'	完全一致
		'nil'	非完全一致
't'	'all-completions'	一致補完候補のリスト	

プログラム補完 (Programmed Completion)

あらかじめ補完候補の集合を用意することが困難な場合，入力文字列を基にして補完候補の集合を動的に求めることができます．これをプログラム補完といいます．補完を行なう関数の，引数補完候補の集合に，自作関数のシンボルを渡して使います[†1]．

自作関数は，三つの引数「入力文字列，述語フィルタ，フラグ」を受け付けるように作成します．このフラグの値に応じた三つのモードで動作しなければなりません．

フラグの値と期待される動作は，表4のとおりです．

プログラム補完の実行例を，図10に示します．

ミニバッファ入力

プリミティブ

Emacs がユーザから入力を文字列で受け取る場合は，ミニバッファを使います．ミニバッファからの入力を受け取るには，'interactive' 宣言を使用するのが普通ですが，直接ミニバッファ入力関数を使用することも可能です．

†1　Emacs は，プログラム補完をファイル名補完に使用している．'src/fileio.c' の高水準補完関数 'read-file-name' で使用する，'read-file-name-internal' がそれである．ここでは，述語フィルタに相当する引数を，ディレクトリ名を渡すために流用している．

†2　実装は若干違うが，意味論的にはこう言い切ってよいと思う．'src/minibuf.c' を参照のこと．

†3　'src/callint.c'

†4　'minibuffer-complete-word', 'minibuffer-complete'．いずれも，'try-completion' を利用．

†5　'minibuffer-completion-help'．'all-completions' 利用．

表5　ミニバッファ入力関数

関　数	機　能
'read-from-minibuffer'	ミニバッファから文字列を読み込む
'completing-read'	ミニバッファから補完付きで文字列を読み込む

ミニバッファ入力で使用するプリミティブ関数は，表5の二つです[†2]．

Emacs は，文字列入力，ファイル名入力，バッファ名入力，コマンド名入力などといった使用目的に特化したミニバッファ入力コマンドも提供しています．これら高水準補完関数は，上記プリミティブをラップ (wrap) することで，作られています．

'interactive' 宣言でコード文字を使用した場合には，それら高水準補完関数が，'call-interactively'[†3] の中から呼び出されるようになっています．これは，Emacs のコマンドの引数をミニバッファ入力する場合に使用されます．

したがって，標準の Emacs が補完を使ってミニバッファ入力を行なう場合，いずれにせよ，最終的に 'completing-read' が呼ばれます．

余談ですが，Emacs のミニバッファの入力モードの変更方法は，キーマップを切り替えることで実現しています．つまり，キーマップを切り替えることで，キーに割り当てられているコマンドが変わるので，ユーザからはミニバッファの入力モードが変わったように見えるのです．

補完付きミニバッファ入力 (Minibuffer Input with Completion)

Emacs における補完付きミニバッファ入力を理解するには，結局 'completing-read' を理解すればよいことになります．

その前に，補完付きミニバッファ入力には，3種類あることを説明します．補完付きミニバッファ入力では，SPC や TAB によって，文字列の補完を行なうことができます[†4]．また，? によって，一致補完候補の一覧を別バッファに見ることができます[†5]．

補完付きミニバッファ入力は，その入力文字列の確定動作，すなわち，LFD あるいは RET の働きによって，表6の3通りに分類されます．

'completing-read' のインタフェースは，次のとお

```
;; これが，自作関数です．
;; 入力文字列に，拡張子をつけたものを，補完候補の集合とします．
(defun zic-comp-func (input pred type)
  (let*
      ((idx (string-match "¥¥.[^.]+$" input))
       (substr (if idx
                   (substring input 0 idx)
                 input))
       (zic-table
        (list (cons (concat substr ".bdf") "X")
              (cons (concat substr ".snf") "X")
              (cons (concat substr ".pcf") "X")
              (cons (concat substr ".fon") "WIN")
              (cons (concat substr ".fnt") "DOSV"))))

    (cond
     ((eq type 'lambda) (if (assoc input zic-table) t))
     ((eq type t) (all-completions input zic-table pred))
     (t (try-completion input zic-table pred))
     )))

;; これが，述語フィルタ．
(defun zic-test (s)
  (string= (cdr s) "X"))

;; ------------------------------------------------------------
;; 述語フィルタなしで，補完します．
(try-completion "name" 'zic-comp-func)
⇒ "name."

;; 同じく，補完候補のリストを見ます．
(all-completions "name" 'zic-comp-func)
⇒ ("name.bdf" "name.snf" "name.pcf" "name.fon" "name.fnt")

;; 述語フィルタをかけて，補完します．
(try-completion "name" 'zic-comp-func 'zic-test)
⇒ "name."

;; 同じく，補完候補のリストを見ます．
(all-completions "name" 'zic-comp-func 'zic-test)
⇒ ("name.bdf" "name.snf" "name.pcf")

;; 入力文字列に，拡張子の一部を書きます．
(try-completion "name.f" 'zic-comp-func)
⇒ "name.f"

;; 同じく，補完候補のリストを見ます．
(all-completions "name.f" 'zic-comp-func)
⇒ ("name.fon" "name.fnt")
```

図10　プログラム補完の実行例

りです．

- Function : completing-read PROMPT COLLECTION &optional PREDICATE REQUIRE-MATCH INITIAL HIST

ここで，'completing-read' に与える引数は，表7のようになります．

その他の補完

標準の Emacs が提供するのとは少し違う，補完の仕組みもあります．研究してみると面白いでしょう．

次の二つを紹介します．

- 部分補完（partial completion）
- 段階的補完（incremental completion）

表6　補完付きミニバッファ入力の3タイプ

補完タイプ	使用キーマップ	RET キーの働き	実行例
任意補完 (permissive completion)	'minibuffer-local-completion-map'	補完可能だが，結果は補完候補でなくてもよい． （ミニバッファの内容をそのままにして，ミニバッファを抜ける）	図11
厳密補完 (strict completion)	'minibuffer-local-must-match-map' （変数 'minibuffer-completion-confirm' が 'nil' のとき）	結果は補完候補のいずれかでなければならない． （ミニバッファの内容を補完した結果が，補完候補に等しければミニバッファを抜ける）	図12
確認補完 (cautious completion)	'minibuffer-local-must-match-map' （変数 'minibuffer-completion-confirm' が非 'nil' のとき）	厳密補完の一種． （ミニバッファの内容がすでに補完候補に等しければ，そのままミニバッファを抜ける．あるいは，ミニバッファの内容を補完した結果が，補完候補に等しければ確認を求めてくる）	図13 図14

```
(completing-read    "Name: " pico-table nil nil "C")
;;  評価すると，

---------- Buffer: Minibuffer ----------
Name: C■
---------- Buffer: Minibuffer ----------

;;  RETをタイプ．
⇒   "C"
```

図11　任意補完の例

```
(completing-read    "Name: " pico-table nil t "C")
;;  評価すると，

---------- Buffer: Minibuffer ----------
Name: C■
---------- Buffer: Minibuffer ----------

;;  RETをタイプ．

---------- Buffer: Minibuffer ----------
Name: CHI■
---------- Buffer: Minibuffer ----------

;;  一致補完候補が複数あるので，ミニバッファを抜けることができない．
;;  確認のため，?をタイプ．

---------- Buffer: *Completions* ----------
Click mouse-2 on a completion to select it.
In this buffer, type RET to select the completion near point.

Possible completions are:
CHI-CHAN                        CHIBI
---------- Buffer: *Completions* ----------

;;  "CHIBI"を選択すべく，`B' とタイプし，

---------- Buffer: Minibuffer ----------
Name: CHIB■
---------- Buffer: Minibuffer ----------

;;  RETをタイプ．
⇒   "CHIBI"
```

図12　厳密補完の例

```
(completing-read    "Name: " pico-table nil 'lambda "C")
;;  評価すると,

---------- Buffer: Minibuffer ----------
Name: C■
---------- Buffer: Minibuffer ----------

;;  RETをタイプ.

---------- Buffer: Minibuffer ----------
Name: CHI■
---------- Buffer: Minibuffer ----------

;;  一致補完候補が複数あるので, ミニバッファを抜けることができない.
;;  "CHIBI"を選択すべく, `B'とタイプし,

---------- Buffer: Minibuffer ----------
Name: CHIB■
---------- Buffer: Minibuffer ----------

;;  RETをタイプ.

---------- Buffer: Minibuffer ----------
Name: CHIBI■(Confirm)
---------- Buffer: Minibuffer ----------

;;  "(Confirm)"という確認表示が現われたので,
;;  さらに, RETをタイプ.
⇒   "CHIBI"
```

図 13　確認補完の例(1)

```
(completing-read    "Name: " pico-table nil 'lambda "C")
;;  評価すると,

---------- Buffer: Minibuffer ----------
Name: C■
---------- Buffer: Minibuffer ----------

;;  RETをタイプ.

---------- Buffer: Minibuffer ----------
Name: CHI■
---------- Buffer: Minibuffer ----------

;;  一致補完候補が複数あるので, ミニバッファを抜けることができない.
;;  "CHIBI"を選択すべく, `B'とタイプし,

---------- Buffer: Minibuffer ----------
Name: CHIB■
---------- Buffer: Minibuffer ----------

;;  TABをタイプ.

---------- Buffer: Minibuffer ----------
Name: CHIBI■
---------- Buffer: Minibuffer ----------

;;  さらに, RETをタイプ.
⇒   "CHIBI"

;;  すでに補完候補に一致していたので,
;;  "(Confirm)"という確認要求はなかった.
```

図 14　確認補完の例(2)

表7 'completing-read' の引数

引 数	意 味
PROMPT	ミニバッファに表示するプロンプト.
COLLECTION	補完候補の集合. 'try-completion' に同じ.
PREDICATE	述語フィルタ. 'try-completion' に同じ.
REQUIRE-MATCH	補完動作の指定. 値 / 動作 'nil' / 任意補完 't' / 厳密補完 それ以外 / 確認補完
INITIAL	初期入力. ミニバッファの初期入力.
HIST	履歴リスト変数. ミニバッファ履歴で使用する (今回は説明を省略).

```
;;  `M-x l SPC l'とタイプする.

---------- Buffer: Minibuffer ----------
M-x l-l■
---------- Buffer: Minibuffer ----------

;;  TABによって, (部分)補完する.

---------- Buffer: Minibuffer ----------
M-x lo■-library
---------- Buffer: Minibuffer ----------

;;  `?'によって, 補完候補を見る.

---------- Buffer: *Completions* ----------
Click mouse-2 on a completion to select it.
In this buffer, type RET to select the completion near point.

Possible completions are:
Load-library                    locate-library
---------- Buffer: *Completions* ----------

;;  `a'をタイプし,

---------- Buffer: Minibuffer ----------
M-x loa■-library
---------- Buffer: Minibuffer ----------

;;  さらに, TABによって補完する.

---------- Buffer: Minibuffer ----------
M-x load-library■
---------- Buffer: Minibuffer ----------

;;  完全に補完された.
```

図 15 部分補完の実行例

```
;;  `M-x l' とタイプ.

---------- Buffer: Minibuffer ----------
M-x l■{ocate-library, isp-indent-line, ist-tags, ist-buffers, ist-faes-
---------- Buffer: Minibuffer ----------

;;  このように, `l' で始まるコマンド名が表示される.
;;  さらに1字, `o' とタイプ.

---------- Buffer: Minibuffer ----------
M-x lo■{cate-library, wer-frame, ad-file, ad-library, cal-unset-key, cal-
---------- Buffer: Minibuffer ----------

;;  `lo' で始まるコマンド名に, 候補が絞られる.
;;  さらに, `a d' とタイプ.

---------- Buffer: Minibuffer ----------
M-x load■(-){file, library}
---------- Buffer: Minibuffer ----------

;;  さらに, `SPC l' とタイプ.

---------- Buffer: Minibuffer ----------
M-x load-l■(ibrary) [Matched]
---------- Buffer: Minibuffer ----------
```

図 16　段階的補完の実行例

部分補完

部分補完とは，デリミタで区切られた単語の列がある場合に，各単語がそれぞれ独立に補完できるというものです．図 15 が，その実行例です．

部分補完は，ライブラリ 'complete' をロードすることで，利用可能になります．

段階的補完

ミニバッファからコマンド名入力を行なう際に，今入力している文字列がどんなコマンドに補完され得るかを，入力 1 文字ごとに，つねに表示し続ける補完です．例を図 16 に示します．

段階的補完は，ライブラリ 'icomplete' をロードすることで，利用可能になります．Emacs 19.28 でも利用可能ですが，振舞いが副モードのようになった Emacs 19.29 以降で利用するのが望ましいでしょう．

（かめい のぶよし）