canlisp マニュアル

今　　昭
kon@d1.bs2.mt.nec.co.jp

— 第 2 版 —
『かんな』 Version 3.2
1994 年 4 月 14 日

はじめに

本文書は、『かんな』のカスタマイズファイルのパーサである canlisp の言語定義を与えるものである。

この文書では、.canna ファイルに記述できる言語仕様を lisp 言語の立場から述べる。.canna ファイルのパーサである canlisp インタプリタは独立したコマンドとしても準備されている (canlisp と言う名前のコマンドである)。本言語仕様の確認をしたい場合は canlisp コマンドで確認するとよい。

なお、 canlisp で使われる『かんな』制御の各パラメタの詳細な説明は行わない。各パラメタの詳細については、『かんな』マニュアルを参照されたい。

本文書を読むにあたっては一通りの lisp 言語の知識が必要となる。本文書の他に以下の lisp マニュアルを参照されたい。

- GNU Emacs Lisp Manual
- Common Lisp the Language

なお、本マニュアル作成に当たって菱田修氏には lisp 言語の視点より念入りな査読と、貴重なご意見をいただきました。ここに謹んで感謝の意を表します。

目次

1 canlisp の系統

canlisp は MacLisp 系の lisp 言語である。いやまて、 MacLisp 系と言うのは canlisp のような超小型の lisp が称するのはちょっと大胆である。が、とりあえず defun 構文や defmacro に関しては MacLisp および Franz Lisp に見られる方式を用いている。

canlisp の言語処理を lisp 言語の視点から言うと、深い束縛を持ち、レキシカルスコープを持った言語と言うことになる。レキシカルスコープは Zeta Lisp や Common Lisp で見られるスコーピングルールで、コンパイラに主眼を置いたものであるが、 canlisp はインタプリタであるにもかかわらず採用した。

とは言っても、たかだか日本語入力システムのカスタマイズファイルパーサであるので、それほどたいしたことはしていない。 canlisp の変数は大域で定義されるか、ローカルバインディングがなされるかで、いわゆるスペシャル変数 (`(declare (special x))` の x のような変数) は用意しない。

データの内部表現に関しては Utilisp の影響を受けている。データ表現はデータを指すポインタにデータの種別を判別するためのタグを備えるポインタタグ方式を採用した。またガーベジコレクションも Utilisp と同じコピー方式を用いている。 Utilisp では高速化のためにこれらの手法が取られているが、 canlisp では単に処理が楽だからと言う理由で採用した。

ところで、 canlisp では文字型データの記述法や global-set-key などのキーワードが GNU Emacs Lisp の影響を受けている。ただし、細部に違いがあるので注意されたい。例えば、 GNU Emacs Lisp ではコントロールバックスラッシュ (^\) を表すのに、

```
?\C-\\
```

と記述するが、 canlisp では、

```
?\C-\
```

と記述する。また、ローカルキーマップにおいてキーに機能を割り付けるのに GNU Emacs Lisp では define-key を用いるが canlisp では set-key を用いる。

canlisp を設計するにあたって、できるだけ処理系が小さくなるように心掛けた。と言うのは、 canlisp はそもそも『かんな』のパーサにすぎず、それ以上の何者でもないからである。この方針にしたがってデータ型も最少のものを準備するにとどめた。また、組み込み関数も非常に基本的なものに押さえ、主にカスタマイズ処理にかかわるものに主眼を置いた。

そのため、例えば FUNARG 問題は全く解決していない。だいたいにして、 eval、 apply、 funcall を準備していない。著者は upward/downward の両方を解決する closure やそれをベースとした、手続きをも組み込んだオブジェクトなどに興味があるが、 canlisp ではそこまで踏み込まない。

canlisp は lisp reader を含めて 4000 行 程度の プログラム である。 canlisp の処理系は ファイル lib/canna/lisp.[hc] にて定義されている。

2 使用する型

2.1 型の種類

canlisp が使用するデータの型は以下に分類される。

整数型	`-1, 0, 1, 2, ... 100, 101, ...`
文字型	`?A, ?B, ... ?\C-a, ?\C-b, ...`

```
文字列型        "defalut.kp", "\Up", ...
NIL 型           nil
シンボル型      auto, foo, bar, ...
CONS 型          (a . b), ("a" "b"), ...
```

文字型の内部表現は整数型と全く同じであり、文字型が存在するのはユーザの便宜のためである。文字型のデータ記述は lisp reader により整数型データに置き換えられる。

整数は 24 ビットで表される符号付き整数である。

Common Lisp では nil はシンボル型データであるが、 canlisp では nil はシンボルではないので注意。

2.2 型の分類

上記型を直接分類する述語 (predicate) として Common Lisp などには、 null、 numberp、 symbolp、 stringp、 consp が存在する。

また、複合的な分類述語として、 atom、 listp がある。 atom はこれ以上分割できないアトミック (原子的) データを示しており、 canlisp では、 CONS 型以外のすべてのデータ型があてはまる。

listp は種々のデータの列記を (リスト) しているデータであり、 canlisp では NIL 型と CONS 型があてはまる。

canlisp では上記述語のうち実装されているのは null と atom のみである。

3 canlisp reader (シンタックス)

lisp はシンタックスがない言語といわれているが、 reader の規則と言う最低限のシンタックスはある。 canlisp reader の仕様を以下に示す。

なお、 canlisp は大文字小文字を区別する。

3.1 スペース

スペース (空白文字) として、 space 文字 (' ')、タブ ('\t')、改行 ('\n')、復帰 ('\r') が用いられる。

3.2 かっこ

lisp の基本的な型である CONS 型は開きかっこと閉じかっこで囲まれて表現される。 CONS 型を表すかっことしては、丸かっこが用いられる。

3.3 コメント

セミコロン (;) から行末まではコメントとして読み飛ばされる。

3.4 区切り文字とトークン

スペース (' '、'\t'、'\n'、'\r')、かっこ ('('、')')、コメント開始文字 (';') はトークンを切り出す区切り文字として振る舞う。

トークンは区切り文字で囲まれた文字列である。 canlisp の reader はトークンを切り出し、そのトークンをデータの最少単位として処理する。

```
abcd(     →  abcd がトークンとして切り出される。
bar)      →  bar          〃
foo;bar   →  foo          〃
```

ただし、文字型、文字列型の場合は、区切り文字およびトークンの処理はこの限りではない。文字型、文字列型の場合の処理については、 3.7文字型、 3.8文字列型の項を参照されたい。

3.5 エスケープ文字

バックスペース (\) はエスケープ文字として種々の目的に用いられる。具体的には文字、文字列およびシンボルの項を参照のこと。

3.6 数値型

数値は数字の続きとして表現される。 canlisp では数値型は整数しか取り扱われないので小数点は末尾にしか許されない。

```
123     →  数値
123.    →  数値
123.5   →  シンボル
```

また、マイナスの数値を表すハイフン (−) が符号なし数値の先頭に付く場合にも数値として取り扱われる。ハイフンだけのトークンは数値ではない。

```
-123    →  数値
-123.   →  数値
-       →  シンボル
--123   →  シンボル
```

3.7 文字型

3.7.1 単純な文字表現

文字はクエスチョンマーク (?) に続くデータとして用いられる。クエスチョンマークに続く文字がエスケープ文字 (\) 以外の文字であれば、そのデータは文字データとして用いられる。

```
?A      →  'A'
?(      →  '('
?;      →  ';'
```

C 言語では漢字等の日本語文字は 1 文字としては扱えないが、 canlisp では漢字なども一文字として扱える[1]。

[1] Version 3.2 から

?あ　　→　「あ」と言う文字

クエスチョンマークがトークンの先頭でないときは文字データではなく、シンボルとして扱われることに注意されたい。

Why?　　→　Why? というシンボル。

3.7.2　エスケープ文字を伴うキーワード表現

クエスチョンマークの後ろにエスケープ文字が続く場合には、エスケープ文字から後ろを以下のように解釈する。

エスケープ文字の後ろが以下のキーワードの場合、表内に表されるとおり解釈される。表内のキーワードにおいて、大文字と小文字の違いは区別されるので注意されたい。

```
\Space    空白文字 (' ')
\Escape   エスケープキーを表す文字 ('\033')
\Tab      タブ文字 ('\t')
\Nfer     canlisp の中で、無変換キーを表す文字 (CANNA_KEY_Nfer)
\Xfer     canlisp の中で、変換キーを表す文字 (CANNA_KEY_Xfer)
\Backspace バックスペースキーを表す文字 ('\b')
\Delete   デリートキーを表す文字 ('\177')
\Insert   canlisp の中で、インサートキーを表す文字 (CANNA_KEY_Insert)
\Rollup   canlisp の中で、ロールアップキーを表す文字 (CANNA_KEY_Rollup)
\Rolldown canlisp の中で、ロールダウンキーを表す文字 (CANNA_KEY_Rolldown)
\Up       canlisp の中で、上矢印キーを表す文字 (CANNA_KEY_Up)
\Left     canlisp の中で、左矢印キーを表す文字 (CANNA_KEY_Left)
\Right    canlisp の中で、右矢印キーを表す文字 (CANNA_KEY_Right)
\Down     canlisp の中で、下矢印キーを表す文字 (CANNA_KEY_Down)
\Home     canlisp の中で、ホームキーを表す文字 (CANNA_KEY_Home)
\Clear    クリアキーを表す文字 ('\013')
\Help     canlisp の中で、ヘルプキーを表す文字 (CANNA_KEY_Help)
\Enter    改行文字 ('\n')
\Return   復帰文字 ('\r')
\F1       canlisp の中で、 F1 キーを表す文字 (CANNA_KEY_F1)
\F2       canlisp の中で、 F2 キーを表す文字 (CANNA_KEY_F2)
\F3       canlisp の中で、 F3 キーを表す文字 (CANNA_KEY_F3)
 :
\F10      canlisp の中で、 F10 キーを表す文字 (CANNA_KEY_F10)
\Pf1      canlisp の中で、 PF1 キーを表す文字 (CANNA_KEY_PF1)
\Pf2      canlisp の中で、 PF2 キーを表す文字 (CANNA_KEY_PF2)
\Pf3      canlisp の中で、 PF3 キーを表す文字 (CANNA_KEY_PF3)
 :
\Pf10     canlisp の中で、 PF10 キーを表す文字 (CANNA_KEY_PF10)
\S-Nfer   canlisp の中で、シフト無変換キーを表す文字 (CANNA_KEY_Shift_Nfer)
\S-Xfer   canlisp の中で、シフト変換キーを表す文字 (CANNA_KEY_Shift_Xfer)
\S-Up     canlisp の中で、シフト上矢印キーを表す文字 (CANNA_KEY_Shift_Up)
\S-Down   canlisp の中で、シフト下矢印キーを表す文字 (CANNA_KEY_Shift_Down)
```

\S-Left canlisp の中で、シフト左矢印キーを表す文字 (CANNA_KEY_Shift_Left)
\S-Right canlisp の中で、シフト右矢印キーを表す文字 (CANNA_KEY_Shift_Right)
\C-Nfer canlisp の中で、コントロール無変換キーを表す文字 (CANNA_KEY_Cntrl_Nfer)
\C-Xfer canlisp の中で、コントロール変換キーを表す文字 (CANNA_KEY_Cntrl_Xfer)
\C-Up   canlisp の中で、コントロール上矢印キーを表す文字 (CANNA_KEY_Cntrl_Up)
\C-Down canlisp の中で、コントロール下矢印キーを表す文字 (CANNA_KEY_Cntrl_Down)
\C-Left canlisp の中で、コントロール左矢印キーを表す文字 (CANNA_KEY_Cntrl_Left)
\C-Right canlisp の中で、コントロール右矢印キーを表す文字 (CANNA_KEY_Cntrl_Right)

3.7.3 エスケープ文字を伴ったコントロール文字の表現

エスケープ文字の後ろが上記キーワードになっていない場合でも、 C- で始まる場合にはコントロール文字として解釈される。

\C-a   ^A を表す文字 (ASCII で 0x01)
\C-b   ^B を表す文字 (ASCII で 0x02)
\C-\   ^\ を表す文字

3.7.4 その他注意点

?\C- の後ろにアスキー以外の文字を置いてはいけない。

?\C-あ → エラー

?\C-a のような記述は GNU Emacs Lisp の記述と似ている。ただし、?\C-Up のような記述は GNU Emacs Lisp には存在しないので注意されたい。

また、^\ を表す表現は、 GNU Emacs Lisp では、

?\C-\\

であるのに対して、 canlisp では、

?\C-\

であることに注意されたい。

ところで、? で始まる文字型の読み込みにおいては、文字として読み込めるところまでしか読み込まない。後ろの部分は新たに reader によって解釈されるので注意が必要である。

?abc   → ?a bc として読み込まれる
?\C-\\ → ?\C-\ \ として読み込まれる
?\F35   → ?\F3 5 として読み込まれる

3.8 文字列型

ダブルクオートで囲まれた部分が文字列として取り扱われる。文字列の途中でエスケープ文字に続いてダブルクオートが現れた場合には、そのダブルクオート文字は文字列の一部として解釈され、文字列の終端とはみなされない。

ASCII 以外の文字も文字列に含めて良い。

```
"abc"                    → 文字列
"これも文字列 "          → 文字列
"This is a \"string\"" → This is a "string" と言う文字列
"foo"bar                 → "foo" bar として読み込まれる
```

エスケープ文字をともなって文字として表現できるものは文字列の中でも表現できる。

```
"\C-a"
"\EscapeOA"
"ABC\S-Right\C-Up12345"
```

ダブルクオートがトークンの途中で現れた場合にはトークンの一部としてみなされる。

```
STRING"abc"  → STRING"abc" と言うシンボル。
```

3.9 シングルクオート

シングルクオートがトークンの最初に現れた場合にはは一般の lisp reader と同様の処理が行われる。
すなわち、シングルクオートに続く S 式を読み込み、(quote S) を生成する。
ただし、シングルクオートがトークンの途中で現れた場合にはトークンの一部としてみなされる。

```
'abc    →  (quote abc)
Tom's   →  Tom's と言うシンボル
```

3.10 ドット

ドットは二進木データを表す時に用いられる。

```
(a . b)
```

ドットを二進木表現のために用いるにはドットの左右に区切り文字を置かなければならない。
リストの途中にドットを用いることも可能である。

```
(a b c d . e)
```

ドットの後ろには一つの S 式が続き、その後に閉じかっこがあることが期待される。

```
(a b . c)       →  問題なし
(a b . (a b))   →  問題なし (a b a b) と同じ。
(a b .(a b))    →  問題なし (ドットと開きかっこをくっつけても良い)
(a b . a b)     →  read error
```

ドットの左右に区切り文字を置かない場合はシンボルとして読み込まれる。

```
co.jp   →  シンボル
Done.   →  シンボル
.canna  →  シンボル
```

3.11 nil

() および nil は NIL 型データの nil として読み込まれる。

以下のように入力すると nil としては見なされず、シンボルとして読み込まれる。

```
\nil
```

3.12 シンボル

以上 canlisp のシンタックスルールの説明をしたが、上記ルールにしたがわないすべてのトークンはシンボルとして読み込まれる。

ASCII 以外の文字を使った場合もシンボルとして読み込まれる。

```
symbol    →   シンボル
記号      →   シンボル
/usr/ucb  →   シンボル
```

区切り文字をシンボルの一部として用いたい場合はエスケープ文字を用いる。

```
Symbol\ with\ spaces → Symbol with spaces と言うシンボル。
Left\(               → Left( と言うシンボル
```

数値だけからなるトークンをシンボルとして用いたい場合はトークンの先頭にエスケープ文字を置く。

```
123   →  数値
\123  →  シンボル
```

シンボルの最大長は lib/canna/lisp.h で定義される BUFSIZE で制約される。具体的には文字列表現として 256 Byte 以上になるシンボルは read error を引き起こす。

4 各データ型の評価後の値

各データ型のデータが lisp の evaluator によって評価されたときの値を以下に示す。

4.1 nil 型

NIL 型の唯一のデータ nil は評価されると nil となる。

4.2 整数型、文字型、文字列型

整数型、文字型、文字列型のデータは評価されると自分自身を値として持つ。

4.3 シンボル型

シンボル型のデータは通常は変数として用いられる。しかし、中には例外もある。

変数としては、関数の引数として用いられたり、 lambda 式、 let 式で束縛される局所変数と、それらの場所に現れず setq などで値が設定される大域変数とがある。 (もちろん局所変数に setq を行うことも可能である)。

4.3.1 lisp 言語で定義される特殊なシンボル

まず、例外のシンボルから説明する。

lisp 言語で定義される特殊なシンボルとしては、 nil に対応する t と、 ZetaLisp や Common Lisp でキーワードパッケージと言うパッケージで管理されているシンボルとがある。

t は大域値の初期値として t 自身を持っている。したがって t を評価すると t が局所的に束縛されていない限り t 自身が得られる。

```
t       評価→    t
```

canlisp ではシンボルの管理にはパッケージは用いないが、コロン (:) で始まるシンボルはキーワードパッケージに属するシンボルとして、 canlisp reader により自分自身を大域値として持つ。したがってキーワードパッケージシンボルを評価するとそのシンボル自身になる。

```
:user 評価→    :user
:abcd 評価→    :abcd
```

4.3.2 『かんな』のカスタマイズに用いられるシンボル

『かんな』のカスタマイズに用いられるシンボルの評価は、そのシンボルにリンクされている『かんな』関数を呼び出した結果になる。

例えば、 server-version と言う変数を評価すると、 cannaserver に接続し cannaserver のバージョンを取り出す処理を行う関数が呼び出され、その関数呼出しの結果が server-version の値として返される。

4.3.3 通常のシンボル

シンボルが局所変数として現在の環境に定義されている場合は、その環境において、そのシンボルと束縛されている値が、そのシンボルを評価した結果の値となる。

これは、 4.3.1や 4.3.2で説明されているシンボルにおいても同様である。例えば、以下に示す関数定義、

```
(defun square (x) (* x x))
```

の x を t に変えても関数 square の意味は変わらない。

```
(defun square (t) (* t t))
```

シンボルが局所変数として束縛されていない場合は、大域値が参照される。さらに大域値として値を持たない場合にはそのシンボルの評価は、 Unbound variable エラーを引き起こす。

4.4 CONS 型 (関数呼出し)

CONS 型の式の評価は CONS 型の式の car 部分に記述される関数の型に依存する。 CONS 型の式の評価結果については関数の型についての項を参照のこと。

5 関数の型と評価

canlisp が取り扱う関数の型には以下がある。

```
SUBR 型
SPECIAL 型
EXPR 型
CMACRO 型
MACRO 型
LAMBDA 型
```

SUBR 型、 EXPR 型、 LAMBDA 型の関数はいわゆる関数であり、引数がすべて評価され関数に渡される。

SPECIAL 型、 CMACRO 型、 MACRO 型は特殊形式であり、引数は評価されず関数に渡される[2]。

5.1 SUBR 型、 EXPR 型、 LAMBDA 型

SUBR 型は、最初から準備されている関数である。例えば、 car、 cdr などは SUBR 型の関数である。

EXPR は defun で定義した関数である。

一度しか使わないような関数は LAMBDA 式を用いて記述することができる。 LAMBDA 式で表される関数を LAMBDA 型の関数と呼ぶ。例えば以下の式の評価では LAMBDA 型の関数の適用が行われる。

```
((lambda (x) (cons x x)) 'a)
```

SUBR 型、 EXPR 型、 LAMBDA 型の関数が実行される場合、引数がすべて評価され関数に渡される。引数は左から順番に評価される。

5.2 SPECIAL 型

SPECIAL 型の場合は、引数は評価されず SPECIAL 型の関数に渡される。その後各引数が評価されるかどうかは SPECIAL 型の各関数に依存する。

SPECIAL 型の関数の例としては setq や、 cond などがある。こうしてみると SPECIAL 型を関数とは呼ばない lisp もあるのもうなずける。

SPECIAL 型の関数はすべて C で記述されており、ユーザが定義することはできない。 SPECIAL 型の動作をする関数を定義したい場合は以下で説明する MACRO 型の関数を defmacro で定義して使う。

5.3 CMACRO 型、 MACRO 型

MACRO は SPECIAL と似ており、引数リストが評価されないまま関数に渡される。 MACRO 型関数により一度処理された式はもう一度 evaluator により評価される。

CMACRO と MACRO の違いは、その関数が C で記述されているか、ユーザ定義かの違いである。

CMACRO 型の関数としては if、 let がある。 if や let も確かに関数と言うよりは制御構造である。 if、 let は一度 cond 式、 lambda 式に変換され、変換後の cond 式、 lambda 式が評価される。

[2]SPECIAL 型、 CMACRO 型、 MACRO 型については関数と言わないのが普通である。

MACRO 型の関数の定義は defmacro を使って行う。 defmacro に関しては「7.7関数定義」を参照されたい。

6 エラー処理

canlisp でエラーが発生すると式の評価が中断し、トップレベルに処理が戻る。『かんな』のカスタマイズファイルパーサとして canlisp が実行されている場合は lisp 言語レベルのエラーは静かに処理される。すなわち、何の表示もなされない。

canlisp コマンドおよび cannacheck −v の場合はエラーが表示される。

load 時のエラーはそのレベルの load の中断を引き起こす。

※注意 エラー処理は将来変更する可能性もある。すなわち、式の評価でエラーが返った時点でその式の値を単に nil にし、上位の式の評価を続けるようにする。なぜなら、 progn での処理時には、通常の lisp 言語では progn 全体を終了してもらって差し支えないが、『かんな』のカスタマイズパーサとしては、エラーが出ていない部分に関しても実行して欲しいからである。

7 個々の関数の説明

以下の説明で引数の記述をする場合、引数として求められる型にしたがって、以下のように記述する。

数値が来るべき部分	`n` で始まる記述
文字列が来るべき部分	`s` で始まる記述
シンボルが来るべき部分	`v` で始まる記述
リストが来るべき部分	`l` で始まる記述
何が来てもよい部分	`g` で始まる記述

また、関数呼出しにしたがって評価される引数はシングルクオートを付けて示した。

7.1 数値演算

各数値演算においてオーバフロー時の処理はC言語におけるオーバフロー時の処理に準じる。

また、 0 による除算を行った場合はエラーとなる。

7.1.1 (+ 'n*1* 'n*2* 'n*3* ... 'n*n*) SUBR

n*1* 〜 n*n* を加算し、その結果を返す。 n*1* 〜 n*n* のいずれかに非数値があるとエラーを引き起こす。引数が全く与えられない場合は 0 を返す。

7.1.2 (− 'n*1* 'n*2* 'n*3* ... 'n*n*) SUBR

n*1* から n*2* 〜 n*n* を引いた結果を返す。 n*1* 〜 n*n* のいずれかに非数値があるとエラーを引き起こす。引数が全く与えられない場合は 0 を返す。引数が 1 つだけ与えられると − は単項演算子として働き、 −n*1* を値として返す。

7.1.3 (∗ 'n*1* 'n*2* 'n*3* ... 'n*n*) SUBR

n*1* 〜 n*n* を乗算し、その結果を返す。 n*1* 〜 n*n* のいずれかに非数値があるとエラーを引き起こす。引数が全く与えられない場合は 1 を返す。

7.1.4 (/ 'n*1* 'n*2* 'n*3* ... 'n*n*) SUBR

n*1* から n*2* 〜 n*n* を順番に除算し、その結果を返す。 n*1* 〜 n*n* のいずれかに非数値があるとエラーを引き起こす。引数が全く与えられない場合は 1 を返す。引数が 1 つだけ与えられた場合は n*1* が値として返される。

7.1.5 (% 'n*1* 'n*2* 'n*3* ... 'n*n*) SUBR

n*1* から n*2* 〜 n*n* を順番に割ってはあまりを求める。引数が全く与えられない場合は 0 を返す。引数が 1 つだけ与えられた場合は n*1* が値として返される。

7.2 文字列処理

7.2.1 (concat 's*1* 's*2* ... 's*n*) SUBR[3]

文字列 s*1* ... 's*n* をつなげた文字列を返す。

引数が 0 個の場合はヌル文字列 "" を返す。

7.3 リスト処理

7.3.1 (cons 'g*1* 'g*2*) SUBR

g*1* を car 部、 g*2* を cdr 部に持つ CONS 型データを作成し、それを返す。

7.3.2 (list 'g*1* 'g*2* ... 'g*n*) SUBR

7.3.3 (sequence 'g*1* 'g*2* ... 'g*n*) SUBR

g*1* g*2* ... g*n* を要素として持つ線形リストを返す。全く引数を与えなかった場合は nil を返す。

sequence は list と全く同じ動作をする。 sequence は『かんな』で機能シーケンスを与える部分で使われる。

7.3.4 (car 'l) SUBR

l の car 部を返す。l として nil を与えた場合は nil が返る。

7.3.5 (cdr 'l) SUBR

l の cdr 部を返す。l として nil を与えた場合は nil が返る。

[3]『かんな』 Version 2.2 からの機能

7.4 述語

以下は canlisp の述語である。一般に述語では、偽を表すときに nil を返し、真を表すときは非 nil の値を返す。多くの述語では真を表す値として t を返す。

7.4.1 (eq 'g*1* 'g*2*) SUBR

7.4.2 (= 'g*1* 'g*2*) SUBR

g*1* と g*2* が全く同じ lisp オブジェクトを指しているときに t を返す。それ以外の場合は nil を返す。

シンボルどうし、 nil どうしは eq で同値性をチェックできる。

canlisp の整数はポインタ表現の即値形式であるため、数値どうしの同値性も eq でチェックできる。

7.4.3 (equal 'g*1* 'g*2*) SUBR

g*1* と g*2* が同じとみなされる場合に t が返される。 2 つの文字列の比較や、 2 つの CONS 型データの比較には eq ではなく、 equal を用いるべきである。

7.4.4 (null 'g*1*) SUBR

7.4.5 (not 'g*1*) SUBR

g*1* が nil の場合 t を返し、それ以外の場合は nil を返す。

7.4.6 (atom 'g*1*) SUBR

g*1* がアトミックである場合、すなわち、 CONS 型ではない場合に t を返す。 g*1* が CONS 型の場合は nil を返す。

(atom nil) は t である。

7.4.7 (> 'n*1* 'n*2* ... 'n*n*) SUBR

n*1* n*2* ... n*n* が大きい順に並んでいるときにのみ t を返す。それ以外の場合は nil を返す。 n*i* は n*i-1* よりも真に小さくなければ t にはならない。すなわち、 n*1* 〜 n*n* のいずれか 2 つが同じ値だった場合は nil を返す。

引数が 1 個以下しか与えられなかった場合は t を返す。

n*1* 〜 n*n* のいずれかが非数値の場合はエラーとなる。

7.4.8 (< 'n*1* 'n*2* ... 'n*n*) SUBR

< は > の反対である。 > と同じく n*1* 〜 n*n* のいずれか 2 つが同じ値だった場合は nil を返す。

7.4.9 (boundp 'v) SUBR

シンボル v が変数として値を持っていれば t を返す。そうでなければ nil を返す。

7.4.10 (fboundp 'v) SUBR

シンボル v になんらかの関数定義がおこなわれていれば t を返す。そうでなければ nil を返す。

7.5 制御構造

7.5.1 (progn 'g*1* 'g*2* ... 'g*n*) SPECIAL

式 g*1* 〜 g*n* を順番に評価し、 g*n* の評価結果を返す。引数をあたえなかった場合は nil を返す。

7.5.2 (cond (g*11* g*12* ... g*1n*) (g*21* g*22* ... g*2n*) ... (g*m1* g*m2* ... g*mn*)) SPECIAL

cond 式は if 〜 then 〜 else if 〜 then 〜 else if 〜 then ... else 〜の構造を与える式である。上記の cond 式は C の if-then-else で記述するとだいたい以下のようになる。

```
if (g11) {
  g12; ... g1n;
}
else if (g21) {
  g22; ... g2n;
}
else if (gm1) {
  gm2; ... gmn;
}
```

すなわち、 g*11*、 g*21*、... g*m1* が、いずれかが非 nil を返すまで順番に評価される。 g*i1* が非 nil を返した場合、 g*i+11* 〜 g*m1* までの式は評価されない。

条件式 g*11*、 g*21*、... g*m1* を順番に評価していき、 g*i1* が非 nil を返した場合、 g*i1* の右に記述されている g*i2*、... g*in* が順に評価され、 g*in* の評価結果が cond 式の値として返される。 g*i1* の右側に全く式が存在しない場合には g*i1* の評価結果そのものが cond 式の値として返される。

条件式 g*11*、 g*21*、... g*m1* がいずれも nil を返した場合は cond 式の結果は nil となる。

ところで、 cond 式の条件式として t が与えられた場合には t の右側が else 条件として評価される。これはたとえ t に nil が束縛されていたとしてもそうである。以下の関数 double は引数として nil を与えた場合、 (nil nil) を返し、 (foo nil) は nil を返すのに対して、 (bar nil) は nil ではなく、文字列 "nil である " を返すことを注意されたい。

```
(defun double (t) (list t t))

(defun foo (x) (cond ((not (null x)) "nil ではない ")
                     (x              "nil である ") ))

(defun bar (t) (cond ((not (null t)) "nil ではない ")
                     (t              "nil である ") ))
```

7.5.3 (and g*1* g*2* ... g*n*) SPECIAL

and は論理積を返す関数であるが、引数 g*1*、 g*2*、... g*n* は順番に評価され、評価結果として nil が戻ったところで評価が打ち切られ、 and 式の値として nil を返す。したがって and は制御構造として用いることができる。例えば、以下の式、

```
(and (> a 0) (do_proc a))
```

は C 言語での、

```
if (a > 0) do_proc(a);
```

のような意味を持つ。

g*1* 〜 g*n* すべてが非 nil だった場合は g*n* の値が and 式の値として返される。

7.5.4 (or g*1* g*2* ... g*n*) SPECIAL

or は論理和を返す関数であるが、 and と同様制御構造である。すなわち、 g*1*、 g*2*、... g*n* はいずれかの式の評価結果として非 nil が返った時点で打ち切られ、その値が or の値として返される。

7.5.5 (if g*1* g*2* g*3*) CMACRO

(if a b c) は、以下のようにマクロ展開され評価される。

```
(cond (a b) (c))
```

(if a b) は、以下のようにマクロ展開され評価される。

```
(cond (a b))
```

if の条件が真だったときの処理をたくさん書きたいときに、

```
(if predicate
  proc1
  proc2
  ...
  procn)
```

のように書きたくなるが、これは正しくない。以下のように progn を使わなければならない。

```
(if predicate
  (progn
    proc1
    proc2
    ...
    procn))
```

7.6 変数束縛

7.6.1 (set 'v 'g) SUBR

シンボル v に対して値 g を代入する。シンボル v が局所的に束縛されていれば v の局所値が変更される。シンボル v が局所的に束縛されていなければ v の大域値として g が代入される。 g の値が set の値となる。

set は以下に示す setq とは違い SUBR であり、各引数が評価される。

7.6.2 (setq v*1* 'g*1* v*2* 'g*2* ... v*n* 'g*n*) SPECIAL

シンボル v*1* に対し式 g*1* を評価した結果を代入する。 v*1* への代入の後 g*2* を評価しその結果を v*2* に代入する。以下 g*n* まで評価を繰り返し、 g*n* の評価結果を setq 式の値として返す。

代入の意味については set の部分を参照のこと。

引数が全く与えられなかった場合は setq の値は nil となる。

7.6.3 (let ((v*1* 'g*1*) (v*2* 'g*2*) ... (v*n* 'g*n*)) gg*1* gg*2* ... gg*n*) CMACRO

let 式はローカル変数 v*1*、 v*2*、 ... v*n* を準備し、それぞれの変数に g*1*、 g*2*、 ... g*n* の評価結果をそれぞれ代入し、その環境において式 gg*1*、 gg*2*、 ... gg*n* を順番に評価する。 gg*n* の評価結果が let 式の値となる。

let 式はマクロであり、

```
(let ((var1 val1) (var2 val2) ... (varn valn)) e1 e2 ... en)
```

は以下のようにマクロ展開され評価される。

```
((lambda (var1 var2 ... varn) e1 e2 ... en) val1 val2 ... valn)
```

7.7 関数定義

7.7.1 (defun vfname (v*1* v*2* ... v*n*) g*1* g*2* ... g*n*) SPECIAL

defun は関数を定義する。全ての引数は評価されない。

第一引数 vfname は定義される関数の名前をしめす。次に現れるのは引数リストである。最後はその関数の本体が現れる。

例えば与えられた数の二乗の数を返す関数 square は以下のように定義する。

```
(defun square (x) (* x x))
```

ここで、 square が定義される関数の名前、 (x) が引数リストを表す。この場合、関数 square は唯一の引数 x を持つ。

(* x x) はこの関数の本体であり、引数 x 自身を掛け合わせた値を戻り値とする。

この定義を行った後、 canlisp インタプリタでこの関数の呼出を行うと以下のようになる。

```
-> (square 2)
4
-> (square 5)
25
```

7.7.2 (defmacro vmname gvars g*1* g*2* ... g*n*) SPECIAL

defmacro はマクロを定義する。全ての引数は評価されない。

defmacro で定義された関数では、その関数への全ての引数はリストとしてまとめられ、 gvars に束縛される。例えば以下の定義を見てみよう。

```
(defmacro spread-value x
  (if (not (null (cdr x)))
     (cons 'let
           (cons (list (list 'spread-tmp-val (car x)))
                 (make-setq-form (cdr x)) ))) )

(defun make-setq-form (l)
  (if (null l) nil
    (cons (list 'setq (car l) 'spread-tmp-val) (make-setq-form (cdr l))) ))
```

マクロ関数 spread-value への引数はリストにまとめられ x に束縛される。例えば、

```
(spread-value 5680 iroha canna)
```

と言う呼び出しの場合は (5680 iroha canna) が x に束縛される。

さて上記の定義で、 make-setq-form は引数に与えられた物に対して spread-tmp-val を setq する式を作成する。

```
-> (make-setq-form '(a b))
((setq a spread-tmp-val) (setq b spread-tmp-val))
```

マクロ spread-value の本体は変数リストの各変数に対して x の car 部にある引数を let 式を組み合わせて代入する lisp 式を生成する。

例えば、 x が (0 a b) であるならば、 defmacro で与えた本体は以下の式を生成する (ここでは見やすさのためプリティプリントしている)。

```
(let ((spread-tmp-val 0))
  (setq a spread-tmp-val)
  (setq b spread-tmp-val) )
```

マクロ式では、 defmacro で与えられた本体により得られる式をもう一度評価する。上の例で spread-value を以下のように呼び出した場合、

```
-> (spread-value 5680 iroha canna)
```

は、あたかも、

```
-> (let ((spread-tmp-val 5680))
     (setq iroha spread-tmp-val)
     (setq canna spread-tmp-val) )
```

を実行したかのように評価が行われ、変数 iroha および canna には 5680 が束縛される。

ここで、 spread-value の第一引数だけは評価が行われ、その他の引数は全く評価されないことに注意されたい。

このようにマクロはスペシャル型の関数のように、一部引数が評価されない関数を作成するのに用いられる。

なお、 Common Lisp などには defmacro を助ける記法としてバッククオートマクロ (lisp reader により処理されるマクロ記述) を用いることができるが、 canlisp ではバッククオートマクロはサポートしていない。

7.8 UNIX 関連

7.8.1 (getenv 's) SUBR

s で示される環境変数の値を返す。返される値は文字列型のデータである。環境変数が定義されていない場合は getenv は nil を返す。

文字列以外のデータを getenv に与えるとエラーになる。

7.9 その他

7.9.1 (quote g*1* g*2* ... g*n*) SPECIAL

quote 式は g*1* 自身を値として返す。 g*1* 〜 g*n* は評価されない。引数がまったく与えられない場合は nil が値となる。

7.9.2 (load 's) SUBR

load は s で示されるファイルを canlisp ファイルとして読み込み、その中に記述されている lisp 式を評価する。

load 中に load を行うことも可能であり、入れこの深さは最大 20 までである。 load 中 EOF になった時点で読み込みが終了し、元のファイルの処理に戻る。

7.9.3 (gc) SUBR

ガーベジコレクションが行われる。

7.9.4 (copy-symbol 'v*1* 'v*2*) SUBR

シンボル v*2* のすべての属性をシンボル v*1* にコピーする。

これは『かんな』の各シンボル名を変更したい場合に用いる。例えば『かんな』でモードや機能を表すシンボルに、 yomi-mode や、 kigou-mode、 kakutei がある。これらのシンボル名はローマ字表記されているが、 copy-symbol を用いて、これらの名前に別の名前を振ることができる。

```
(copy-symbol '読みモード 'yomi-mode)
(copy-symbol '記号モード 'kigou-mode)
(copy-symbol '確定 'kakutei)
```

このようにすることにより、キーバインディングのカスタマイズをするときに次のような書きかたができる。

```
(set-key '読みモード "\C-l" '確定)
```

例えば copy-symbol を行って適当な名前を付けるファイルを別に用意して、それを load することにより、種々のモード名や、機能名に日本語の名称を付けることができる。例えば copy-symbol をしているファイルを /usr/local/canna/lib/jpn.canna とすると、以下のような記述が可能である。

```
(load "/usr/local/canna/lib/jpn.canna")

(設定 ローマ字かな変換テーブル "default.kp")
(辞書利用 "iroha" "fuzokugo" :部首 "bushu" :ユーザ "user")

(設定 逐次自動変換 する)
```

この場合 jpn.canna には最低限以下の記述があったことになる。

```
(copy-symbol '設定                    'setq)
(copy-symbol 'ローマ字かな変換テーブル 'romkana-table)
(copy-symbol '辞書利用                'use-dictionary)
(copy-symbol '逐次自動変換            'auto)
(setq :部首    :bushu
      :ユーザ  :user
      する     t)
```

『かんな』の各種シンボルにはローマ字をベースにした名前が多いが (例: henkan、 kakutei) 将来『かんな』が韓国語や中国語に対応する場合、ローマ字ベースの各種シンボルを copy-symbol 機能を用いて別の名前に変更することが可能である。

7.10 『かんな』関連

7.10.1 (defmode vname ['sdpy ['srt ['gfn ['gus]]]]) SPECIAL

あたらしい入力モードを定義する。このモードでは、独立のローマ字かな変換テーブルを持ち、キーバインディングもこのモードでだけ独立で行うことができる。

vname は新しいモードを表すシンボルとして用いることができるようになる。また、 vname は、そのモードへ移行する機能を表すシンボルとしても用いることができるようになる。

sdpy はこのモードになったときにモード表示部分に表示される文字列である。

sdpy の指定がなければ nil が指定された場合と同じ振る舞いをする。

srt はこのモードで用いられるローマ字かな変換テーブルである。通常ののローマ字かな変換テーブルと同じローマ字かな変換テーブルを用いたい場合は、

```
(setq romkana-table 'srt)
```

が defmode よりも前に存在することを確認して、 defmode の srt の部分に romkana-table を指定すれば良い。

srt の指定がなければローマ字かな変換は行わない。

gfn はこのモードで、入力が行われたときに実行される機能である。機能としては kakutei、 henkan、 zenkaku、 hankaku、 hiragana、 katakana、 romaji、 to-upper、 capitalize、 to-lower が指定できる。各指定の意味については『かんな』マニュアルの方を参照すること。

ただし、 henkan、 to-upper、 capitalize、 to-lower は『かんな』 Version 2.2 〜 3.2 では単に無視される。

gfn の指定がなければ、空の機能リストが渡されたのと同じ意味となる。

gus は defsymbol でのシンボル定義をそのモードでは使うかどうかを表す。例えば普段はローマ字かな変換を用い、 defmode により疑似カナ入力を定義し時々用いる場合、疑似カナ入力では数字キーも用いるため数字キーに対する defsymbol は邪魔になる。

gus が nil であれば、 defsymbol での定義はそのモードでは用いられない。非 nil であれば、用いられる。指定がなければ nil とみなされる。

7.10.2 (defsymbol nkey*1* s*11* ... s*1n* ... nkey*m* s*m1* ... s*mn*) SPECIAL

各引数は評価されない。 nkey*i* は文字を記述する。

defsymbol の意味については『かんな』マニュアル参照のこと。

7.10.3 (defselection vname sdpy 'llist) SPECIAL

なんらかの一覧表示を簡単に出せるように定義する[5]。

最後の引数のみが評価される。 llist には文字のリストか、あるいはハイフン (-) で範囲指定を与える。 defselection の詳細に関しては『かんな』マニュアルを参照のこと。

7.10.4 (defmenu vname (sentry*1* vfn*1*) ... (sentry*n* vfn*n*)) SPECIAL

メニューの定義を与える[6]。

引数は評価されない。この定義をすると vname の呼び出しにより sentry*1* 〜 sentry*n* からなるメニューが表示され、特定の項目 sentry*i* の選択によりその項目に対応する機能 vfn*i* が実行される。 vfn*i* はさらに defmenu で定義されたメニューでも構わない。

defmenu の詳細に関しては『かんな』マニュアルを参照のこと。

7.10.5 (set-mode-display 'vmode 'sdpy) SUBR

vmode のモード表示文字列を sdpy にする。

[5] Version 3.2 からの機能

[6] Version 3.2 からの機能

7.10.6 (set-key 'vmode 'skeys 'gfn) SUBR

『かんな』のモード vmode にて、１つ以上のキーの並び skeys に１つ以上の機能の並び gfn をバインドする。

gfn は機能を表す単一のシンボルか、機能を表すシンボルからなる線形リストでなければならない。

vmode がモードを表す機能でなかったり、 gfn の機能の中に、機能を表すシンボルでないものが含まれている場合はエラーになる。

7.10.7 (global-set-key 'skeys 'gfn) SUBR

すべてのモードに対してキーと機能のバインディングを行う点を除いて set-key と同じ。

7.10.8 (unbind-key-function 'vmode 'gfn) SUBR

vmode において gfn で指定された機能が割り当ててあるキーを無効にする。 gfn は機能を表す単一のシンボルか、機能を表すシンボルからなる線形リストでなければならない。

7.10.9 (global-unbind-key-function 'gfn) SUBR

すべてのモードに対してキーと機能のバインディングを行う点を除いて unbind-key-function と同じ。

7.10.10 (define-esc-sequence 'sterm 'sseq 'nkey) SUBR

7.10.11 (define-x-keysym 'skey 'nkey) SUBR

これら２つの関数は将来の拡張用に予約してある。

7.10.12 (use-dictionary 's*1* ... [':bushu 's*i*] ... [':user 's*j*] ... 's*n*) SUBR

辞書の利用を指示する。すべての引数は評価される。キーワード引数がいくつか指定できる。利用できるキーワードは :bushu および :user である。

use-dictionary の詳細は『かんな』マニュアルを参照のこと。

7.10.13 (initialize-function 'gfn) SUBR

初期化時に実行される機能を指定する。

gfn は機能を表す単一のシンボルか、機能を表すシンボルからなる線形リストでなければならない。

A 特殊なシンボルの表

A.1 『かんな』関連の変数

以下のうち canna-directory は『かんな』 Version 2.2 で追加された変数である。また、 auto-sync, force-kana, hiragana-touroku, ignore-case, katakana-touroku, quickly-escape-from-kigo-input, renbun-continue, romaji-yuusen は Version 3.2 で追加された変数である。

abandon-illegal-phonogram	ローマ字かな変換に使われない文字を捨てるか
allow-next-input	一覧表示時に次の入力を許すか
auto	逐次自動変換するか
auto-sync	単語登録削除時に自動的に辞書が SYNC されるか
backspace-behaves-as-quit	バックスペースが quit のように働くか
break-into-roman	バックスペースでローマ字に戻すか
bunsetsu-kugiri	文節間にスペースを入れるか
canna-directory	カスタマイズファイルなどがおいてあるディレクトリ
canna-version	『かんな』のバージョン
character-based-move	カーソル移動が文字単位か
chikuji-continue	逐次自動変換候補表示中の入力で確定するかどうか
chikuji-force-backspace	逐次自動文節時バックスペースで必ず 1 文字消すかどうか
cursor-wrap	右端から右へ行くと左へ行ったりするのをするか
english-table	外来語変換用辞書を入れておく変数
force-kana	《使っていない》
gakushu	学習するか
grammatical-question	単語登録時に品詞分類のための質問をするか
hex-direct	１６進コード入力で４ケタ目にワンクッション置くか
hiragana-touroku	《使っていない》
ignore-case	ローマ字かな変換で大文字小文字を区別しない
index-hankaku	候補一覧のインデックス数値を半角にするか
index-separator	候補一覧のインデックス数値と候補の間の文字を入れておく
inhibit-list-callback	リストコールバックを禁止するか
kakutei-if-end-of-bunsetsu	文節の最後から右へ行こうとした時に確定するか
katakana-touroku	《使っていない》
keep-cursor	《使っていない》
keep-cursor-position	変換から読みに戻した時のカーソル位置をどこにするか
kojin	個人別学習をするか
kouho-count	候補が何番目かを候補一覧時に表示するか
learn-numerical-type	《使っていない》
n-henkan-for-ichiran	何番目の変換キーで候補一覧を出すか
n-keys-to-disconnect	どのくらいかな漢字変換しないと接続を切るか
n-kouho-bunsetsu	逐次自動変換で確定しないでおく文節数
numerical-key-select	候補一覧の時に数字キーを出すか
protocol-version	プロトコルのバージョンが入っている変数
quickly-escape-from-kigo-input	記号入力でひとつ選んだらすぐ抜けるか
quit-if-end-of-ichiran	一覧の最後で次の候補を出そうとすると一覧を終わるか
renbun-continue	連文節変換時、次入力開始でも前の文節が残るか
reverse-widely	読み入力中のリバースの幅を広くするか
reverse-word	一覧時のリバースの幅を広くするか
romaji-yuusen	ローマ字かな変換を優先させるか
romkana-table	ローマ字かな変換テーブルを入れておく
select-direct	候補一覧で数字を選ぶと一覧が終るか
server-name	サーバ名が入っている変数
server-version	サーバのバージョンが入っている変数
stay-after-validate	候補一覧で選択した後、同じ文節にとどまるか

A.2 『かんな』のモードを表すシンボル

以下「＊」が付いているのはリアルモードで、他はイマジナリモードである。リアルモードではモード名の変更とそのモードだけでのキーの割当ができるが、イマジナリモードに関してはモード名の変更しか行えない。イマジナリモードはリアルモードの組合せで作られているのでイマジナリモードでのキーバインディングはベースとなるリアルモードによって決められる。例えば extend-mode のキーバインディングは ichiran-mode でのキーバインディングがそのまま用いられる。

以下のうち、 bubun-muhenkan-mode は Version 3.2 以降は無効である。

alpha-mode	＊アルファベットモード
bubun-muhenkan-mode	編集モード
bushu-mode	部首モード
changing-server-mode	サーバを変更しようとしているモード
chikuji-bunsetsu-mode	＊逐次自動変換で文節上にいるモード
chikuji-yomi-mode	＊逐次自動変換で読みの上にいるモード
delete-dic-mode	単語削除モード
empty-mode	＊まだ入力をしていないときのモード
extend-mode	拡張メニューモード
greek-mode	ギリシャ文字選択モード
han-alpha-henkan-mode	半角アルファベット変換入力モード
han-alpha-kakutei-mode	半角アルファベット確定入力モード
han-hira-henkan-mode	半角ひらがな変換入力モード
han-hira-kakutei-mode	半角ひらがな確定入力モード
han-kata-henkan-mode	半角カタカナ変換入力モード
han-kata-kakutei-mode	半角カタカナ確定入力モード
henkan-method-mode	変換方式切替えモード
henkan-nyuuryoku-mode	変換入力モード
hex-mode	１６進コード入力モード
ichiran-mode	＊候補一覧モード
kigou-mode	＊記号入力モード
line-mode	罫線素片選択モード
mojishu-mode	＊字種変換モード
mount-dic-mode	辞書マウントアンマウントモード
on-off-mode	＊選択 / 非選択モード
quoted-insert-mode	＊引用入力モード
russian-mode	ロシア文字 (キリル文字) 選択モード
shinshuku-mode	＊文節伸縮モード
tankouho-mode	＊単候補表示モード
touroku-dic-mode	単語登録時辞書選択モード
touroku-hinshi-mode	単語登録時品詞選択モード
touroku-mode	単語登録モード
yes-no-mode	＊「はい」か「いいえ」を答えるモード
yomi-mode	＊読み入力モード
zen-alpha-henkan-mode	全角アルファベット変換入力モード
zen-alpha-kakutei-mode	全角アルファベット確定入力モード
zen-hira-henkan-mode	全角ひらがな変換入力モード
zen-hira-kakutei-mode	全角ひらがな確定入力モード

zen-kata-henkan-mode	全角カタカナ変換入力モード
zen-kata-kakutei-mode	全角カタカナ確定入力モード

A.3 『かんな』の機能を表すシンボル

以下は『かんな』の機能を表すシンボルである

chikuji-mode, delete-dic-mode, disconnect-server, greek-mode, henkan-or-do-nothing, henkan-or-self-insert, jisho-ichiran, line-mode, renbun-mode, russian-mode, show-canna-file, show-canna-version, show-gakushu, show-romkana-table, show-server-name, switch-server, sync-dictionary, touroku-mode は Version 3.2 での追加機能である。

alpha-mode	アルファベットモードになる
backward	後方（左）へ行く
base-eisu	ベースを英数にする
base-hankaku	ベースを半角にする
base-henkan	ベースを変換にする
base-hiragana	ベースをひらがなにする
base-hiragana-katakana-toggle	ベースをひらがなとカタカナのと間で切替える
base-kakutei	ベースを確定にする
base-kakutei-henkan-toggle	ベースを確定と変換との間で切替える
base-kana	ベースをかな（ひらがなかカタカナ）にする
base-kana-eisu-toggle	ベースをかなと英数との間で切替える
base-katakana	ベースをカタカナにする
base-rotate-backward	ベースを順方向に切替える
base-rotate-forward	ベースを逆方向に切替える
base-zenkaku	ベースを全角にする
base-zenkaku-hankaku-toggle	ベースを全角と半角との間で切替える
beginning-of-line	行頭に行く
bushu-mode	部首モードになる
capitalize	先頭文字を大文字にする
chikuji-mode	逐次自動変換入力モードにする
convert-as-bushu	読みを部首名とみなして部首変換する
convert-as-hex	読みを１６進コードとみなして１６進コード変換する
delete-dic-mode	単語削除モードに入る
delete-next	次（右）の文字を削除する
delete-previous	前（左）の文字を削除する
disconnect-server	サーバとの接続を切断する
end-of-line	行末に行く
extend	文節伸ばし
extend-mode	拡張メニューモードへ行く
forward	前方（右）に行く
greek-mode	ギリシャ文字一覧表示する
han-alpha-kakutei-mode	半角アルファベット確定入力モードになる
han-kata-kakutei-mode	半角カタカナ確定入力モードになる
hankaku	半角に変換する
henkan	変換する

(henkan-naive	ナイーブに変換する)
henkan-or-do-nothing	変換または何もしない
henkan-or-self-insert	変換または self-insert
henkan-nyuuryoku-mode	変換入力モードになる
henshu	編集モードに入る
hex-mode	１６進コード入力モードになる
hiragana	ひらがなに変換する
japanese-mode	日本語入力モードになる
jisho-ichiran	辞書マウント / アンマウントモードに入る
kakutei	確定する
katakana	カタカナに変換する
kigou-mode	記号入力モードになる
kill-to-end-of-line	行末まで削除する
kouho-ichiran	候補一覧を表示する
line-mode	罫線一覧を表示する
mark	マークを設定する
next	次を出す
previous	前を出す
quit	とりやめる
quoted-insert	引用入力をする
renbun-mode	連文節変換モードにする
romaji	ローマ字に変換する
russian-mode	ロシア文字 (キリル文字) 一覧表示をする
self-insert	自分自身を入力する
shinshuku-mode	文節伸縮モードになる
show-canna-file	どのカスタマイズファイルを使っているかを表示
show-canna-version	『かんな』のバージョンの表示をする
show-gakushu	学習状態の表示をする
show-romkana-table	どのローマ字かな変換テーブルを使っているか表示
show-server-name	サーバ名を表示する
shrink	文節縮め
sync-dictionary	辞書を SYNC する
switch-server	サーバの切り替えを行う
temporary	一時モードに入る
to-lower	小文字に変換する
to-upper	大文字に変換する
touroku	単語登録モードに入る
touroku-mode	単語登録 / 削除モードに入る
undefined	未定義の機能
zen-alpha-kakutei-mode	全角アルファベット確定入力モードになる
zen-hira-kakutei-mode	全角ひらがな確定入力モードになる
zen-kata-kakutei-mode	全角カタカナ確定入力モードになる
zenkaku	全角に変換する