20. エディタ論— CRT向きポータブルLisp エディタPLETを中心として—

京都大学 数理解析研究所　小島啓二
Keiji Kojima

はじめに

エディタは、プログラムのインプット及び修正に不可欠なツールであり、その使い勝手はプログラム作成の能率に大きな影響を与える。ここではプログラミング支援ツールとしてのエディタ、特に構文向きのエディタを比較論的に考察しつつ、「便利で使い易いエディタ」の設計思想について、筆者が開発したポータブルなLispエディタPLETを中心にして論じる。

§1. プログラミング・ツールとしてのエディタ

現在プログラミング支援ツールとして使用されているエディタは、その目的及び使用される環境等によって次の二つに大きく分類できる。

(1) 汎用エディタ（General Purpose Editors）
　　例　行単位のエディタ
　　EMACS，TV
(2) 構文向きエディタ（Syntax-directed Editors）
　　例　Lispエディタ（INTERLISP，PLETなど）
　　MENTOR，IOTA

(1)の汎用エディタはファイルの内容の編集用に使用されるもので、任意の形式のテキストを対象とする。プログラム作成に使用される場合は、プログラムのテキストを単なる文字列として扱って編集を行うことになる。もちろん例えばEMACSは主要なプログラミング言語（アセンブリ言語、PL/I、PASCAL、LISP等）の編集に適したモードや各種コマンドをそれぞれの言語について持っている。また、行単位のエディタの中には各行に対してFORTRAN文として構文的に正しいか否かのチェックを行う機能を持つものもある。しかしこれらはあくまで補助的機能であり、構文的に正しくないプログラムテキストのまま編集を完了する可能性が常に存在する。この点が、(2)の構文向きエディタと本質的に異なる点であると考えられる。一般に汎用エディタの設計は、そのエディタの属するオペレーティングシステムの機能と深いかかわり合いを持ち、単独で論じるのは無意味な場合も多い。

ここで紹介するポータブルLispエディタPLETは(2)の構文向きエディタに属する。構文向きエディタは特定の言語専用に設計されたもので、その言語のプロセッサを含むプログラミングシステム内に支援ツールとして存在する場合が多い。これは、プログラムの修正、テストをくり返すデバッギングの際の能率向上に役立つ。構文向きエディタは特にオンラインでプログラミング・デバッギングを行う場合には非常に有効な支援ツールである。実際、INTERLISP，UCILISPなどのLispプログラミングシステムにおけるデバッグの容易さの主な原因の一つにはLispエディタの存在があげられる。一方Lispにはさまざまな個性を持

った方言があり、これらの中には裸の（オンライン・サポートを持たない）ものも多いので、これらに共通に使えるポータブルなLispエディタがあると非常に便利である。さらに、Lispをシステム開発用言語として見た場合、エディタが大きくスペースをとってユーザーのプログラムスペースを圧迫する様では意味がない。またエディタが必要なのは主にプログラムの開発中であり、デバッグが終われば不要となる。この面からもポータブルなLispエディタは有利である。そこでPLETは

(a) プログラミング支援ツールとして必要十分な機能を持つこと
(b) 覚え易く、使い易いこと
(c) ポータビリティ

の3点を基本的な目標として設計された。以下、デザイン上のポイントを順を追ってみていく。

§2. PLETの基本構成

作業能率を考えれば、§1でもふれたように、構文向きエディタを使って編集を完了したにもかかわらず、そのテキストの構文が正しくなかった?!などという事態が起こり得ない様にしたい。（これを構文向きエディタの定義としてよいと思う。）これを実現する為のエディタの基本構成はいくつか考えられる。最も自明な構成は、図1の構成(a)であろう。即ち、プログラムテキストは文字列として文字列編集部（ストリングエディタ）で編集された後、構文解析部に送られて解析され、構文エラーがあれば再びストリングエディタが呼ばれ、エラーがなければ内部形式（構文木など）が出力される。従来の汎用エディタとコンパイラの組み合わせはこのタイプの構文向きエディタとみなすことができる。

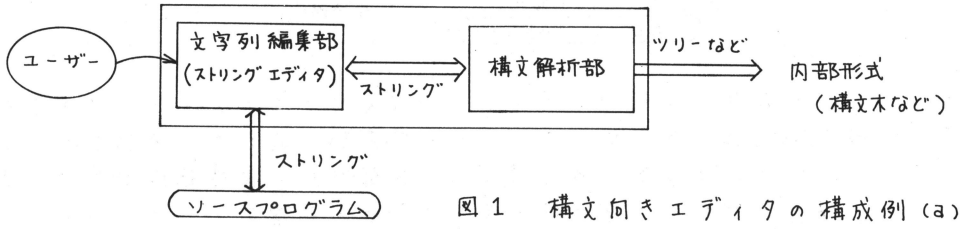


図1　構文向きエディタの構成例(a)

この方式の場合、ストリングエディタには言語の構文に適した種々の機能が必要となる。例えば汎用エディタではあるが、EMACSはLisp関数の編集用に便利な機能を持っている。（図2）しかし、EMACSの様に使い易く、機能の豊富なストリングエディタを実現する為には画面制御など、機械に依存する部分が圧倒的に多くなり、サイズも大きなものとなってしまう。従って当然ポータビリティは皆無で、PLETの構成としては採用し難い。また、構成(a)の欠点は、構文解析が常にテキスト全体に対してなされる、即ち処理が重複してしまう点にもある。特にアルゴル系の言語の場合は構文解析に時間がかかる為にこの構成には問題がある。Lispの場合でも、大きなS式（テキスト全体）において一度かっこの対応がくずれると、正しく回復させるのはかなり困難であり、ストリングとしての編集の対象は小さく制限した方が賢明である。そこで編集を局所化するために、図3の構成(b)が考えられる。

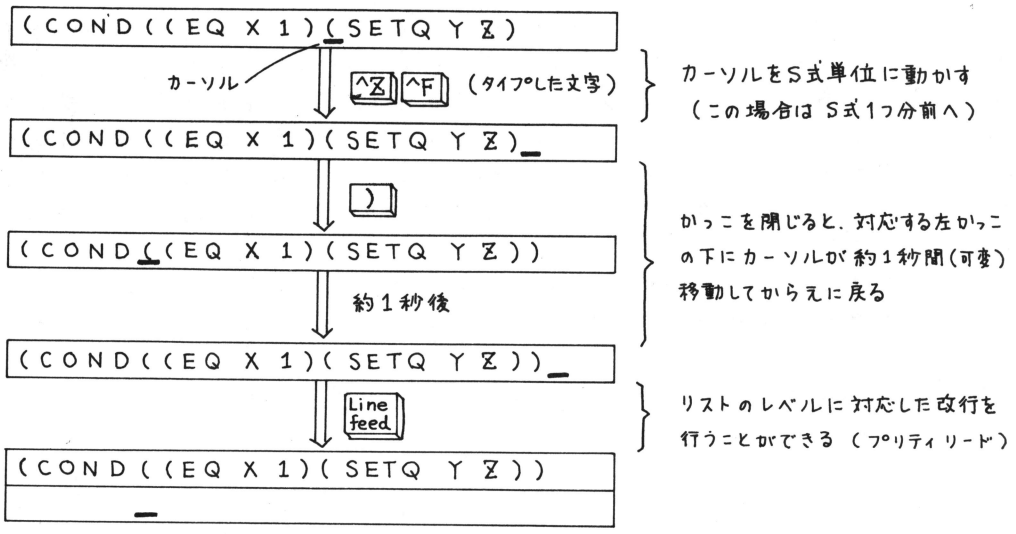


図2　EMACS の Lisp 用機能の一例

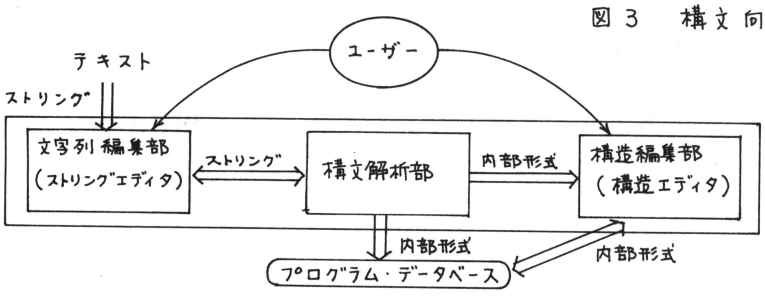


図3　構文向きエディタの構成例(b)

この方式では、ソーステキストはまずエラーがなくなるまでストリングエディタで編集されて構文解析部に送られ、でき上がった内部形式がまず保存される。これ以後、このプログラムが変更される際には、内部形式が直接構造編集部に送られる。ユーザーは、内部形式の中の修正すべき部分を構造エディタによって指定する。修正部分の新しいソーステキスト(文字列)は、ストリングエディタ及び構文解析部を経て内部形式となり、全体の内部形式に加えられる。この様な方法で処理を局所化するのである。アルゴル系の構文を持つイオタ語の為のプログラミングシステムであるイオタ・システムのエディタは基本的にはこの(b)の構成を採用している。ただし、より能率を向上させる為に図4のような構成をとっている。(b)の方式でも初めのテキストが構文解析部を通過しないうちはテキスト全体を処理しなければならない。そこでイオタシステムのエディタでは、構文エラーを一応の木構造がつくれるものとそうでないものとに分け、(これに対応して構文解析部はIとIIとにそれぞれ分けられている。)つくれる場合には、スケルトンと呼ばれる擬構文木を生成してツリーエディタに送っている。この様にして、より早く木構造の上での編集に移行することによって、処理の重複を極力避けている。(b)タイプの構文向きエディタは、二つのエディタ(ストリングエディタと構造エディタ)を持つため、操作が複雑になる恐れがある。

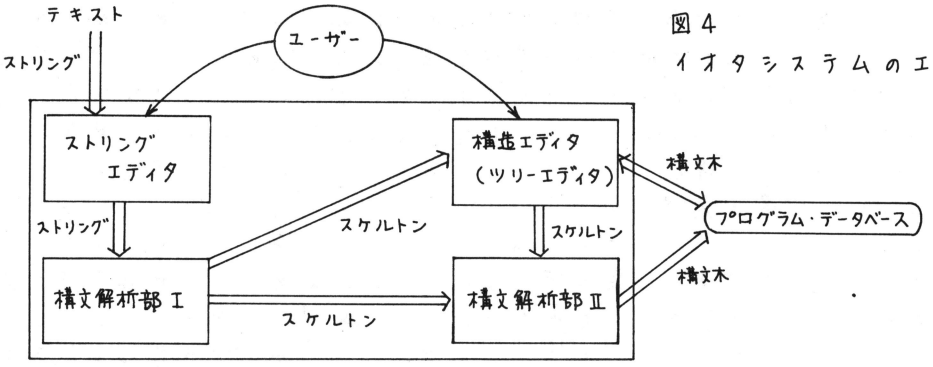


図4
イオタシステムのエディタ

これは覚え易さ、使い易さというPLETの目標からみるとマイナス要因である。また、サイズも当然かなり大きくなると予想される。さて、今まで論じて来た様に、構文木などの内部形式の上で直接に編集を行うことは、処理の能率の面から有利であるし、言語の構文に応じた種々の機能が自然に得られるという長所もある。そこで最初から最後まで、常に内部形式のまま編集を行う様な構成(c)(図5)が考えられる。当然問題は言語の構文と内部形式とのギャップであるが、幸いLispという言語はそれ自体が内部形式といっても良く、この構成(c)は極めて自然なものとなる。

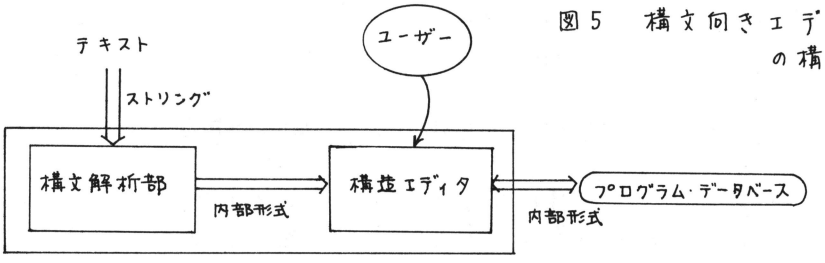


図5 構文向きエディタの構成例(c)

前にあげたINTERLISPのエディタは、この構成である。この(c)タイプの構文向きエディタは、構造エディタだけしか含まない為操作が簡単でかつコンパクトにまとめることができるので、PLETもこの構成を採用している。アルゴル系の言語においても(c)の構成は可能で、PASCAL用の構文向きエディタであるMENTORはこの構成をとっている。

§3. 構造(木・リスト)の編集

ここでは木構造やリスト構造を手軽に編集する為のコマンド体系について考える。一般に、エディタによる編集の一単位は、次の各ステップに分けられる。

ステップ1　現在どの様な状態にあるか確認する
ステップ2　修正対象を限定する
ステップ3　修正コマンドを出す
ステップ4　修正が正しく行われたかどうか確認する

PLETの基本目標の一つである「使い易さ」の達成の為には、これら各ステップがクリアーで、バランスがとれている事が要請される。上の各ステップが具体的にどういうものであるかによって、そのエディタの個性が大部分決定する。例えば、カーソル（普通のblinking cursorだけでなく、テキスト中の一点を示す何らかのindicatorをこう呼ぶことにする。）を持つエディタの場合ならば、ステップ1はカーソル位置の確認、ステップ2は修正対象のある場所へカーソルを移動させることに対応している。これに対し、カーソルを持たないエディタの場合、ステップ2における対象の限定方法として、その対象を一意的に指定するような量（行番号、リストの何番目の要素かetc.）を発見し、ステップ3で出すコマンドのパラメータとして送るのが普通である。今までふれてきた構文向きエディタ（INTERLISP, IOTA, MENTOR）はいずれもカーソルを持たないエディタである。INTERLISPのエディタは、「現在注目しているS式」（Current Expressionと呼ばれている。）を持ち、それを扱う対象としている。この注目部分を変えるコマンドとして整数nを与えている。即ち、"n"（n>0）によって現在注目しているリストのn番目の要素が新たな注目部分となり、"0"によって注目部分が1レベル大きくなる。（図6）

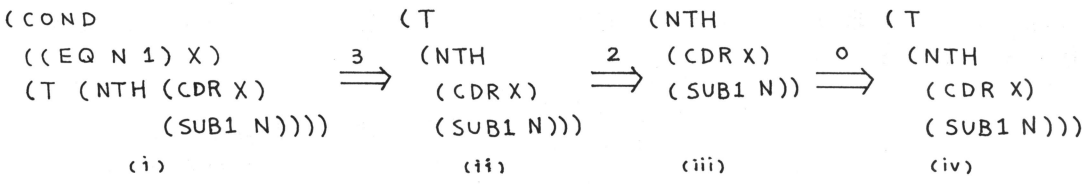


図6　Current Expression の変化（INTERLISP）

修正の対象を限定するには、この様に扱うリストを小さくしてから、その修正対象が現在注目しているリストの何番目の要素であるかをいえばよい。例えば、図6(i)でNTHを指定したい場合は、まず注目部分を(iii)にする。この状態でNTHは最初の要素であるから整数1がNTHを指定する値となる。イオタシステムのエディタも木とリストの違いだけで同じ手法を用いている。（図7）

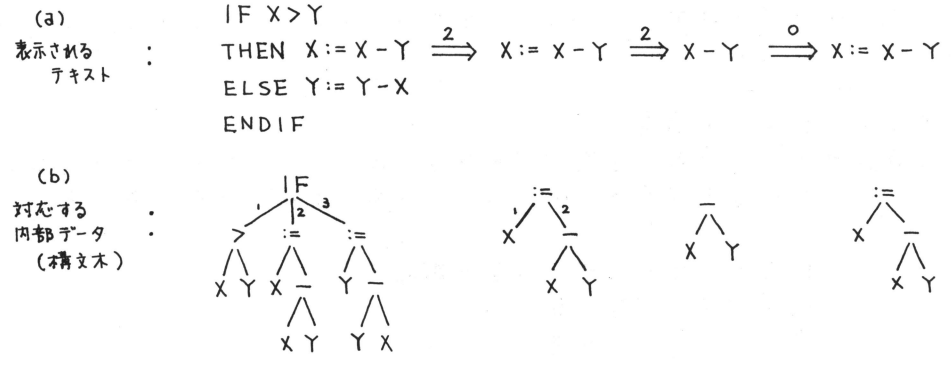
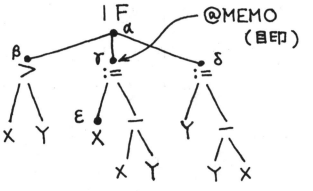


図7　Current Tree の変化（IOTA）

いま述べたCurrent Expression（Current Tree）の方式で問題となるのは同じ部分リスト（部分木）が2個所以上に現われる場合である。その中のどれに注目しているかを知る為には、一度注目している部分を大きくして違いを発見してから元の状態に戻るか、或いは編集の履歴を考えることによるかのどちらかしかない。即ち前にあげた各ステップのうち、ステップ1，ステップ4はともに自明とはいえない。

さてカーソルのないツリーエディタで、別の方法をとっているのはMENTORである。MENTORは、構文木の節（node）に目印（repère）をつける為の種々のコマンドを持っていて、修正対象の指定はこの目印を足場に行われる。



@MEMO U(Up)	d を示す
@MEMO R(Right)	δ "
@MEMO L(Left)	β "
@MEMO S(Son) 1(First)	ε "

図8　目印法（MENTOR）

例えば図8の構文木において節γに目印"@MEMO"をつけたとする。この時IF文全体を指定する場合は「@MEMOの上」といえばよいし、代入文Y:=Y-Xを指定するには「@MEMOの右」と表現する。この方法は、一度目印をつけてしまえば、以後いつでもアクセスできる点が優れている。しかし現在どのような状態にあるかのステップ1はやはり自明でない。

一方カーソルを使用した場合、ステップ1の状況確認は、INTERLISPなどの場合と違って自明となる。（よほど見つけにくいカーソルでなければ）また、ステップ3で出す修正コマンドに、リストの何番目の要素かなどのパラメータを送る必要がなくなり、コマンド体系が簡素なものとなって憶え易いという長所がある。これらの点を重視し、PLETはカーソルを持つLispエディタとして設計されている。PLETでは特殊な記号'ハ'をテキスト中に挿入することによってカーソルとしている。（図9）

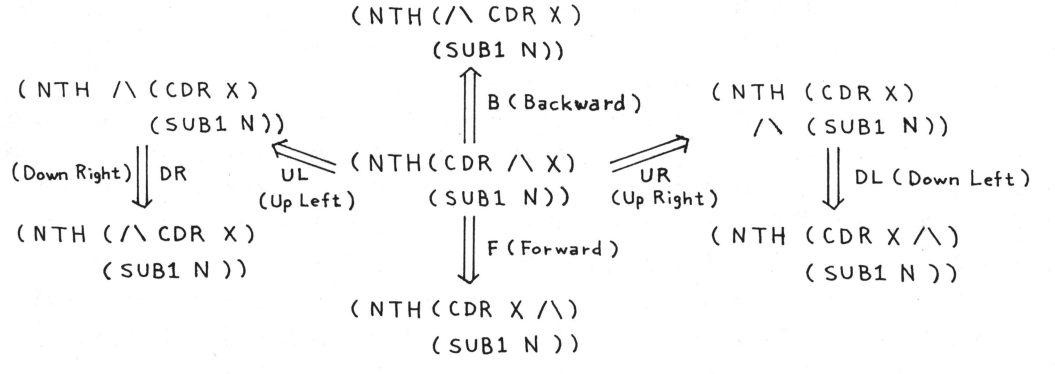


図9　リスト構造におけるカーソルコントロール（PLET）

図9のカーソルコントロールコマンドはリスト構造に従ったもので、PLETはこれらのほかにS(Search)とRS(Reverse Search)の二つのカーソルを動かすコマンドを持っている。

ステップ2の修正対象の限定の方法についていろいろみて来たが、次にステップ3における修正コマンドについて検討する。PLETのようにストリングエディタを持たない構文向きエディタの場合、テキストの構造を変化させることは、I(Insert), D(Delete)やR(Replace)などの基本的な操作と、テキストの適当な部分の移動とによって行われる。例えばリスト構造の編集の場合、結局あるS式を別の位置へ移動させる場合が非常に多く、INTERLISPでもこの目的の為に多くのコマンドが用意されている。そこでこの移動用コマンドを使い易く、かつ少なくまとめることが、修正コマンド体系をすっきりしたものとするのに大きな効果がある。EMACSではテキスト移動に使えるコマンドとしてKillとYankと呼ばれる二つのコマンドを持っている。PLETではこのK(Kill)とY(Yank)をスタックをコントロールしながら使用している。即ち、Killでカーソルの次にあるS式を消去すると同時にスタックにプッシュする事によって記憶し、Yankでスタックをポップしてスタックトップにあった要素をカーソルの次に挿入する。従ってKillし、カーソルを動かした後Yankすることによってテキストの移動が行われる。さらにYankと同種のコマンドで、スタックをポップしないものを用意しておけば、テキストのコピーを作ることもできる。PLETではこのコマンドをYC(Yank for Copy)と呼んでいる。図10にPLETにおける、KillとYankによる移動の例を示す。

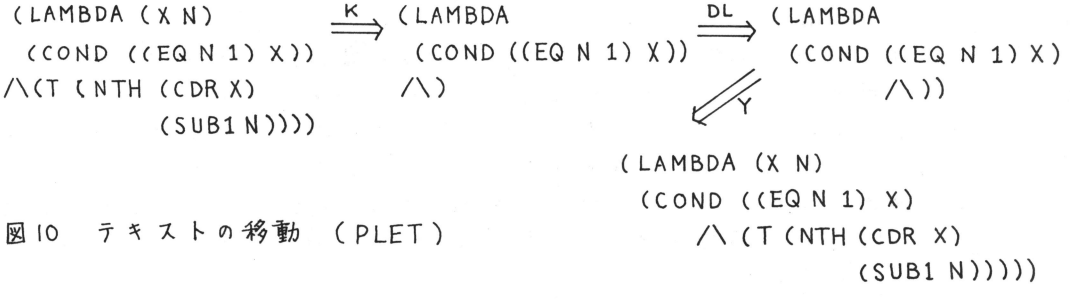
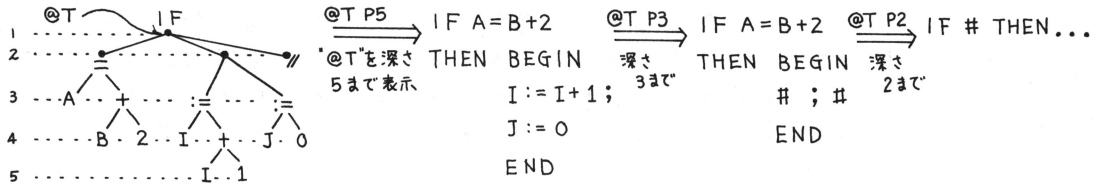


図10　テキストの移動　(PLET)

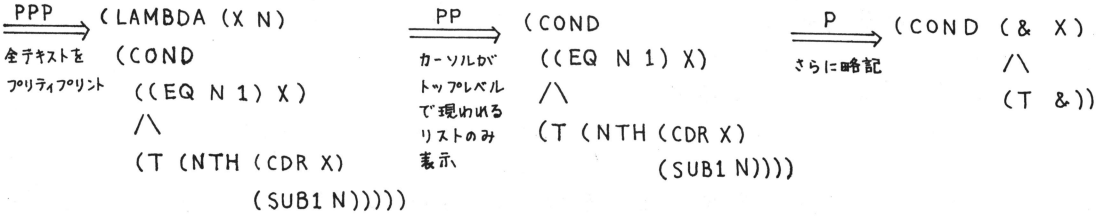
PLETの修正コマンド全体は、I, D, R, K, Y及びYCから成り、KillとYankの採用により簡素なものとなっている。PLETのKillとYankは非常にflexibleで、機能的にも上のコマンド群は十分である。

最後にステップ1とステップ4つまり「確認」のステップについて考えてみる。ここで最も重要なのは、いうまでもなくプリントコマンドである。プログラム構造をユーザーにわかり易い形で表示するプリティプリンタは構文向きエディタには不可欠である。実際多くのエラーがプリティプリントすることによって発見される。大きなテキストを編集する場合、プリント命令を出す度に全テキストをプリティプリントされると非常に煩わしい。INTERLISPのエディタは前述の様に常に一部分(Current Expression)しか見えないので一般に出力量は小さい。(この意味ではテレタイプ向きといえるかも知れない。)　しかしPLETでもカーソルがトップレベルの要素として現われるリストのみプリティプリントする機能を

用意しているので同じ効果が出せる。この様にテキストの一部分だけ表示したり、さらに構造の深さに応じて略記を行ったりする機能は大変有効である。図12に種々のプリティプリントの例を示す。



(a) 深さによる略記 (MENTOR)



(b) PLETのプリティプリント

図12 種々のプリティプリント

ステップ4の確認で誤りを発見した場合、ステップ2及び3を破棄してステップ1の状態に戻る機能(PLETではUNDOと呼んでいる。)は使い易さにとって重要である。(例えば誤ってうっかり大きなリストを消去してしまった時など)
PLETはさらに、UNDOが連続して行えて、次々と以前の状態に復帰できるように設計されている。

§4. PLETのその他の機能およびまとめ

基本編集コマンドとしては§3で述べたもので十分で、PLETはこれらのくり返し機能とマクロ機能とを持っている。くり返しは、あるコマンド列全体を任意の回数くり返させるものである。マクロ機能はコマンドの拡張用で、PLETの様に、記憶のし易さの為にコマンド数を少なく設計したエディタの場合欲しい機能である。図13にくり返しとマクロの例を示す。

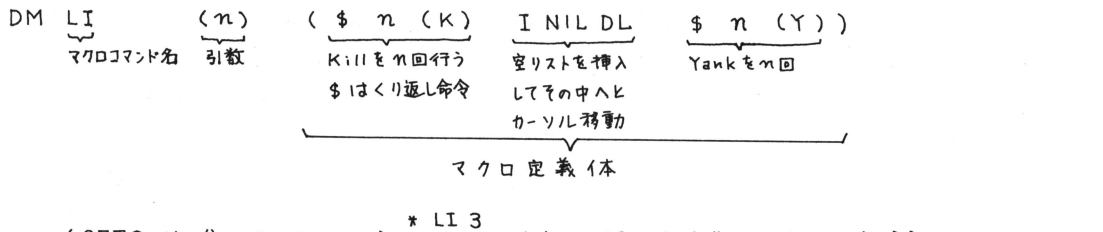


図13 PLETのくり返しとマクロ

PLETはこのほかに、Lisp特有のコマンドをいくつか持っている。例えば、現在編集しているリストの性質（EXPR かFEXPRかなど）を変えるコマンドCT（Change Type）や、そのリストの名前を変えるコマンドCN（Change Name）などがある。特にCNは関数や変数のコピーを作るのに便うことができる。今までふれなかったが、PLETはもちろん関数定義だけでなく変数の値の編集にも使える。これはデバッグの際にテスト値を作ったりするのに大変便利である。またエディタの中でS式を評価する為のコマンドE（Evaluate）も有効で、これらがLispエディタならではの強みといえる。PLETの全コマンドをまとめると下のようになる。

1	カーソル制御コマンド	F，B，UL，UR，DL，DR，S，RS
2	リスト構造修飾コマンド	D，I，R，K，Y，YC
3	各種プリントコマンド	P，PP，PPP
4	くり返しおよびマクロ機能	
5	その他	UNDO，CT，CN，E，HELP

PLETのコマンド体系は単純ではあるが、機能的には必要十分で統一がとれており、筆者は標準的なものと考えている。PLETは現在、ユタ大学で開発された、Standard Lisp の上にのせられ、DEC SYSTEM 20 およびFACOM，HITACのMシリーズの上で走り、システムの開発や移殖に大きな効果をあげている。なお図14にPLETのサンプルセッションを示す。

謝辞　この原稿を書くにあたって適切なアドバイスを再三にわたって与えていただいた京都大学数理解析研究所の中島玲二助教授並びに湯浅太一氏に、また、PLETのユーザの立場から多くの有益な指摘を受けた、京都大学の萩野達也氏と柴山悦哉氏に、深く感謝致します。

Reference

(1) EMACS Reference Guide.
(2) W. Teitleman et al., INTERLISP Reference Manual. XEROX
(3) G. Huet et al., Environnement de programmation Pascal, Manual d'utilization. sous siris 7/8, IRIA, 1979.
(4) R. Nakajima, T. Yuasa and K. Kojima, The ν Programming System — A Support System for Hierachical and Modular Programming —, Proc. of IFIP Congress 1980.
(5) T. Yuasa, Syntax-directed module editing (to appear)

```
@SL

SL

(VERSION 10)

*(TV NTH)
NEW GENERATION!
WHAT TYPE? (EXPR OR FEXPR OR MACRO OR VALUE)
#EXPR
#P
(LAMBDA /\)

#(X N) P
(LAMBDA (X N) /\)

#(COND ((EQN N 1) X))(T (NTH (CDR X) SUB1 N)] PP
(LAMBDA (X N)
 (COND ((EQN N 1) X)) (T (NTH (CDR X) SUB1 N)) /\)

#P
(LAMBDA (X N) (COND &) (T &) /\)

#B K DL Y PP
(COND ((EQN N 1) X) /\ (T (NTH (CDR X) SUB1 N)))

#D PP
(COND ((EQN N 1) X) /\)

#UNDO PP
(COND ((EQN N 1) X) /\ (T (NTH (CDR X) SUB1 N)))

#DR PP
(/\ T (NTH (CDR X) SUB1 N))

#DR F F P
(NTH (CDR X) /\ SUB1 N)

#K K NIL DL Y Y PP
(/\ SUB1 N)

#PPP
(LAMBDA (X N)
 (COND ((EQN N 1) X)
       (T (NTH (CDR X) (/\ SUB1 N)))))
#OK

NTH

*(NTH '(A B C D) 3)

(C D)

*
```

PLETはLisp関数TVを介して呼ばれる。"#"はエディタのプロムプトである。

PLETのコマンドリストは、すべてそのまま挿入される。カーソルは挿入されたS式の次に来る。

コマンド"OK"によってNTHが登録されると共にエディタからぬける。

'(A B C D)は、(QUOTE (A B C D))の略記

図14　PLETのサンプルセッション