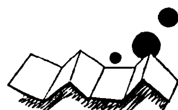
解 説



LISP 構造エディタ†

長谷川 洋††

1. はじめに

LISP[1] は記号処理を円滑に行うために開発されたリスト構造を処理対象とするリスト処理言語であり，ラムダ計算法に基づく強力な計算機能や単純な構文，そして動的記憶管理などによるプログラミングのし易さから，記号処理が大きな比重を占める人工知能の研究などでは欠くことのできないプログラム言語となっている．

LISP の基本設計方針の1つに，プログラムはデータ構造として表現すべきであるという考え方があり，その結果プログラムとデータを共通のリスト表現によって表わす表記法を採用している．このリスト表現はS式 (S-expression) とよばれ，対応する左右の括弧を多重に使用して入れ子構造を表わし，また，それはS式の LISP システム内部の表現であるリスト構造(木構造) と1対1に対応している．

括弧を用いた入れ子構造によるプログラムの表現は多数の括弧の存在により非常に見にくく読みにくいとの定評を得ている．このため LISP プログラムを文字単位で編集した場合には，プログラムの修正に注意を注ぐのと同時に左右の括弧の対応をとって正しい入れ子構造を保つ努力をしなくてはならず，編集作業の効率を悪くするばかりでなく誤りを生ずる原因ともなってしまう．したがって LISP プログラムの編集に当っては，文字単位による編集ではなく，その代りにS式の表わす構造にそって，左右の括弧の対応を取る作業から解放された状態で行える方が望ましく，また自然なものと言えよう．

このような構造的編集を行う上で，代表的な LISP システムである MACLISP[2] と INTERLISP[3]ではそれぞれ異なった方法で対処している．まず MACLISP では，LISP プログラムのテキストが存在するソースファイルを汎用のスクリーンエディタである EMACS[4] を用いて編集する方法をとっている．S式の構造にそった編集を実現するために，テキストファイルの文字列上でS式を走査して構造を取り出し，それに対して編集作業を行うコマンド群が EMACS に用意されている．

これに対し INTERLISP では，LISP インタプリタによって LISP システム内部に読込まれたS式の内部表現を直接操作する構造エディタによって編集を行っており，テキストファイルを対象とした編集は行っていない．

本稿では INTERLISP のエディタを中心に LISP 構造エディタの特徴や基本構造について述べ，他のエディタとの比較を行う．最新の INTERLISP-D[5] には，テレタイプ端末での使用を前提として開発された旧来の INTERLISP のエディタと，従来のエディタの機能の一部にビットマップディスプレイ向きのユーザインタフェースを付加した DEDIT とよぶ2種類の構造エディタがある．しかし，本稿ではこれらに共通する構造エディタの基本構造についてのみ述べることとし，ビットマップディスプレイ向けユーザインタフェースについては触れないことをお断りしておく．

2. 特　徴

LISP の処理系はインタプリタ形式を基本としており，LISP システム内部に読込まれたプログラムは，その外部表現であるS式と1対1に対応するリスト構造（木構造）で表わされている．これと同様にS式で表わされたデータもまたシステム内部ではS式と1対1に対応するリスト構造（木構造）として表わされている．このようにプログラムとデータが同一の構造によって表わされることが LISP の特徴の1つであり，プログラムとデータの同一性とよばれる．したがってこの性質を利用すれば，LISP システムに読込まれた LISP プログラムの木構造をデータとして扱い，そのポインタを直接つけ変えることによってプログラムの

† LISP structural editor by Hiroshi HASEGAWA (Programming Research Section, Computer Science Division, Electrotechnical Laboratory).
†† 電子技術総合研究所ソフトウェア部プログラム研究室

構造を変える LISP プログラムを作成することができる．このS式の構造にそって編集を行う LISP 専用エディタを LISP 構造エディタ（以後 LISP エディタと記す）とよび，LISP エディタの特徴として次のものを挙げることができる．

（1） プログラムとデータの同一性のため，LISP エディタ自身を LISP で記述することが非常に適している．またこのため保守・改造が容易である．

（2） （1）と同様に LISP で記述できることから，編集作業によって発生する不要セルの処理は LISP システムの記憶再生システム（garbage collection）に任せることができる．このためテキストエディタの場合とは異なりファイルのレコード管理をする必要がなく，LISP エディタを小型化することができる．

（3） テキストファイルを編集対象とするテキストエディタでは，プログラムの実行はエディタとは別のシステムである言語処理系で行わなくてはならない．これに対して LISP エディタは LISP システム内部に存在するため，プログラムの編集中にプログラムの実行を行うことができる．

一般に，上記の特徴の（3）であるプログラムの編集と実行を同一システム内で行うことができるエディタを常駐型（residential）エディタ[6]とよび，プログラムの内部表現すなわち木構造を直接の編集対象とするため，必然的にその言語専用の構造エディタとなっている．しかし，特徴の（1）と（2）に関しては言語処理系の作成方式に依存するものであり，現在のところ LISP エディタ固有の特徴と言うことができる．

3. 編集コマンドとその使用例

LISP エディタでは LISP システム内部に存在する木構造を直接の編集対象として，その木構造を形成するポインタを付け変えて行くことで編集が行われる．そのため LISP エディタには，まず，

（1） 木構造をたどり望む任意の部分木に到達する機能，

（2） ポインタを付け変えて構造を変更する機能，

が必要である．また，エディタの使用者に対しては，

（3） 任意の部分木を見やすいS式で印字する機能，

を用意しておかなくてはならない．この他にも，常駐型エディタのもつ強力な特徴である，

（4） プログラムを実行する機能，

を欠くことはできない．

表-1 INTERLISP エディタの主要編集コマンド（n, m は自然数を表わす）

1. 注目S式変更コマンド（木構造をたどる機能）
 〔a〕 局所的移動
  n ：注目S式の前から n 番目の要素へ行く．
  $-n$ ：注目S式の後から n 番目の要素へ行く．
  0 ：1段上のS式へ行く．
  ! ：最上段のS式へ行く．
  NX：注目S式の右隣りのS式へ行く．
 〔b〕 探　索
  F pattern：S式を探索する
2. 構造変更コマンド（ポインタをつけ変える機能）
 〔a〕 S式の構成要素を対象とするもの
  (n)　：n 番目の要素の削除．
  (n e_1, …, e_m)：n 番目の要素を e_1, …, e_m と置換．
  ($-n$ e_1, …, e_m)：n 番目の要素の前に e_1, …, e_m を挿入．
  (N e_1, …, e_m)：注目S式の終りに e_1, …, e_m を付加．
 〔b〕 括弧を対象とするもの
  (BI n m)：n 番目の要素の前に左括弧，m 番目の要素の後に右括弧を挿入．
  (BI n)：=(BI n n)
  (BO n)：n 番目の要素の両括弧を削除．
  (LI n)：n 番目の要素の前に左括弧，最後の要素の後に右括弧を挿入．
  (LO n)：n 番目の要素より左括弧を取り，$n+1$ 番以降の要素を削除．
  (RI n m)：n 番目の要素中の m 番目の要素の後に右括弧を挿入．
  (RO n)：n 番目の要素より右括弧を削除．
 〔c〕 全 置 換
  (R x y)：注目S式中のすべての x を y に変える．
 〔d〕 復　元
  UNDO：直前に実行された構造変更コマンドによる変更を元に戻す．
3. 印字コマンド
  PP：プログラム全体の清書．
  P：注文S式を深さ2レベルまで印字．
  ?：注文S式を深さ100レベルまで印字．
4. 評価コマンド
  E：続く2入力を評価する．
5. 雑コマンド
  OK：エディタから出る．

LISP エディタの設計者はこれらの機能を満足し，また利用者が使い易いコマンドを考案しなくてはならない．例として LISP エディタを代表する INTERLISP のエディタを取上げ，そのコマンドのうち主要なものを **表-1** に示す．木構造をたどり選択した部分木はエディタの利用者が現在注目しているS式であるので，注目S式（current expression）とよぶ．利用者は，注目S式変更コマンドによって選択した注目S式を，構造変更コマンドによって修正して行くことになる．

さて，自然数の階乗を求める関数 FACTORIAL を編集する模様を **図-1** にそってながめてみよう．端

```
EDITF (FACTORIAL)                                    ……①
*PP                                                  ……②
(LAMBDA (N)
  (COND (ZEROP N 1)
        (T (TIMES N (FACTORIAL SUB1 N M)))))
*3 P                                                 ……③
(COND (ZEROP N 1) (T &))
*2 P                                                 ……④
(ZEROP N 1)
*(BI 2) P                                            ……⑤
(ZEROP (N) 1)
*(BI 1 2) P                                          ……⑥
((ZEROP (N)) 1)
*UNDO UNDO P                                         ……⑦
(ZEROP N 1)
*(BI 1 2) P                                          ……⑧
((ZEROP N) 1)
*F FACTORIAL P                                       ……⑨
(FACTORIAL SUB1 N M)
*(4) P                                               ……⑩
(FACTORIAL SUB1 N)
*(BI 2 3) P                                          ……⑪
(FACTORIAL (SUB1 N))
*PP                                                  ……⑫
(LAMBDA (N)
  (COND ((ZEROP N) 1)
        (T (TIMES N (FACTORIAL (SUB1 N))))))
*E FACTORIAL (4)                                     ……⑬
24
*OK                                                  ……⑭
```

(*はエディタの入力促進記号，& は S 式の省略表示を表わす)

図-1 編集例

末から INTERLISP システムに直接入力されたか，あるいはテキストファイルから読込まれた FACTORIAL の関数定義は，性質リスト (p-list) に格納される．関数を編集する関数 EDITF は性質リスト上に存在する FACTORIAL の関数定義を捜し出し，これによって関数 FACTORIAL の編集が始まる(①)．この時，関数定義全体が注目S式となっている．

②で“PP”と入力して FACTORIAL を清書 (Pretty Print) すると，3番目のリスト要素である条件文の中に誤りがあることに気付く．そこで③の注目S式変更コマンド“3”によって条件式に到達すると，条件式全体が注目S式となる（エディタの利用者は，コマンド“P”によって注目S式を見ることができる)．さらに④で条件式の中の第2要素へと進み，(ZEROP N 1) を注目S式とし，この注目S式の2番目の要素Nの両側に括弧を挿入するために“(BI 2)”を入力する (⑤)．この構造変更コマンドによって FACTORIAL の関数定義の木構造は変更され，注目S式は (ZEROP (N) 1) となる．

引続き⑥で“(BI 1 2)”と入力することによって注目S式の第1要素の前に左括弧を，第2要素の後ろに右括弧が挿入され，現在の注目S式は ((ZEROP (N)) 1) となる．しかしこれは望むS式ではない．そこで⑦に見られるように，まず1番目の“UNDO”で⑥で行われた“(BI 1 2)”の効果を打消して注目S式を (ZEROP (N) 1) に回復させ，さらに2番目の“UNDO”で⑤の“(BI 2)”を取消して注目S式を元の (ZEROP N 1) の状態に戻す．⑧では⑥と同じコマンド“(BI 1 2)”によって注目S式は ((ZEROP N) 1) となり誤りは訂正された．

⑨で FACTORIAL というアトムをコマンド“F”で探索すると，FACTORIAL を含む (FACTORIAL SUB1 N M) が注目S式となる．そこで⑩で注目S式の4番目の要素Mを削除し，⑪で2番目の要素 SUB1 の前に左括弧を，3番目の要素Nの後に右括弧を挿入する．これにより⑫で関数全体を清書して修正がうまく行ったことを確認する．

最後に関数の評価に進む．⑬で関数名 FACTORIAL と引数(4)を評価コマンド“E”によって評価すると，4!=24 と正しい結果が得られたので，“OK”と入力してエディタを離れる (⑭)．

4. 内部構造

4.1 注目S式変更コマンドの実現

LISP では木構造を形成するポインタは根 (root) から葉 (leaf) に向っての1方向にしかついていない．そのため，木構造全体から出発してその部分木を注目S式として行く時，すなわち根から出発して葉に向ってポインタを進む時には，木構造をたどった道筋をスタックに保存するのが一般的である．INTERLISP エディタでは **図-2** に示すようにスタックをリストによって実現しており，このスタックのことを編集スタック (edit chain) とよんでいる．編集スタックの先頭のセルの car 部分には注目S式を指すポインタが，また末尾のセルの car 部分には木の根を指すポインタが入れられている．

LISP エディタの利用者が木の中を葉の方向へ進む場合には，新たな注目S式つまり部分木を指すポインタを編集スタックにプッシュ (push) すればよい．この時編集スタックには自由リストから取出されたセルの car 部分に新しい注目S式を指すポインタが入れられ編集スタックの先頭に付加され，このセルが編集

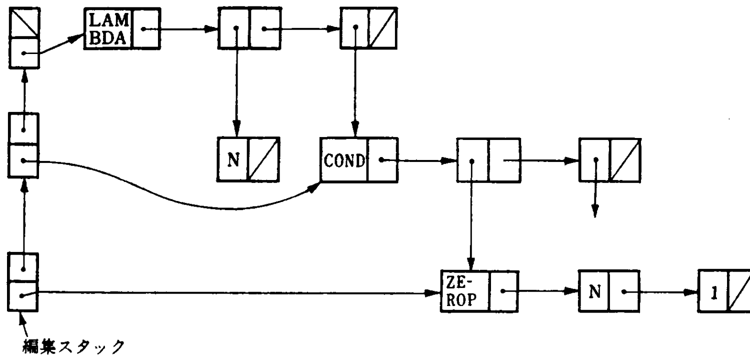
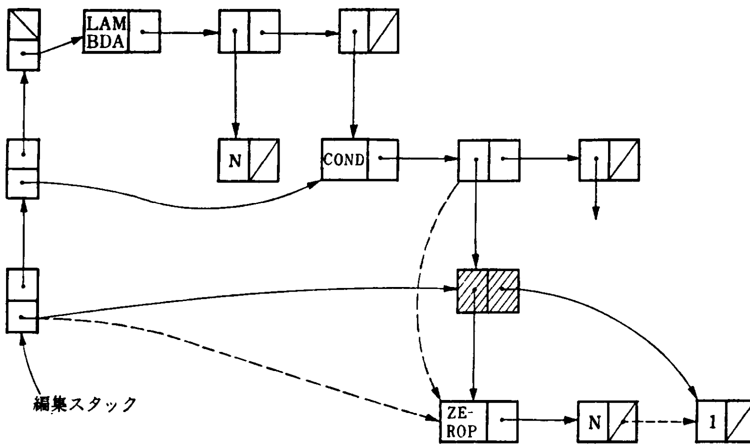


図-2 編集スタック



(破線で示すポインタはコマンドの実行前の状態を表わす)
図-3 構造変更コマンドによるポインタつけ変え

スタックの新しい先頭となる.

また木の根へ向って戻る場合には,編集スタックをポップアップ(pop up)すればよく,ポップアップするたびに1段上のレベルのS式が注目S式となる.この結果,表-1中の木構造を局所的にたどる機能は容易に実現できることになる.

さて,任意のS式へ即座に到達するためのコマンドとして“F”(Find)がある.いま編集する関数が

```
(LAMBDA (X)
  (PROG (Y)
    (COND (X (……)))
      ⋮
    (COND (Y (……)))))
```

となっているとしよう.まず“F COND”とLISPエディタに入力すると,注目S式は(COND (X (…)))となる.引続き“F COND”と入力すれば,注目S式は次の COND 文である(COND (Y(……)))となり,この注目S式が編集スタックの先頭に置かれる.エディタの利用者が見ているS式は,木構造を前順序(preorder)でたどって印字したものであるので,木構造上での探索は前順序に行えばよいことになる.また探索用のスタックとしては編集スタックを使用すればよく,別にスタックを用意する必要はない.

4.2 構造変更コマンドの実現

構造変更コマンドはリスト構造のポインタを関数RPLACA, RPLACD によって直接つけ変え,注目S式中の要素の削除・置換え・挿入・付加などの修正作業を行う.また,構造変更コマンドは“UNDO”の場合を除けばどれも,注目S式の構造は変えても編集スタックの長さを変えることはない.(“UNDO”については後述する.)

さてここでは,図-1の例⑧で行われた括弧の挿入の模様をながめてみよう.いま注目S式は(ZEROP N 1)であり,これをコマンド“(BI 12)”によって注目S式の1番目の要素の前に左括弧を,2番目の要素の後に右括弧を挿入して,注目S式を((ZEROP N) 1)に変えようとする.この時,**図-3**で斜線で示す1セルが自由リストより取出され,つづいてポインタの付け変えが行われる.この結果編集スタックの先頭の car 部分のポインタが指す部分木すなわち注目S式は((ZEROP N) 1)となる.

4.3 UNDO の実現

コマンド“UNDO”はこれまでに行われた構造変更コマンドによる木構造の変更を元の状態に戻すコマンドであり,また“UNDO”を連続して入力することにより,過去の構造変更の履歴を逆上り,入力した“UNDO”の回数分だけ前の状態に戻ることができる.

変更された構造を再び元の状態に戻すためには,

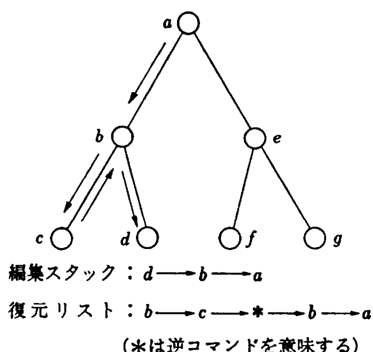


図-4　コマンド UNDO の実現

(1)構造変更が行われた時点での注目S式の位置にまで編集スタックを戻し，(2)その注目S式に対して構造を元の状態に回復する処理を行わなくてはならない．ここでは文献7) で採られた実現法について述べよう．

さて，これまでに **図-4** に示すように，木構造の根 *a* より出発し，節 (nodo) *b* を経て節 *c* で構造変更を行い，その後節 *b* へ戻り，現在，節 *d* が注目S式になっているとする．したがってこの時編集スタックは

$$d \to b \to a$$

となっている．節 *c* で行われた構造変更を元に戻すためには，その構造変更が行われた時点での編集スタックを再現しなくてはならない．そのため編集スタックとは別に復元リスト (undo list) とよぶスタックを作成しておく．復元リスト上には2種類の情報が存在する．1つは過去の注目S式の履歴であり (正確には，編集スタックの先頭のポインタの履歴であり)，他は構造変更コマンドの逆コマンドが存在している．この逆コマンドとは，たとえばコマンド "(BI 1 2)" に対しては，注目S式の1番目の要素の左右の括弧を取りはらうコマンド "(BO 1)" を作成しておくことを意味する．この結果，節 *c* で行われた構造変更コマンドの逆コマンドを＊で表わせば，復元リストは

$$b \to c \to * \to b \to a$$

になっている．したがって復元リストに保存されている過去の木構造をたどった履歴と現在の編集スタックとをつき合せることによって，構造変更が行われた時点の編集スタックと復元リストを復元することが可能となる．この結果，編集スタックと復元リストはそれぞれ

編集スタック：$c \to b \to a$

復元リスト：$* \to b \to a$

となり，復元リストの先頭にある逆コマンド＊を実行することにより，元の状態を回復することができる．

ここで1つ注意しておかなくてはならないのは，注目S式中のすべての x を y に置き換えるコマンド "(R x y)" についてである．いま，x と y が共にアトムであり，木を構成するアトムがすべて x と y から成っていたとする．この時コマンド "(R x y)" によりすべてのアトム x は y に置き換わるが，これに対して作成される逆コマンド "(R y x)" を実行した場合には，木のアトムはすべて x になってしまい元の状態を回復することはできない．

4.5　清書プログラム

LISP には貧弱な入出力機能しかなく，特に LISP エディタのように木構造を直接変更して行く場合には，利用者にS式を見易く段付けして印字する (Pretty Print) 機能が不可欠といえる．この清書プログラムは木構造を文字列に変換することから逆パーサととらえることができ，また，よい清書プログラムを作成することはエディタ本体を作成するよりも難しい．

清書プログラムについては種々の文献が存在するが，LISP の清書プログラムについては具体的なプログラムが示されている文献8) が参考となろう．

5.　LISP エディタの欠点と他方式との違い

INTERLISP エディタは最も早期に開発された常駐型の構造エディタであり，その後の構造エディタの開発に大きな影響を与えている．しかし，LISP プログラムを編集する上で構造的編集を行うという点に関しては異論のないものの，LISP のエディタとして常駐型の構造エディタが適当であるか否かについては依然として意見の分かれるところとなっている．**1.** でも述べたように，代表的な LISP システムである INTERLISP と MACLISP とではまったく異なった編集方式を採用している．常駐型構造エディタを有利とする Sandewall[6] に対し，汎用テキストエディタをよしとする Stallman[9] は構造エディタの欠点とテキストエディタの利点を強調しているが，この中で本質的な点は次の2点であろう．

(1)　テキストファイルではコメントが容易に貯えられ，利用者の好むようにフォーマットを決めることができる．

(2)　LISP 構造エディタは専用エディタであるため LISP 以外の編集に適していない．

(1)のコメントについては常駐型システムに常に付随する面倒な問題であり，また(2)については必然的

に専用エディタとなる常駐方式とそうでない非常駐方式との間での有利さの問題となるため意見が分かれてしまう．(1)と(2)に関する Sandewall の見解は Sandewall[6] に詳しく記述されているので参照されたい．

Stallman の批判の他にも LISP 構造エディタの欠点として，次のようなものがあろう．

(1)　常駐型 LISP 構造エディタでは，アトムの文字列の部分的修正がしにくく，また効率の悪い方法でしか実現できない．

(2)　文字ファイルにプログラムを入力し，編集だけを LISP システム内の構造エディタで行うというやり方は不自然である．

後者の(2)に関しては，プログラムやデータを始めから LISP システム内で構造的に入力する Pretty Reader[10]を構造エディタに組込み，入力から評価，出力までを構造エディタの中で行うべきであると筆者は考えている．

ここで少し視点を変えて，構造エディタが構築される言語処理系による違いについて述べよう．

LISP エディタでは，その編集対象である LISP 言語の言語処理系はインタプリタベースであり，LISP エディタはインタプリタが解釈実行するプログラムの内部構造を直接操作して編集している．この結果，LISP エディタ内部での，S式の読込みや評価，さらには編集によって生じる不要セルの回収などに関しては LISP システムのもつ機能を利用できるため，エディタの構造が単純となる．

これに対して，MENTOR[11]，Cornell Program Synthesizer[12]，ALOE[13]といった構造エディタは，PASCAL, PL/I サブセット，Cなどのブロック構造をもつ言語を扱い，コンパイラをベースとした言語処理系上に構築される．これらの構造エディタは，プログラムの解析木を編集対象として，“いかに構文的に正しいプログラムを作成するか”という点に重点を置いているため，構文指向 (syntax-directed) エディタとも呼ばれ，コンパイラの構文解析部であるパーサが発展したものという見方ができる．常駐型エディタでは，利用者がエディタと会話しながら，プログラムの修正と実行とを交互にエディタの中で行うが，この種の処理にはコンパイラは不向きである．そのため，構文指向エディタ内でのプログラムの実行に関しては，コンパイラとは別に，インタプリタを作成したりインクリメンタルコンパイル (incremental compile) を行って対処している．

なお，構文指向エディタを概観するには Meyrowitz[14]が参考となろう．

6. 今後の方向

今後，プログラム編集用のすべてのエディタが目指すべき方向として，

(1)　ビットマップディスプレイを用いた画面指向化，

(2)　統合プログラミング環境の実現，

(3)　機能の高級化，

の3点を挙げることができよう．

ビットマップディスプレイによる画面指向化の動きは，LISP に関しては Teitelman[15] や Sproull[16] による研究段階を経て，既に INTERLISP-D や LISP マシンでの実用化が始まっている．

(2)の統合プログラミング環境の実現に当っては，ユーザインタフェースが統一されることが極めて重要である．これは各サブシステムをウインドウ別に表示すれば解決するものではない．現在の INTERLISP-D のようにエディタやデバッガなどが別々のサブシステムとなっている状態では利用者はサブシステムごとに異なったユーザインタフェースと対面しなくてはならず，この点の改善を計らなくてはならない．

最後に，機能の高級化の方向には種々のレベルが考えられるが，大別してソフトウェア工学の成果を導入する方向と，プログラミングを人工知能や知識工学の対象と考える方向がある．前者では例えばモジュール管理の機能を組込むことなどが挙げられる．後者の方向としては，問題領域やプログラミングの知識を利用したエディタやプログラミングシステムそして自然言語によるユーザインタフェースなどを挙げることができる．現在これらに関する研究が活発化しつつあるが，その方向をつかむ上で Barstow[17] が手助けになると思われる．

参考文献

1) MacCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine-I, Comm. ACM Vol. 3, No. 4, pp. 184-195 (Ap. 1960).
2) Pitman, K. M.: The Revised MACLISP Manual, MIT-LCS-TR-295 (June 1983).
3) Teitelman, W. et al.: INTERLISP Reference Manual, Xerox Corp. (Oct. 1978).
4) Stallman, R. M.: EMACS Manual for

TWENEX Users, MIT-AImemo 555 (Oct. 1981).

5) Teitelman, W. et al.: INTERLISP-D Reference Manual, Xerox Corp. (Oct. 1983).

6) Sandewall, E.: Programming in an Interactive Environment: The Lisp Experience, ACM Computing Surveys, Vol. 10, No. 1, pp. 35-71 (Mar. 1978).

7) 長谷川洋：LISP EDITOR について，情報処理学会第14回全国大会 (Dec. 1973).

8) 長谷川洋：LISP Pretty Printer, 情報処理学会記号処理研究委員会報告集 1976 (Mar. 1977).

9) Stallman, R. M.: in Surveyer's Forum, ACM Computing Surveys, Vol. 10, No. 4, pp. 505-506 (Dec. 1978).

10) 長谷川洋：LISP Pretty Reader について，情報処理学会第17回全国大会 (Nov. 1976).

11) Donzeau-Gouge, V. et al.: A Structure-Oriented Program Editor: a First Step Towards Computer Assisted Programming, Proc. Int'l Computing Symposium, pp. 113-120 (June 1975).

12) Teitelbaum, T. and Reps, T.: The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, Comm. ACM. Vol. 24, No. 7, pp. 563-573 (Sept. 1981).

13) Medina-Mora, R.: Syntax-Directed Editing: Towards Integrated Programming Environments, CMU-CS-82-113 (1982).

14) Meyrowitz, N. and Dam, A.: Interactive Editing System: Part II, ACM Computing Surveys, Vol. 14, No. 3, pp. 353-415 (Sep. 1982).

15) Teitelman, W.: A Display Oriented Programmer's Assistant, CSL-77-3, Xerox PARC (Mar. 1977).

16) Sproull, R. F.: Raster Graphics for Interactive Programming Environments, CSL-79-6, Xerox PARC (June 1979).

17) Barstow, D. R. et al.: Interactive Programming Environments, McGraw-Hill (1984).

(昭和59年6月11日受付)