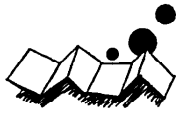
解　説



Lispのプログラミング環境†

奥乃　博††　丸山　宏†††

1. はじめに

プログラミング環境とはプログラマを取り巻き，プログラマと何らかの関係をもって直接あるいは間接に影響を与える外界を意味する．プログラミング環境にはさまざまな側面がある．たとえば，居室は個室か，グループの他のメンバの性格や能力は，上司のコントロールは，などの人間的側面や，居室と実験室は別か，端末を置く机や椅子の高さは，照明や騒音は，といった物理的側面から，プログラマは少なからず影響を受ける．しかし，何にもまして影響を受けるのはプログラミング・ツールから構成される計算機環境である．プログラミング・ツールは本来ユーザと計算機との間で行われる対話を手助けするものであるから，よいプログラミング・ツールが提供されていると計算機の使い勝手は大いに向上する．本稿ではプログラミング環境をプログラミング・ツールの集成としての計算機環境という意味に限定して用いる．ただし，同じ言語を使用しているコミュニティは含まれるものとする．

プログラミング環境[4]の重要性は多くの分野で認識されている．Interlisp のプログラミング環境については優れた解説[17]がある．また，医療診断エキスパート・システム MYCIN プロジェクトが成功した一つの理由としてプログラミング環境の充実したInterlispを使用したことが挙げられている[18]．世界初の本格的コンピュータ・グラフィックスとして評判を呼んだ映画『トロン』の製作においても MacLisp の対話機能を活用したシステム ASAS[16]が使用された．

プログラミング環境は Lisp の専売特許ではないことは言うまでもない．ADA では言語仕様だけでなくプログラミング環境も標準化するべきであるという提案書 STONEMAN が出されている．プログラミング環境の整備はソフトウェアの生産性の向上，信頼性の向上および高い移植性がその目的である．具体的な項目として次のような目標が設定されている[21]．

（1） 開発段階と保守段階でのサポート
（2） ADA 自身によるツールの開発
（3） ADA 用プログラム・ライブラリの完備
（4） プロジェクト・チームに対するサポート

これらの中核となるのは，コンパイラ，テキスト・エディタ，ライブラリ，ローダ，コマンド言語プロセッサ，記号デバッガ，データベース管理プログラムといったツールである．STONEMAN ではツールの整備に重点が置かれ，対話型でしかもそれらが統合された形で提供されるという視点は Lisp 程明確ではない．

人工知能研究では対話型プログラミング環境が重要な役割を果たす[14]．人工知能研究は人間の情報処理過程のモデル化を通じて知的処理システムを構築して行こうという研究分野である．この分野の研究の中心は，概念・モデルの形成およびそれをもとにした処理アルゴリズムの開発であり，そのアプローチとして仮説を立て，それをシミュレーションで検証するという試行錯誤の方法論がとられる．言い換えると，研究者はプログラミングによって実験を行っているのである．実験を行う上で，充実したプログラミング環境は研究者のプログラム作成を支援するだけでなく，いわば研究者の思考の手助けも行っている．つまり，プログラミング環境はプログラマの思考過程に対する知的増幅器 (Intelligence Amplifier, IA) としての役割を担ってきたわけである．

Lisp は知的増幅器の第一段階として位置付けられる．このゆえに，人工知能研究で主要言語として使用されてきた．もちろん，現在のプログラミング環境では知的増幅器としてはまだまだ不十分な機能しか提供されていないことは言うをまたない．それでも，他の言語よりデバッグ・サイクルが早いのでプログラムの生産性は向上する．

---

† Lisp Programming Environments by Hiroshi G. OKUNO (Musashino Electrical Communication Laboratories, N. T. T.) and Hiroshi MARUYAMA (Science Institute, IBM Japan, Ltd.).

†† NTT 武蔵野電気通信研究所
††† 日本 IBM(株) サイエンス・インスティチュート

本稿では Lisp のプログラミング環境を構成するさまざまなツールについて概説する．第2章でエディタ，第3章でデバッギング・ツール，第4章でその他のツールについて概観する．コンパイラは実行高速化のためのツールではあるが，処理系作成技法と密接な関係があるので本稿では割愛する．本稿で取り上げた Lisp システムは，Interlisp[22),23)] (−D)[8)]，MacLisp[15)] Zetalisp[13)]，VAX Lisp[26)]，NIL[5)]，Franz Lisp[7)]，UTI Lisp[6)]，PSL[25)]，Lisp/VM[1),11),12)]などのいわゆるメジャ Lisp である．しかし，マイコン上の Lisp にも同じような機能を提供しているものもある．また，本稿で取り上げた機能の多くは他の言語にも応用できる．

2. エディタ[9)]

プログラミング・ツールの中で最も使用頻度が高く，使用時間の長いのがエディタである．たとえば，エール大学の計算機使用時間 (elapse time) の 50% 強がエディタ使用に費やされている．Lisp で使用されるエディタはエディットする対象のモデルから構造エディタとテキスト・エディタに，また，実現方法から独立ユーティリティか否かで独立型か常駐型に分けることができる．

構造エディタはプログラムの構造を反映しているエディタであり，Lisp や Pascal 用に作成されている．Lisp 用構造エディタは S 式単位のエディット・コマンドから構成されている．たとえば，カーソルの移動はリストたどりに，データの変更は S 式の変更に相当する．Lisp 用構造エディタは Lisp での操作と親和性がよいので Lisp 自身で作成するのが最も容易である．したがって，構造エディタは Lisp の中に組み込まれた常駐型システムであることが多い．常駐型構造エディタ*では『見えているものはいつでもそこにある (What you see is what you get.)』[11)]という原則が成立し，エディタの表示と Lisp が保持するデータとは常に一致する．つまり，構造エディタのコマンドで与えられるデータ（S式）は構文チェックを受けた正しいデータであるので，括弧の不足や過多は生じない．

構造エディタでの表示は括弧のレベルに応じて字下げが行われたプリティ・プリントの出力である．しかし，プリティ・プリントの字下げや注釈のつけ方はシステムが定めたものであり，ユーザが制御できないことが多い．また，構造エディタで作成・修正した関数定義やデータは Lisp 中の内部データに対して変更が行われるだけで，ファイルに対して変更が反映されない．このために，何らかのユーティリティが必要となる（たとえば，後述するファイル・パッケージ）．

テキスト・エディタは文字単位のエディット・コマンドから構成される．したがって，カーソルの移動やデータの変更は任意の場所で行える．さらに，エディットする対象の構造に応じた構造コマンドを提供しているエディタも多い．たとえば，テキストのエディットの場合には単語・文・パラグラフ単位のコマンドが，プログラムのエディットの場合には Lisp だと S 式単位のコマンドが使用できる．各種構造コマンドが提供されているテキスト・エディタでは，そのコマンドさえ覚えておけばあらゆるものをエディットすることができるので，一つのプログラミング言語にしか使えない構造エディタと異なり，極めて便利であり，かつ，一種類のコマンドさえ覚えればよいのでユーザの負担も少ない．さらに，このようなテキスト・エディタは構造エディタの 95% の機能を包含しているという報告もある[27)]．ただ，テキスト・エディタでは『見えているものはいつでもそこにある』という構造エディタで成立する原則は保証されないので，括弧の不足・過多や不一致というエラーが生じやすい．これを防ぐためにさまざまな機能が提供されている．

常駐型エディタでは Lisp が提供している全機能，全ユーティリティが使用できるので，それらと組み合わせて高度な機能を提供することができる．たとえば，エディット中に関数やデータの定義を見たり，関数の引数の個数や種類を問合せたりすることができる．独立型エディタでこのような機能を直接実現することは難しい．もちろん，オペレーティング・システム*が提供する機能を活用すれば上述の機能を実現することは可能である．

以下，二種類のエディタについて具体的なシステムの概要を述べる．

2.1 構造エディタ

(1) Interlisp (−D)[8),22),23)]

Interlisp の構造エディタは Lisp 構造エディタのひな型であり，同じものあるいはそのサブセットがさまざまな Lisp システムで提供されている．主なエディット・コマンドを**表-1**に示す．

プログラムを構造という面だけでとらえ，テキストと見なさないと，エディット操作にある種の歯がゆさ

* 以下，単に構造エディタと呼ぶ．

* 以下，OS と略記する．

表-1 構造エディタの主なコマンド

0	一つ上のレベルへカーソル移動
n	リストの*n*番目の要素へカーソル移動
−*n*	リストの後ろから*n*番目の要素へカーソル移動
∧	トップ・レベルへカーソル移動
F〈パターン〉	パターンを探索し，カーソルを移動
BF〈パターン〉	パターンを逆向きに探索し，カーソルを移動
?	プリント
P	深いレベルは省略してプリント
PP	プリティ・プリント
E〈式〉	式を評価
UNDO	最後の修正の取消し
(B〈S式〉…)	カーソルの前に挿入
(A〈S式〉…)	カーソルの後ろに挿入
(: 〈S式〉…)	カーソルのあるS式を置換
(*n*)	*n*番目の要素を削除
(*n*〈S式〉…)	*n*番目の要素を置換
(−*n*〈S式〉…)	*n*番目の要素の前に挿入
(BI *n m*)	*n*番目の要素から*m*番目の要素までを一つのリストにする
(BO *n*)	*n*番目の要素の両括弧を削除
(R *x y*)	すべての*x*を*y*で置換
OK	エディット終了

を感じる．その一つが文字列レベルでの修正であり，もう一つがカーソル移動である．たとえば，次の関数

```
(FACT
  [LAMBDA (N)
    (COND
      ((GREATERP N 0)
        N*(FACT N-1))
      (T 1)])
```

をエディットしており，カーソルが N-1 のところにあるとしよう．このとき，カーソルをTのところへ移すには，リストを3レベル上ってからTのところまで下りてくる必要がある（具体的には，000-11というコマンド系列）．

Lisp マシン・システムである Interlisp-D ではビットマップ・ディスプレイとマウスを活用した構造エディタ DEDIT が提供されている．DEDIT ではマウスによってカーソル移動が行えるので上述したカーソル移動における構造による制約が解消でき，直接操作性という利点が得られる．プログラムには構造という見方と，テキストという見方の二面性が存在する．この二面性は DEDIT になって始めて提供されるようになった．Interlisp-D ではウィンドウをいくつでも開くことができ，かつ任意のウィンドウの任意のS式をエディタ中の任意の場所へコピーできる．

Interlisp では，この他に DED[3]という画面用構造エディタが作成されている．DED ではカーソルのあるS式の括弧の組が高輝度で表示される．また，一画面にできるだけ多くの情報を表示するために，画面全体に入りきらないときには，カーソルのある部分は詳しく，カーソルから遠い部分は省略して簡単に表示する自動ズーム機能が備っている．

Interlisp からは後述するテキスト・エディタである画面エディタ Emacs[19] も使える．Interlisp から直接 Emacs を起動でき，また構造エディタの中から呼ぶこともできる．後者では，カーソルのあるS式が Emacs の画面にプリティ・プリントして表示されるので，文字列に空白を挿入して分割したり，文字列を修正したりでき，また，ファイルから読込んで一部を抜き出して挿入することもできる．さらには，履歴をエディットすることもできるので，以前の入力や出力された結果が再利用可能となる．

（2） Lisp/VM[1),11),12)]

Lisp/VM では LISPEDIT と呼ばれる画面を用いた構造エディタが提供されている．LISPEDIT ではカーソルをフォーカスと呼び，フォーカスのあるS式全体が高輝度で表示される．もし，S式全体が画面に入りきらない場合には，フォーカスのある近傍は詳しく，カーソルから遠くなるにつれて省略形で表示し，S式全体が画面内に入るように工夫する．省略された部分は一つのS式だと '&' と，複数のS式だと '…' と表示される．フォーカスの移動はカーソル移動キーで行えるので，構造エディタ固有のカーソル移動のもどかしさはない．

LISPEDIT は構造エディタであるので '(' を入力すると同時に ')' が挿入される．ユーザが入力した ')' はフォーカスの移動の意味しか持たない．Lisp/VM はシステム 370 の VM/CMS の下で走るので端末との会話は半二重で行われる．したがって，入力は行端位となり，一字入力するごとに何らかの情報（括弧のレベルや引数の種類についての情報）を受けとる機能を実現することは難しい．S式の探索は Interlisp と同様パターンが使えるほかに，述語によるものもある．述語は1変数の関数であり，各S式に対して述語が適用され，述語を満足するS式が見つかるまで探索を行う．

2.2 テキスト・エディタ

（1） Emacs[19),9)]

Emacs は画面を利用したテキスト・エディタであり，Lisp システムとは独立したユーティリティである．MacLisp 等米国東海岸系の Lisp コミュニティで

テキスト・エディタが使用されている理由の一つは，プログラムとは構造を持った文字列であるというモデル化の方が，構造が中心であるというモデル化よりもユーザの直感に訴えやすいからである．Emacs の便利な機能としては，エディット中にプリティ・プリントが行える（タブを入力すると前の行にしたがって字下げを行う）機能や，閉じ括弧を入力すると対応する開き括弧へカーソルが一時的に移動して点滅する機能がある．Emacs はテキスト・エディタであるから，プリティ・プリントはユーザの好むような形式で行うことができる．

Emacs と Lisp という2つの独立したユーティリティを OS (TOPS-20, Unix*) の機能を用いて融合させて，常駐型システムと同様の機能を提供することができる．TOPS-20 では MacLisp (Interlisp でも可) の下に Emacs をサブ・プロセスとして起動し Emacs でファイル中の関数を変更するとともにその一部だけを抜き出して Lisp に戻り評価することができる．さらには，評価した結果を再び Emacs に持ち帰ることもできる．また，Emacs に入った時に指定された関数をファイルから読み込み関数定義の先頭へカーソルを移動させることもできる．これらの機能は，プロセス間のコマンドのやりとりと一時ファイルを用いて実現される．Unix では Emacs のウィンドウからプロセスが起動できるので一つのウィンドウをFranz Lisp の Lisp（リスナトップ・レベルの read-eval-print ループ）とすることができる．この結果，以前の入力や出力を再利用したり，修正して新たな入力とする履歴機能が実現できる．これらの機能は OS の機能に依存しているので，OS が充分な機能を提供していない場合や移植性を重視し OS の依存度を最少にしたい場合には，常駐型システム構成をとらなければ実現できない．

多くの Lisp システムで Emacs 風エディタが提供されている．たとえば，NIL[5]では STEVE という，PSL[25] では NMODE という Emacs のサブセットが作成されており，いずれも Lisp 自身で書かれ常駐型システムとして提供されている．VAX Lisp[26]では VMS 標準の EDT に似たエディタが使用できる．

Zetalisp[13]上ではビットマップ・ディスプレイとマウスを活用した Zmacs が提供されている．Zmacs は Emacs の拡張版であり，かつ常駐型システムである．Zmacs で便利なのは，コンパイラが出したエラー・メッセージを別のウィンドウに表示しながらエディットできる機能である．後述するように Zmacs を応用してさまざまな高度なツールが開発されている．Zetalisp ではすべてのウィンドにある S 式がエディットの対象とはなっていないので，エディタ以外のウィンドウに表示されている S 式は他のウィンドウへはコピーできるわけではない．

* Unix はベル研究所が開発したソフトウェアです．

3. デバッギング・ツール

Lisp のデバッギング・ツールは次の4種類に分類することができる．

（1） 実行が中断か停止した状態で使用するツール
　例：ブレイク・パッケージ，デバッガ
（2） 実行させながら使用するツール
　例：トレーサ，ステッパ
（3） データ，関数の内容を調べるツール
　例：インスペクタ，クロス・レファレンス
（4） その他
　例：ファイル・パッケージ，スペル修正，履歴機能

以下，主なツールについて概観する．

3.1 ブレイク・パッケージ

ブレイク・パッケージは '(break)' を実行するか実行中に割込みをかけることで起動される．割込みのかけ方は OS の機能に依存している．OS が一文字入力を許している場合には特定のコントロール文字にブレイク処理ルーティンを起動するように割り当てる．もし，そのコントロール文字が入力されると '(break)' が実行される．ブレイク・パッケージ中では，任意の S 式を中断した環境下で実行できるので計算の途中結果を見たり，無限ループに陥っていないかを調べたりすることができる．また，中断された計算は再開することもできる．もしブレイク・パッケージ中で環境が変更されれば新しい環境の下で再開される．もちろん実行を破棄することもできる．Interlisp-D では，ブレイク・パッケージに入ると Lisp リスナとは別に新たなウィンドウが作成される．これにより，他のシステムでは混在して表示されていた Lisp リスナとブレイク・パッケージとの入出力が分離でき，Lisp システムとユーザとの対話のチャンネルが広がることになる．

3.2 デバッガ

デバッガは実行時にコントロール・スタックに作成されるフレームを調べるツールである．後述するイン

表-2 デバッガの主なコマンド

BACKTRACE	関数がどう呼ばれているかを表示
BOTTOM	最初のフレームへ移動
CONTINUE	デバッガから抜ける
DOWN	一つ前のフレームへ移動
ERROR	現在のエラー・メッセージを表示
EVALUATE〈式〉	現在のフレームで式を評価
GOTO〈フレーム番号〉	指定されたフレームへ移動
HELP, ?	コマンドの要約を表示
QUIT	実行を破棄
REDO	現在のフレームで再実行
RETURN〈値〉...	指定された値を返し，現在のフレームをたたむ
SEARCH〈関数〉	関数を含むフレームを探す
SET	現在のフレームの内容を変更
SHOW	現在のフレームの内容を表示
STEP	ステッパに入る
TOP	一番上のフレームへ戻る
UP	フレームを一つ上がる
WHERE	現在のフレームを表示

スペクタはデータを詳しく調べるツールであり，デバッガは制御情報に対するインスペクタと考えられる．デバッガはブレイク・パッケージやステッパといったデバッギング・ツールやエラー処理ルーティンから起動される．デバッガの機能はほとんどどの Lisp でも似たりよったりである．一例として，表-2 に VAX Lisp のコマンドを示す．Zetalisp 上ではウィンドウを用いてフレームの情報やデータを見ることができる．ウィンドウを用いることによって得られる最大の利点は同時に表示される情報量が単純な画面端末と比べて格段に多いことである．

Lisp/VM ではコンパイル・コードに対してもフレームが作られ変数名が保存されるので，デバッガが

表-3 主なトレース条件

(trace〈トレース条件〉...) で与えられる．

〈関数〉	無条件に関数をトレース
(〈関数〉 break)	ブレイク・パッケージに入る
(〈関数〉 if〈条件〉)	条件が non-NIL ならトレース
(〈関数〉 ifnot〈条件〉)	条件が NIL ならトレース
(〈関数〉 evalin〈式〉)	関数に入った直後に式を評価
(〈関数〉 evalout〈式〉)	関数から抜けるときに式を評価
(〈関数〉 evalinout〈式〉)	evalin と evalout の両方を行う
(〈関数〉 lprint)	トレースの出力をレベルプリントする．関数に入ったときに呼ばれる
(〈関数〉 traceenter〈トレース関数〉)	トレース関数の指定
(〈関数〉 traceout〈トレース関数〉)	関数から出るときに呼ばれるトレース関数の指定
(〈関数〉 printargs〈プリント関数〉)	引数をプリントする関数の指定
(〈関数〉 printres〈プリント関数〉)	結果をプリントする関数の指定

```
-> (defun fact (n)
     (cond ((zerop n))
           (t (times n
                (fact (subl n))
           ))))
fact
-> (trace fact)
(fact)
-> (fact 4)
1 〈Enter〉 fact (4)
| 2 〈Enter〉 fact (3)
|   3 〈Enter〉 fact (2)
|   | 4 〈Enter〉 fact (1)
|   |   5 〈Enter〉 fact (0)
|   |   5 〈EXIT〉 fact 1
|   | 4 〈EXIT〉 fact 1
|   3 〈EXIT〉 fact 2
| 2 〈EXIT〉 fact 6
1 〈EXIT〉 fact 24
24
-> (untrace fact)
(fact)
```

図-1 トレーサの出力例

インタプリタと同じように有効である．

3.3 トレーサ

トレースは指定された関数が呼ばれたときにその引数を表示し，関数の実行が終了したときにその値を表示する機能である．トレース情報を得るためには，

(trace 関数名...)

を，トレース機能を中止するときには

(untrace 関数名...)

を行う．トレーサはたいていの Lisp システムで提供されており，トレース条件を指定できるものもある．Franz Lisp で提供されている Joseph Lister というトレーサのトレース条件を 表-3 に示す．また，階乗のプログラムを実行したときのトレーサの出力を図-1 に示す．

3.4 ステッパ

S式を評価するときに1ステップずつ実行できるステッパは，バグを局所化するときに極めて有効なツールである[10]．トレーサでは非常に多くの情報が出力されるので有用な情報がその中に埋もれてしまう可能性が大きい．一方，ステッパでは各式を評価する前にどう実行するかをユーザに問合わせてくるので，バグに関係のない個所はステップ実行ではなく通常の実行をし，怪しい個所をステップ実行し詳細な情報を得ることができる．MacLisp, Franz Lisp, Zetalisp, VAX Lisp, NIL, PSL 等で提供されているステッパに共通する基本コマンドを 表-4 に示す．

表-4 ステッパの主なコマンド

コマンド	MacLisp	VAX Lisp	Franz Lisp
一回ステップ実行	〈sp〉	〈cr〉	〈cr〉
i回ステップ実行	—	—	$n\langle i\rangle$
現在のS式を通常に実行	〈rubout〉	O(VER)	c
ステップ実行せずに残りの実行をする	〈cr〉	F(INISH)	g
ブレイク・パッケージに入る	m	すでに入っている	b
デバッグに入る	—	D(EBUG)	d
ステッパから出る	—	Q(UIT)	q

ステッパと画面エディタを組み合わせるとプログラムの動きを表示することができる．Lisp/VM で提供されている HEVAL は画面エディタ LISPEDIT のもとで走るステッパである[1]．HEVAL では次に評価すべきS式を EP (Evaluation Pointer) が指しており，LISPEDIT のフォーカスと同様に高輝度で表示される．このS式をステップ実行あるいは通常の実行を行うことができる他に，この時点でフォーカス(カーソル) を移動させることによりどこまでステップ実行をせずに一気に実行するかを指定できる．HEVAL の主なコマンドを 表-5 に示す．ステッパで実行している関数がトレーサの対象となっているとその部分もステップ実行されるが (他の Lisp も同様)，FAST というオプションを指定するとトレーサは無視される．HEVAL での表示には LISPEDIT と同様にズーム機能が働く．また，表示されている関数を修正したり，変数束縛といった環境を変更すると，新たな関数定義と環境の下で実行が再開される．

ステッパでは評価する前にどう実行するかの判断を行わなければならない．ここはバグに無関係であるからステップ実行しないと判断し，通常の実行を行ったとしよう．もしバグが生じなければこの判断は正しかった．しかし，判断が間違っていてバグが生じた場合にはバグの生じる時点を通り過してしまったのでもう一度初めからやり直さなければならない．また，安全をとってすべてステップ実行するとバグに出会うまでに遠々と時間を要することになる．HEVAL のようにどこまで一気に実行するかが指定できればこの操作を端折ることができ，注目する点へすぐに移れるので，便利である．しかし，HEVAL でも結果を見てから前に戻ることはできない．

表-5 HEVAL の主なコマンド

STEP	一回ステップ実行
RUN	EP を通常に実行
FIN	EP から実行をはじめフォーカスの実行を終了して止まる
COME	FINと同じ．ただし，フォーカスの直前で実行を終了
VALUE 〈式〉	現在の環境で式を評価
CONT 〈引数〉	引数を実行しその値を EP の指すS式の値とする
CONT	EP をフォーカスへ移し実行
CHECK 〈引数〉	実行を行うごとに引数を評価し，その値を表示する．

ステッパと Zmacs を組み合わせた Zstep[10]では時間を逆上りながらステップ実行する機能が備っているので，結果を見てから判断を下すことができる．この機能は，ステップ実行を式をその値で置換する代入機構とみなし，代入過程の履歴を保持することによって実現されている．したがって，実行は前方向にも後ろ方向にも進むことができる．前方向に進むとはS式の値を計算してS式にその値を代入することであり，後ろに戻るときには値が元のS式で置換される．このように Zstep では前方向にも後ろ方向にも実行が行えるので，結果を見て判断を下せ，他のステッパのような判断を誤ったときの損失は少ない．ただ，現在の Zstep は副作用の対処は行っていない．また，基本コマンドしか備っていないので，HEVAL のようなユーザが指定した個所まで一気に実行するコマンドも提供されていない．

3.5 クロス・レファレンス用ユーティリティ

MASTERSCOPE[23]は Interlisp で提供されているクロス・レファレンス用ユーティリティである．解析されたクロス・レファレンスはデータベースに格納されており，ユーザは対話的にそのデータベースを参照することができる．たとえば，

WHO CALLS CLEAR-TOP

WHO USES ＊NUMBER-OF-OBJECTS＊

は関数や変数の参照関係を問合わせるコマンドである．後者のコマンドに相当する機能は Lisp/VM の WHOSEES という関数で実現されている．また，MASTERSCOPE で SHOW PATHS FROM MAIN TO CLEAR-TOP というコマンドは関数 CLEAR-TOP が関数 MAIN からどういう道筋で呼ばれるかを教えてくれる．さらに，MASTERSCOPE とエディタを組み合わせると，ある関数を変更したときにその関数を使っているすべての関数を修正することができる．たとえば，

EDIT WHERE ANY CALLS MODIFIED-FN

によって修正すべき関数をすべてエディットでき，修正忘れを防止することができる．

Interlisp-D では SHOW PATH によってビット

マップ・ディスプレイ上に関数の呼応関係が表示できる．この呼応関係の木の節には関数名が表示されており，マウスで節を指すことにより関数定義をプリティ・プリントし，その定義をエディットすることができる．ただ，あまりに大きなプログラムに対してその呼応関係を表示させると一画面に入りきらないので，縦方向横方向にスクロールして見なければならない．

3.6 インスペクタ

デバッガがコントロール・スタック上に置かれている制御情報を見るのに用いられるのに対して，インスペクタはデータを調べるのに用いられる．データにはデータ型の他に関数定義，属性リスト，さらにはコンパイル・コードも含まれる．Interlisp-D や Zetalisp ではウィンドウを活用したインスペクタが提供されている．

Zetalisp ではオブジェクト指向プログラミング・システム FLAVOR のためのインスペクタが提供されている．これはウィンドウ・システムを用いたインスペクタの一種であり，FLAVOR の階層関係を表示し，操作するシステムである．

4. その他のユーティリティ

Interlisp では Lisp リスナの履歴を保持しており，履歴を見たり，以前の入力を RETRY によって再実行したり，以前に行った副作用を UNDO によって取消したりする機能が提供されている．この機能は Programmer's Assistant (PA) と呼ばれており，同様の機能が他の Lisp でも提供されている．

Lisp リスナとエディタを組み合わせると，PA よりもはるかに豊富な機能を実現することができる．たとえば，Emacs と Franz Lisp の組み合わせ，Zmacs と Lisp マシン LAMBDA 上の Zetalisp の組み合わせ (Ztop と呼ぶ)，Interlisp-D などでは，Lisp システムとの対話の全過程がエディットでき，かつ再利用可能である．

Interlisp はプログラミング環境の先駆的システムであり，以上の他にさまざまなユーティリティを提供している[23]．たとえば，オンラインでマニュアルが参照できる HELPSYS，入力誤りを自動的に修正する DWIM (Do What I Mean,『どういう意味』とでも訳せよう)，ファイル管理を行うファイル・パッケージなどがある．アトムとして使用できる文字列の長さはどの Lisp でも少なくとも 32 文字はあるので長い名前を使うことが可能である．しかし，多くのユーザは入力誤りを恐れて符牒のような短い文字列の名前を使用している．DWIM があればそのような心理的圧迫から逃がれて長い名前が使えるので，プログラムの見やすさが向上する．

DWIM は単なるスペル自動修正にとどまらないで人間との対話をより快適にするために使われる．たとえば，Interlisp-D ではメニュ選択のときに，一つのメニュ要素に対してマウスのボタンを押し続けていると*，その要素の内部を表示し，ユーザが選択に迷ったときの手助けをしてくれる．

ファイル・パッケージは関数とファイルとの対応関係を管理する機能である．コンパイルされた関数の定義をエディタに与えるのはこの機能による．また，エディタで変更した関数をファイルに書き出すのにもこの機能を使うとファイルへの出力忘れを防ぐことができる．同様の機能は他の Lisp でも実現されている．

Common Lisp[20] では与えられた文字列を含む名前をリストで返す APROPOS-LIST，その名前と性質(関数か大域変数か)を表示する APROPOS，引数の性質を記述する DESCRIBE の仕様を定めている．これらの機能は Common Lisp の母体となった Lisp で提供されている．

最後に忘れてはならない重要な機能としてネットワークがある．Lisp が人工知能研究で主たる言語として使用されてきた理由の一つに，Lisp が自然言語と同様に常に発展し続けてきた言語であるという点を挙げることができる**．言語が常に成長し続けるためには言語設計者，処理作成者に対してユーザからのフィードバックが生かされなければならない．つまり，言語設計者，処理系作成者およびユーザから構成される Lisp コミュニティの存在が不可欠である***．このコミュニティ作りに多大な貢献を行ってきたのがネットワークである[24]．

ネットワークはローカル・エリア・ネットワーク (LAN) と広域ネットワークに分けることができる．Lisp コミュニティで使用されている LAN として，スタンフォード大学と Xerox PARC で開発された PUP ネット，MIT で開発された CHAOS ネットなどがあり，各サイトでさまざまなネットワークが使用されている．各サイトの LAN は独立しているのではなく，ARPA ネットという広域ネットワークと相互接

* ボタンを押してすぐ離すとその要素が選択される．

** Common Lisp も決定版ではなく，オブジェクト指向，ウィンドウ・システム等を取り込む改訂作業が進行中である．

*** MacLisp に対して成熟したコミュニティがあったので，完全なマニュアル[15]は 1983 年までなかった．

続されているので，居ながらにして全国の研究者と電子メールやファイルのやりとりが行える．さらには，電子掲示板も提供されているので，ネットワークを介して討論を行うこともできる．最近では，ARPA ネットの他に NSF が援助する CSNET や Unix ユーザをバケツリレー方式で接続する USENET も研究者用広域ネットワークとして普及している．

ワークステーションとして提供されている Interlisp-D や Zetalisp では LAN が Lisp システムから直接使用することができる．もちろん，メール関係のユーティリティはウィンドウ・システム，エディタと組み合わされて提供されている．汎用機上では OS レベルでネットワークをサポートするのが通常であるが，NIL では Lisp システムの中から直接ネットワークへのアクセスが可能である．

5. おわりに

Lisp のLisp たるゆえんはプログラミング環境にある．プログラミング環境が充実していなくては Lisp とは言えない．本稿では，過去 15 年以上の間にどのような技術の蓄積が行われてきたかを簡単に概観した．これらの機能は言葉では言い尽せない．百聞は一見に如かずということわざもある通り，是非いずれかのシステムに触れて，プログラミング環境の良さを体験されるようにお勧めする．

15 年以上の蓄積と言ったが，現在提供されているシステムが十分な機能を持っているとはまだまだ考えられない．Lisp マシンの登場によって，ビットマップ・ディスプレイとマウスを活用したウィンドウ・システムやメニュ方式などの技術が開発されるようになった．Lisp リスナ，ブレイク・パッケージ，デバッガ，プリティ・プリントの出力などが別々のウィンドウに割り当てられると，計算機との対話の文脈がいくつも持てることになり，対話のチャンネルが拡大する．また，マウスによるメニュ方式は直接操作方式やモードレス・モードという対話技法の開発を促した．しかし，人間同士が行う対話のエッセンスが計算機上に十分に実現されたわけではない．たとえば，計算機からの出力が計算機には見えないという対話の双方向性の欠如や，計算機側から対話の主導権がとれないという対話主導権の片務性など不十分な点は多い．また，対話のインタフェースをユーザのレベルに応じてカスタム化したり拡張できる機能も必要である．

人間同士の対話のエッセンスを実現するためには，対話型プログラミング環境では不十分であり，知的プログラミング環境へと発展していかなければならない．現在人工知能研究の一分野として研究が進められている自動プログラミングは知的プログラミング環境に大きな影響を及ぼすと考えられる．たとえば，Programmer's Apprentice（プログラマの弟子）[2]は，プログラマの意図を知識ベースに維持するによって，プログラマの操作に助言を与えたり，あるいはプログラミングのこまごまとした細部を引き受けてくれる．つまり，プログラマがより難しい本質的な問題に取り組めるように，Apprentice が補助を行うのである．このプロジェクトは壮大な計画でありプロトタイプが完成しただけであるが，知的プログラミング環境に一つの方向として重要な示唆を与えてくれよう．このような研究の中から知的増幅器としてのプログラミング環境が明確になると期待される．

本稿が Lisp コミュニティの成長，知的プログラミング環境の研究の進展，および，研究用広域ネットワーク構築の一助となれば幸いである．

参考文献

1) Alberga, C. et al.: A Program Development Tool, IBM Journal of Research and Development, Vol. 28, No. 1 (Jan. 1984).
2) Baar, A. and Feigenbaum, E. (eds.): Handbook of Artificial Intelligence, Vol. 2, William Kaufmann, Inc. (1982). (田中・淵監訳：人工知能ハンドブック第２巻，共立出版，1983).
3) Barstow, D. R.: A Display-Oriented Editor for INTERLISP, in 4).
4) Barstow, D. R., Shrobe, H. E. and Sandwall, E. (Eds.): Interactive Programming Environments, McGraw-Hill (1984).
5) Burke, G. S. et al.: NIL Notes for Release 0.259, MIT (1983).
6) Chikayama, T.: UTILISP MANUAL, METR 81-6, Mathematical Engineering Section, Univ. of Tokyo (Sep. 1981).
7) Foderaro, J. K. and Sklower, K. L.: The Franz LISP Manual, UCB (1981).
8) Interlisp Reference Manual, Xerox Corporation (Oct. 1983).
9) 情報処理，特集エディタ，Vol. 25, No. 8 (Aug. 1984).
10) Lieberman, H.: Steps Toward Better Debugging Tools For Lisp, Conf. Rec. of 1984 ACM Symp. on Lisp and Functional Programming, ACM (Aug. 1984).

11) 丸山 宏：LISP/VM のプログラム開発環境，情報処理学会記号処理研究会資料，31-10 (Mar. 1985).
12) Mikelsons, M.: Interactive Program Execution in LISPEDIT, Proc. of ACM SIGPLAN/SIGSOFT Symp. on High Level Debugging, ACM (Mar. 1982).
13) Moon, D., Stallman, R. M. and Weinreb, D.: Lisp Machine Manual, Fifth Edition, LMI (Jan. 1983).
14) 奥乃 博：知識工学とプログラミング環境，計測と制御，Vol. 22, No. 9 (Sep. 1983).
15) Pitman, K. M.: The Revised MacLisp Manual, MIT/LCR/TR 295, MIT LCS (May 1983).
16) Reynolds, C. W.: Computer Animation with Scripts and Actors, Computer Graphics, Vol. 13, No. 3 (1982).
17) Sandwall, E.: Programming in an Interactive Environ ment: The "LISP" Experience, ACM Computing Surveys, Vol. 10, No. 1 (1979). (黒川利明訳：対話環境でのプログラミング-Lispでの経験，**bit** 別冊コンピュータ・サイエンス，1979).
18) Shortliffe, E. H.: Computer-Based Medical Consultations: MYCIN, Elsevier (1976). (神沼，倉科訳：診療コンピュータシステム，文光堂，1981).
19) Stallman, R. M.: EMACS Manual for TWENEX Users, AI Memo 555, MIT (Oct. 1981).
20) Steele, G. L. Jr.: Common LISP: The Language, Digital Press (1984).
21) Stenning, V. et al.: The ADA Environment: A Perspective, IEEE Computer, Vol. 14, No. 6 (June 1981).
22) Teitelman, W. et al.: The Interlisp Reference Manual, Xerox PARC (1978).
23) Teitelman, W. and Masinter, L.: The Interlisp Programming Environment, IEEE Computer, Vo. 14, No. 4 (Apr. 1981).
24) 徳田雄洋，徳田英幸：広がる電子コミュニティ 1, 2, 日経エレクトロニクス，1984年10月8日号，10月22日号.
25) The Utah Symbolic Computation Group: The Portable Standard LISP User Manual, The Utah Symbolic Computation Group Technical Report TR-10, University of Utah (Jan. 1981).
26) VAX LISP User's Guide, Digital Equipment Corporation (June 1984).
27) Wood, S.: Z-95% Program Editor, Proc. of ACM SIGPLAN/SIGOA Symp. on Text Manipulation, SIGPLAN NOTICES, Vol. 16 No. 6 (Jun. 1981).

(昭和60年4月10日受付)