Vol. 30 No. 4 情 報 処 理 Apr. 1989

解 説

# 1. プログラミング言語と環境

## 1.1 Lisp のプログラミング環境†

上 田 良 寛††

### 1. は じ め に

Lisp は，世の中に現れた当初から，言語とプログラミング環境が密接した形で発展してきた．この意味では，Lisp のプログラミング環境は決して新しいものではなく，「新しいプログラミング環境」という今回の特集にはそぐわないように思える。しかし，この環境から生まれた多くのアイデアが，他の新しいプログラミング環境で生かされており，今後も十分に手本となりうると考えられるので，ここで「新しいプログラミング環境の先頭バッター」としての Lisp のプログラミング環境をまとめることは，意味のあることかもしれない．

上谷[1987] や鈴木[1988a] のように，「プログラミング環境」という言葉に，プログラム開発ツールだけでなく，言語処理系の機能（抽象化機構，強力なコンパイル時チェックなど），実行系の機能（記憶領域管理，ライブラリ群など）とを含める立場がある．Lisp の環境の場合もそのほうが適切ではあるが，ここでは他の解説と歩調を合わせ，開発ツールに絞りたい．

この解説でとりあげるのは，Xerox Lisp と Symbolics Lisp という二つの代表的な Lisp のプログラミング環境である．これらは，それぞれ Interlisp (Interlisp-D) と Maclisp/Zetalisp という Lisp の2大潮流の直接の後継者である．現在は両者ともに Common Lisp の仕様[Steele 1984] を満たしていて言語としては差はないが，環境は大幅に異なっている．ここでは，二つの環境を比較しながら，Lisp のプログラミング環境に必要な機能を考察する．さらに，両者ともワークステーションに搭載されており，ワークステーションの機能を生かした環境の強化がなされているので，この点についても言及する．

なお，Xerox Lisp および Symbolics Lisp という名前は，それぞれの Lisp の現在のバージョンを指している場合があるので，歴史的なバックグラウンドをもつツールの説明の中では Interlisp および Maclisp という名前を適時用いることにする．

### 2. Lisp のプログラミング環境の特徴

#### 2.1 統合環境の必要性

通常の言語におけるプログラミングでは，「概要設計→詳細設計→コーディング→テスト・デバッグ」というように，しっかりした開発サイクルに沿って行うことが重要視されている．これに従わねば，効率的な開発はおろか，正しいプログラムの開発もおぼつかないといわれている．

しかし，AI などの分野においては，問題がプログラム開発の前に明らかになっていない場合も多い．このような場合には，プログラムを開発しながら問題を探究していくことになる．このようなプログラミングの進め方を「探究的プログラミング（exploratory programming)」と呼ぶ[Seil 1983]．また，AI 以外の分野においても，最初は必要な機能が明確にできず，使いこみながら仕様を固めていかねばならないものもある．このような開発方法は，ソフトウェアプロトタイピングと呼ばれている．

「正しい開発サイクル」で開発している人からは，とんでもないプログラミングスタイルにみえるだろうが，このような進め方が有効な分野もあるということである．

このようなプログラミングスタイルで重要なことは，まず，作成したプログラムが小さな単位ですぐに実行できることである．プログラミングツールがすぐに使えること，別のツールへの移動が容易にできること，ツールが密接に協調しあっている統合環境であることも重要である．

統合環境において統合されているものは，プログラ

† The Lisp Programming Environment by Yoshihiro UEDA (ATR Interpreting Telephony Research Laboratories).
†† ATR 自動翻訳電話研究所

ミングツールだけとは限らない．文書エディタとの統合により，システムの画面を文書に取り込むことができる．これは，たとえば，マニュアルや論文などの文書化に有効である．また，メイルシステムとの統合により，バグレポートを送ることが容易になる．

### 2.2 環境の発展の経緯

Lisp は探究的プログラミングに向いていたので，最初から開発環境とともに発展してきた．

TSS の出現は，他の言語に比べて Lisp にもっとも大きな影響を与えた．Lisp はインタプリタから発達した言語なので，作成したプログラムをすぐに実行することができたからである．もちろん Lisp にもコンパイラはあるが，コンパイルも一つの関数単位ですぐにできるし，リンク時間が必要でないため，ほとんど影響を及ぼさない．

Lisp の基本的な内部データ構造であるリストおよびアトムは，容易に入出力を行えるため，すぐにテストデータを作ることができる．通常のプログラミング言語では，テストデータを作成するためにはテストデータの入出力ルーチンから作らねばならない．些細なことかもしれないが，これも Lisp の関数が作成と同時に実行できることの助けになっている．

Lisp は，プログラムとデータが同じ構造をしていたので，プログラムをデータとして扱うことが容易にできた．このため，プログラムをS式の形式で編集する構造エディタや，プリティプリンタ，プログラム解析ツールなどの作成を促した．また，Interlisp においては，対話的なプログラム開発・実行を支援する，DWIM (Do-What-I-Mean エラー修正機構) やプログラマーズアシスタント（ヒストリ機能，後述）も加えられるようになった[Teitelman and Masinter 1981]．

さらに，Lisp は統合環境の構築にも向いていた．Lisp のツールは実際は Lisp 関数でできており，引数やリターンバリュなどのインタフェースが分かっていれば，どこから（どのツールから）でも自由に呼び出すことができるし，複数のツールを自由に組み合わせることができる．このようなツールの組合せを実現したものとして Unix☆ があるが，Unix での組合せは，ある程度のまとまったプログラムを，シェル内で標準入出力ファイルによるパイプでつなぎ合わせることである．また，プログラム間で渡されるデータは基本的には文字列で，Lisp におけるような結合の自由度は少ない[鈴木1986 b]（もちろん，だから，簡単に扱えるということはいえる）．

☆ Unix は米国 AT&T ベル研究所で開発されたオペレーティングシステムの名前です．

### 2.3 ワークステーション上の環境

Lisp がワークステーションに搭載されるようになると環境もそれに応じて進化してきた．環境に大きな影響を与えるワークステーションの特性としては次のようなものがある．

1）グラフィックス：ワークステーションで用いられるビットマップディスプレイは，環境が提示する情報の表現力を豊かにした．たとえば，それまで木構造といえば頭のなかで作られる概念にしかすぎなかったものが，実際に目で見れるようになったのである．

2）マウス：マウスの使用により，オブジェクトの選択が（変数に代入しておくことなしに）直接的に行えるようになった．Symbolics Lisp では，対話の履歴がマウスで扱え，コマンドを再実行したり，リターンバリュを選択したりできる（**4.1** 参照）．

3）メニュー：さらにこれをコマンドに応用したものがメニューである．メニューを用いることにより，「コマンドを選ぶ」ことが「コマンドを覚える」ことにとって代わるようになった．

4）マルチウィンドウ：統合されたツールへの移動を容易にしたばかりでなく，移動する前に使っていたツールへの復帰も，以前に使っていたそのままの状態に復帰できるようになった．あるツールを使っている間も他のツールは画面上にあるため，状態（ツール）の遷移の経緯を覚えておく必要はなくなったのである．

5）マルチプロセス：マルチプロセス自体はワークステーションに限った機能ではないが，マルチウィンドウでその威力を発揮する．プロセス（ツール）ごとに入出力のチャネルが与えられることになるので，自由にメッセージを出し，パラメータの入力をユーザに要求しても，混乱を招くことがない．

## 3. プログラミング支援ツール群

### 3.1 エディタとファイルシステム

（1） 常駐システムとソースファイルシステム

表題にエディタとファイルシステムが並べて書かれていることは，奇異に感じられるかもしれない．エディタとファイルシステムとはかなり独立したもののはずである．ここで二つを並べたのは Interlisp 側の事情による．

通常のプログラミングシステムでは，プログラムは

エディタを用いてファイル上に作成され，コンパイラがそのファイルを読み，コンパイルした結果もファイルに書きだす．Maclisp は，多少事情は異なるものの基本的にはこのような種類に属する．このようなシステムはソースファイルシステムと呼ばれている[Sandewall 1978]．

一方，先に述べたとおり，プログラムはデータと同じ内部構造であるS式で表現される．このため，S式を操作する関数（CAR, CDR, CONS, RPLACA, RPLACD など）を用いれば，プログラムも編集できることになる．このような考え方で，直接内部構造を編集するのが Interlisp の**構造エディタ**である．Interlisp では，実行・テストだけでなく，このようにプログラムの編集も Lisp 環境で行われる．プログラム開発に関連する作業をすべて Lisp 環境の中で行うようになっているので，Interlisp は**常駐システム**と呼ばれる．

内部構造でプログラムの編集ができても，これらをファイルへ（テキスト形式で）書き出したり，ファイルから読み込んだりする機能は不可欠である．ファイルは，バックアップの意味もあるし，他の機械へ移植するのにも必要だからである．Interlisp では，関数や変数・定数と，それらを格納するファイルの関係を管理しており，ある関数が変更されたらファイルも再度書き込みを行う必要があることを通知する機能などがある．これは**ファイルパッケージ**と呼ばれている．

（2） Interlisp の構造エディタ

Interlisp 側の事情を簡単に説明したところで，構造エディタの説明にもどろう．この構造エディタによる編集中は，全体の構造の中の一部のみに視点（attention）が向けられており，以下のようなコマンドを用いてプログラム／データの編集を行う．

視点を相対的に移動する
視点の前または後に要素を挿入する
視点にある要素を削除する
括弧を挿入する，削除する
ある要素を探索してそこに視点を移す
視点要素の印刷，プリティプリント
先に実行したコマンドの取り消し

コマンドを実行しても印刷コマンドを実行しない限り結果は表示されない．慣れるまでは構造を変えたり視点を移動させたりする度に印刷コマンドを実行することになるが，慣れてくると複数のコマンドを一気に実行しても平気になる．

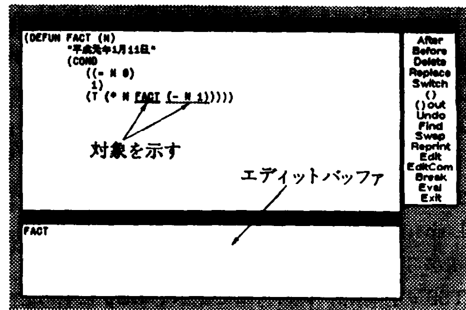



図-1 Xerox Lisp の DEdit.
FACT とタイプして（− N 1）の前においたところ，現在 FACT と（− N 1）が選択されている．ここで"( )"コマンドによりこの二つを括弧でくくることができる．

この構造エディタは，Inetrlisp-D ではワークステーションの機能を利用してディスプレイオリエンテッドエディタ DEdit に発展した（それまでのエディタはテレタイプエディタと呼ばれるようになった）．DEdit では，（1）操作の対象を選んで，（2）操作を行うというオブジェクト指向的インタフェースで統一されている．図-1 で，下線（実線および破線）の付けられている部分が操作の対象である．ここで，"( )" メニューコマンドを実行してこの二つを括弧でくくる（実行のたびにプリティプリントが行われる点はテレタイプエディタと異なる）．

DEdit では，関数定義一つに対して一つのウィンドウが割り当てられるマルチウィンドウエディタになっている．ある関数内で用いられている関数の編集に移ることが容易にできる（Edit コマンド）し，もとの関数の編集にもどることも容易である．ウィンドウ間でコードの一部をコピーしたり移動したりできるので，大きくなりすぎた関数を分割，再構成することも簡単にできる．

Interlisp の構造エディタの問題点として，1文字単位の編集の問題と，コメントの保持の問題がある．

構造エディタでは，構造を入れかえることが基本で

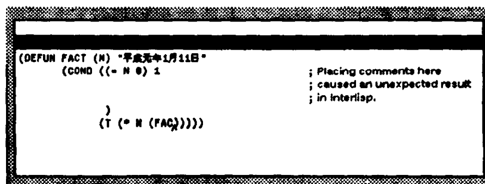



図-2 Xerox Lisp の SEdit. "FACT" の "FAC" までタイプしたところ．括弧の対応がとられている．コメントも Common Lisp の流儀にしたがっている．

あるため，アトムの中を1文字単位で編集することは基本的にはできない．1文字単位での編集のためのコマンドもあるが，アトム程度なら全部打ち直して置き換えるほうが早かった．

最新のバージョンで用いられる SEdit と呼ばれるエディタ（図-2）は，構造エディタにテキストエディタの機能が融合されており，この1文字単位の編集の問題は解決されている．

もう一つの問題は，コメントの保持に関するものである．ソースファイルシステムの場合は，ファイル中のコメントは単に読みとばせばよかったが，常駐システムの場合はコメントを含んだ形で編集しなければならない．Interlisp の場合は，コメントは*という関数で表現されており，エディタで同じように編集する．関数で表現されているということは，値を返すということを意味しているので，コメントをおく場所には注意が必要であった．PROGN の途中に置くのはよいが，その最後に置いたり，関数の引数の並びの途中に置いてはならないのである．

Xerox Lisp の SEdit は Common Lisp の形式でコメントを入力でき，コメントの位置も気にする必要はない．しかし，実際は，今までと同様に内部的にはリストの形態で保持されている．

（3） ファイルパッケージ

ファイルパッケージは，関数定義，変数・定数の値とファイルの対応をとるものであるということを述べた．この関係を保持するものは，“ファイル名+COMS”という名前の変数である（これを COMS 変数と呼ぶ場合がある）．たとえば，FOO というファイルに関数 MAIN-FN，SUB-FN1，SUB-FN2 および，（パラメータ定義した）変数 *CONTROL-FLAG* が保持される場合，変数 FOOCOMS には，次の値が入れられる．

```
((FUNCTIONS MAIN-FN SUB-FN1 SUB-
 FN2)
 (VARIABLES *CONTROL-FLAG*))
```

ここで関数 MAKEFILE を実行する．

```
(MAKEFILE 'FOO)
```

このとき，ファイル FOO には，

```
(DEFUN MAIN-FN (...)
    ...)
(DEFUN SUB-FN1 (...)
    ...)
...
(DEFPARAMETER *CONTROL-FLAG* ...)
```

というロード可能な形式で書き出される．変数 FOO-COMS も同時にこのファイルに記録される．

ファイルパッケージで管理されるのは，関数，変数だけではない．同時にロードするファイルやロード時に評価するフォームも COMS 変数に指定できる．

各関数，変数が変更されるとファイルパッケージに通知されるので，ファイルパッケージは，どのファイルが再 MAKEFILE，再コンパイルを行わねばならないかを知っている．ユーザは，関数 FILES? によりその情報を知ることができる．同時に，格納されるファイルが決まっていない新規の関数，変数に関して，一つ一つどのファイルに入れるべきかを聞いてくる．これに答えていくと，COMS 変数にその関数，変数が追加される．

（4） Symbolics Lisp の ZMACS

Symbolics Lisp には，代表的なテキスト型エディタの EMACS にビットマップやマウスの機能などの拡張を施した ZMACS と呼ばれるエディタが用意されている．もちろん，ZMACS 自身も Lisp 言語で記述されているので，コマンドの拡張や新しいメジャモードの定義などのカスタマイズが柔軟にできる．

Symbolics Lisp は ZMACS を中心にして統合化されているということができる．ZMACS は，Lisp 言語のコーディングやデバッグに便利なインクリメンタルコンパイル，括弧の対応の表示，マクロの展開，ソースコードの検索，S式のインデントなどの機能を用意している．

特に，その中で関数や大域シンボルなどの定義の場所を調べる機能が強力である．単純な使用方法としては，エディタの M-.（メタ-ピリオド）コマンドや ED 関数などを用いる．それらを使用すると，ZMACS はディスクファイルやバッファから指定された関数定義を捜し出しエディタのウィンドウに表示してくれる．この機能の仕組みは，ワールドと呼ばれる Lisp 環境中のシンボルのプロパティリストにソースコードの場所が保持されており，ZMACS がその内容を使用しているのである．このようにソースファイルシステムである Symbolics Lisp でも，ファイルと関数の対応の管理は行われており，編集に効果を発揮する．

さらに高度な使用例として，ZMACS とコンパイラの統合を利用することがある．Symbolics Lisp の場合，コンパイル時に発生したエラー，ウォーニングメッセージをデータベース管理する．このコンパイルの単位は，ファイル，バッファ，リージョン，S式な

どがあるが，それぞれのコンパイルで発生したエラー，ウォーニングはソースコードの場所との対でデータベースに格納される．ZMACSの“EDIT COMPILER WARNINGS”コマンドを使用すると，エラー，ウォーニングのメッセージとともに，それに対応するソースコードを検索し表示してくれる．プログラマは，エラー，ウォーニングを発生させた部分を修正し，エラー，ウォーニングがなくなるまで，その部分だけをC-Sh-c（コントロール-シフト-c）コマンドでインクリメンタルコンパイルすればよい．そして，その部分のエラー，ウォーニングがなくなれば，次のエラー，ウォーニングの発生している定義へC-.（コントロール-ピリオド）コマンドで移るという手順をふむ（図-3）．この機能は，大規模なプログラムをコンバージョンするようなバッチ処理的な作業に非常に有効である．

（5） 再び常駐システムとソースファイルシステム

常駐システムとソースファイルシステムの差については，Sandewall[1978]による議論がある．ここではかなり多くの観点から，二つの優劣を検討している．また，ここにはStallmanのEMACS側からの反論とそれに対するSandewallの返答がつけくわえられており，非常に興味深いものになっている．

しかし，ここでの議論は，ワークステーション文化の前のものであって，現在はまた違った状況になっている．たとえば，前述したとおり，関数単位で編集する常駐システムは，マルチウィンドウを用いて複数の関数を参照したり，他の関数からコードの一部を借りてきたりすることができる．ウィンドウを並べかえて関連する関数を近くにもってくることも自由である．しかし，ソースファイルシステムでは，せいぜい画面のスプリット程度しかできない．M-.（メタ-ピリオド）コマンドを用いて別の関数定義に移ることはできるが，その関数の編集を終了したときに，もとあった位置にカーソルを戻すためには，レジスタにポジションを記憶させておく必要がある．

このように，常駐システムはマルチウィンドウと組み合わせると，効果が高くなると考えられる．また，後述するオブジェクト指向プログラミングのサポートも，ソースファイルシステムでは難しいのではなかろうか．このような新たな議論が起こってもよいのではないかと考える．

## 3.2 デバッグ支援

（1） デバッガ

```
Warnings for file LM01:>lispm>flavor-test

  For Function (FLAVOR:METHOD CL-USER::XOR CL-USER::JIGGLER)
    While compiling CL-USER::Z:
      The variable CL-USER::Z is unknown and has been declared SPECIAL

  For Function (FLAVOR:METHOD CL-USER::XOR CL-USER::ZOOMER)
    The variable CL-USER::Z was never used.

*Compiler-Warnings-1*
; 6

(defmethod (xor jiggler) ()
  (graphics:draw-circle x y z :alu :flip :stream *aquarium*)

(defmethod (xor zoomer) (z)
  (graphics:draw-rectangle x y (+ x 6) (+ y 6) :alu :flip :stream *aquarium*))

(defmethod (xor rightfish) ()
  (graphics:draw-line (- x 5)  y (+ x 5)  y  :alu :flip :stream *aquarium*)
  (graphics:draw-line x (- y 5)  x (+ y 5)  :alu :flip :stream *aquarium*))

; 7

(defun draw (fish)
  (xor fish))

(defun undraw (fish)
  (xor fish))

; 8

(defun d+ (m n)
  (mod (+ m n) 500))

flavor-test.lisp >lispm LM01: (1)
Zmacs (LISP) flavor-test.lisp >lispm LM01: (1) *  [More above and below]

[Fri 25 Nov 4:20:14]  Keyboard          CL USER:          User Input
```

このウォーニングはカーソルの位置でXORジェネリック関数定義上で発生していることがわかる．

エラー表示ウィンドウ

バッファ表示ウィンドウ

図-3 Symbolics Lisp の ZMACS
“EDIT COMPILER WARNINGS”コマンドを発行したところ．Control-.（コントロールピリオド）コマンドで次のエラー，ウォーニングの発生している場所へ移ることができる．

プログラムを実行してエラーに遭遇するとデバッガに自動的に入る（Xerox Lisp ではデバッガはブレークパッケージと呼ばれている）．また，ブレークポイントの設定により，デバッガに入ることを指定することができる．

Lisp では，変数の束縛環境は実行時に決定されるので，スタックフレームを調べる機能が重要になる．スタックフレームは，ネストして呼び出された関数の一つ一つに対応していると考えてよい．機能としては，スタックフレームの名前を順にリストさせること（バックトレース），そこで束縛されたパラメータやローカル変数の内容を知ること，あるスタックフレームに環境を移すこと，そこから実行を再開すること，あるスタックフレームの返す値を一時的に与えてそのまま実行を続けることなどが考えられる．Xerox Lisp も Symbolics Lisp もほぼ同様の機能をもち，これらの要求を満たしている．

統合化という観点では，まず，エディタとの統合がある．ある関数に誤りが発見されたら，エディタを呼び出して修正する．修正が済んだら，デバッガに戻る．その関数の実行の直前へスタックフレームを戻す．その関数から実行を再開する．Lisp の場合，関数単位でコンパイルでき，インタプリタコードとコンパイルドコードの混在が可能であることにより，このように，スタックの状態を保存したままプログラムを変更していける．Lisp のデバッガは，その特徴を最大限に利用している．

Xerox Lisp も Symbolics Lisp もウィンドウシステムのユーザインタフェースを生かす拡張がなされている．メニューによる主要なコマンドの実行，マウスによるオブジェクトの選択が可能になっている．たとえば，Xerox Lisp では，バックトレースにはフレーム名（関数名）のみが記入されたメニューになる（図-4）．

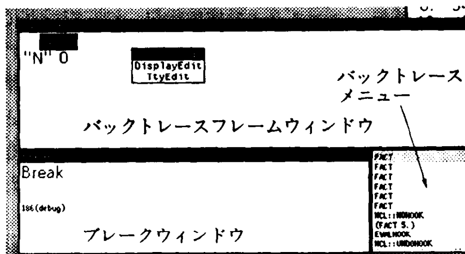



図-4 Xerox Lisp のブレークパッケージ
バックトレースメニューで FACT のフレームを選択した．ここでメニューを出して FACT のコンパイルドコードをみようとしているところ．

フレーム名を選択することにより，フレームを移動し，そのフレームで束縛された変数とその値がインスペクタ（後述）の形式で別のウィンドウに表示する．Symbolics Lisp のウィンドウデバッガも同様の機能をもつ．しかし，このような機能は，Symbolics Lisp では Dynamic Lisp Listener を用いた実装もなされており，ウィンドウデバッガを用いなくとも利用できる．

（2） トレーサ/ステッパ

Common Lisp の仕様では，プログラミング環境に関する指定はほとんどないが，機能の存在を前提にしており，関数（マクロ）名とその引数の形式だけを規定している部分がある．たとえば，break は，ブレークポイントを意味し，ここでデバッガに入ることを規定している．

そのような規定に，トレーサとステッパがある．トレーサは，ある関数に入ったときにその関数名と引数を印字するものである．ステッパは，関数の中の形式一つ一つに対して評価される前に止まり，そのパラメータに与えられた値または形式を表示する．評価後にはその形式の値を表示する．また，ユーザからのコマンドを受け付ける．

（3） インスペクタ

現在の Lisp では，アトム（シンボル）やリストなどの基本的なデータ構造を使うだけでなく，構造体（またはレコード）を用いて種々のデータをまとめたデータ構造を作り，データの抽象化を進めることができる．構造体はいくつかのフィールドから構成されており，そのフィールドの値自体も，ある（同じまたは別の）構造体のインスタンスである場合がある．これで木構造やチェーンなどの大きなデータ構造を組み立てることができる．デバッグ中はこのデータ構造の中を調べる必要が多く発生するが，このように大きくなる構造体の値を見る際には，構造体全体を印刷することは行わず，必要な部分だけを（変数にフィールドの値をセットしながら）みていくことになる．こうしていくと，一時的な変数の数が増えていき，名前を管理するのが不可能になってくる．

これを解決するために，インスペクタと呼ばれるツールが用意されている．インスペクタは，構造体のフィールド名とその値に対して表示する．フィールドの値は，マウスによる選択でさらにインスペクトが可能で，これにより複雑なデータ構造をたどってみることができる．また，インスペクタでは，フィールドに

対して新しい値をセットすることができる．

Xerox Lisp のインスペクタ (図-5) は各構造体に一つのウィンドウを割り当てているのに対して，Symbolics Lisp のインスペクタ (図-6) は，全体が一つのウィンドウで，それがいくつかの領域に分けられており，一度に表示できるデータ構造の数を制限している．Xerox Lisp のような方式では，不要なウィンドウで画面が散らかってくるという問題を引き起こすので，このような制限にも意味がある．

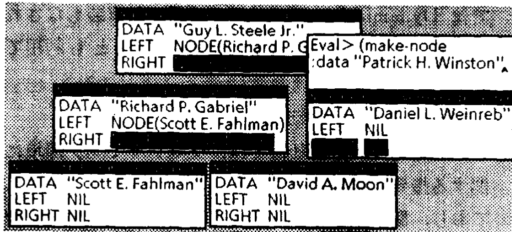



図-5 Xerox Lisp のインスペクタ
"Daniel L. Weinreb" ノードの RIGHT に "Patrick H. Winston" ノードを追加しようとしている．

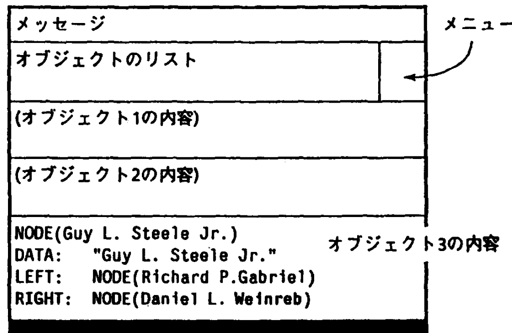



図-6 Symbolics Lisp：インスペクタの画面配置
三つのオブジェクトの内容が表示されている．いずれも最下段のオブジェクト3と同様にフィールドとその値の並びで表現されている．

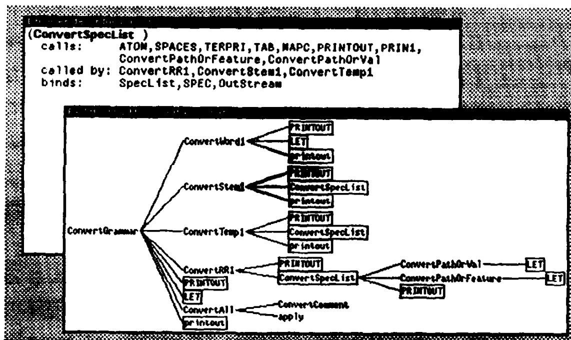



図-7 Xerox Lisp のマスタスコープで SHOW PATHS コマンドを発行したとするとブラウザウィンドウが現れる (前)．後のウィンドウは ConvertSpecList の解析データでブラウザから呼び出されたもの．

(4) プログラム解析ツール

探究的にプログラムを作っていると，最初から詳細な設計をしているわけではないので，プログラムの規模が大きくなるにつれて全体像がつかみにくくなる．プログラムがどのように構成されているか，ある関数が他の関数に対してどのような依存関係 (呼び出し関係，共通の自由変数の使用など) をもっているか，などの静的な関係を調べるのが Xerox Lisp のマスタスコープ (Masterscope) である．英語に近い構文で，マスタスコープに対して質問をすることができる．

SHOW PATHS コマンドにより，呼び出し関係がグラフィカルに木構造表示される (図-7)．ここからエディタを呼び出すこともできる．

(5) パフォーマンス解析ツール

プロセッサの処理速度がいくら速くなっても，実行効率のよいプログラムは常に望まれることである．プログラムを速くするためには，どこがボトルネックになっているのかを知り，そこを重点的に改良することが多い．机上だけでプログラムのボトルネックを判断するのは難しい．人間の頭で，いちいちデータ構造を解釈したり，関数呼び出しをトレースするのには，時間がかかるし，扱うデータ量にも限界がある．このために用意されているのがパフォーマンス解析ツールである．

Symbolics Lisp の METERING は，ソースプログラムに変更を加えずにプログラム実行中の動的なさまざまな情報を得ることができる．関数ごとの処理時間，関数の呼び出し状況，コンスセルの使用状況，ページングに関する情報などを分析し統計をとってくれる．このツールを使用することにより，簡単にプログラムのボトルネックを分析することができ，性能改良の指針を得ることができる．図-8 は FACT 関数の実行を観測した例である．

Xerox Lisp の SPY でとられる統計は処理時間だけで，コンスセルやページングに関するものはない．SPY の特徴は，そのグラフィックインタフェースで，ここからエディ

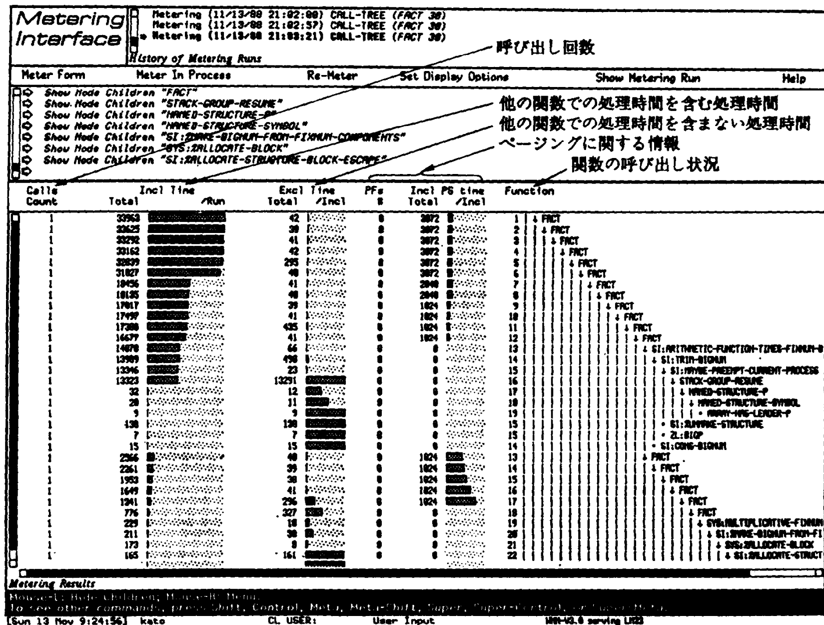



図-8 Symbolics Lisp の METERING
"(FACT 30)" の実行を観測したところ．

タを呼び出したり，木構造のノードをマージして，必要でないデータの表示を抑制することなどができる．

## 4. その他のツール群

### 4.1 履歴管理

プログラムの開発段階においては，サブプログラムを一つ一つ動かして，動作が確認されたらそれをまとめて一つのプログラムにするということがある．このとき，通常は結果を一時的に変数に代入し，次のサブプログラムに渡すが，欲しくない結果が得られるかもしれないのに，変数を用意しておくのはおっくうになる．名前の管理も負担が大きい．できれば，そのような変数なしで結果を取り扱いたい．また，このようなときは，同じサブプログラムをパラメータの一部を変えて何度も実行してみることも多い．

このような対話の進めかたを支援するものとして，対話の履歴の管理がある．

Symbolics Lisp では，この役割は，Lisp のインタプリタ (Read-Eval-Print ループ) である Dynamic Lisp Listener が担っている．Dynamic Lisp Listener のウィンドウは，上下左右にスクロールし，過去に PRINT や FORMAT 関数でプリントされた Lisp オブジェクトや，描画されたグラフィックオブジェクトの履歴がすべて保存される．履歴は単に保存されているだけではなく，表示されているオブジェクトをマウスで選択することが可能である．これにより，コマンドの再実行を行ったり，結果のオブジェクトをインスペクトしたりできる．また，Dynamic Lisp Listener では，エディタと同様に表示されているテキストに対してリージョンを指定することが可能であり，指定したリージョンをエディタのバッファに取り込んだり，直接プリントアウトしたりすることもできる．

Xerox Lisp では，履歴の管理は，プログラマーズアシスタント (PA) と呼ばれている．Xerox Lisp の Read-Eval-Print ループである Lisp Exec は，スクロールウィンドウではなく，また，マウスによる選択が可能になっているわけではない．PA では，対話の履歴は，対話につけられた番号によりアクセスする．コマンドにより，再実行 (REDO コマンド) や TTYIN エディタによって一部修正して再実行すること (FIX コマンド) ができる．実行結果の値は，関数 VALUEOF により参照できる．

PA の特徴は，UNDO コマンドで実行の取消しができることである．たとえば，ある変数に値を代入し

たのを取り消して，その変数のもとの値を復帰させることができる．また，関数定義も取消しが可能なので*，システム関数と同じ名前の関数を定義しても，関数が再定義されたことを示すメッセージが出てから取り消せばよい．このことは，プログラマを精神的な圧迫から開放するのに役立っている．

Xerox Lisp はスクロールウィンドウではないので，少し昔の対話の履歴はウィンドウから消え去ってしまう．このような対話の番号を知るには，？？コマンドを用いる．また，入力されたパターンにより対話を指定することもできる．たとえば，

REDO ED

とすれば，同じ関数を再度編集することができる．

また．ヒストリメニューというツールにより，履歴の操作がメニュー選択でできる．

### 4.2 大規模プログラムの開発サポート

プログラムが大規模になると，ファイルをいくつかに分割したほうが，開発時に頻繁に繰り返すコンパイルやロード/セーブが楽になる．また，意味のある単位で分割することにより，それを別のプログラムで使えるライブラリに発展させることもできる．

しかし，分割すると管理は難しくなる．複数のファイルをロードするだけなら，必要なファイルを一つのファイル中に LOAD 関数の引数として指定しておき，そのファイルを Lisp 環境上にロードすることにより解決できる．しかし，開発中には，現在どのファイルがコンパイルされずに残っているかを把握しなければならない．また，マクロ定義を含むあるファイルをコンパイルした場合に，そのマクロ定義を使っているファイルは再コンパイルしなければならないが，このような依存関係は把握が難しい．

Symbolics Lisp では，これを“システム”という概念でサポートする．いくつかのファイルをグループにまとめて，一つのシステムとして管理する．システムを定義するためには DEFSYSTEM 関数を使って，ソースファイルのディレクトリの指定，コンポーネントモジュールのファイル要素の階層関係，使用パッケージの宣言などを定義する．そうすると，プログラムのロード，コンパイル，エディット，ハードコピー，他システムへのディストリビュート，パッチなどのプログラム管理が簡単にできるようになる．

Xerox Lisp の場合は，ファイルパッケージが，関数間の依存関係を把握するほか，セーブ (MAKEFILE)，コンパイル，ハードコピーされずに残っているファイルを教えるなどのサポートを行っている．しかし，“システム”という概念は希薄である．

### 4.3 オブジェクト指向プログラミングサポート

Xerox Lisp では LOOPS, COMMON LOOPS (Common Lisp Object-Oriented Programming System), Symbolics Lisp では FLAVORS, 新 FLAVORS というオブジェクト指向プログラミングシステムが開発されてきた．これらは，Common Lisp 上のオブジェクト指向プログラミング言語である CLOS (Common Lisp Object System)[Bobrow et al. 1988] に影響を与えた．

Smalltalk-80[Goldberg and Robson 1983] に代表されるオブジェクト指向プログラミング言語は，そのモジュール性，情報隠蔽能力により，生産性を高める上で効果がある．しかし，多重継承とメソッド結合などにより，プログラムの実行時に，どのメソッドが適用されるかをプログラマが判断するのが難しくなるという問題点をもっている．特に，Smalltalk-80 では単一のオブジェクトによってメソッドが決定されていたのに対し，CLOS でのメソッドはそのすべての引数によって決定されるため，問題はさらに複雑になっている．

オブジェクト指向プログラミングによって生産性を向上させるためには，このため，適切なプログラミングツールの存在が不可欠である．Smalltalk-80 の成功も，プログラミング環境抜きには語れない．

CLOS の仕様にはプログラミング環境まで言及されていない．ここでは，Xerox Lisp の LOOPS[富士ゼロックス 1987] における クラスブラウザと Symbolics Lisp の FLAVOR イグザミナ (図-9) を例として用いる．

これらのツールでは，先にあげたようなメソッドの検索のほか，その追加・修正，クラスの追加・修正・階層関係の定義，インスタンスのインスペクトや修正などがメニューによって容易に行えるようになっている．また，クラス階層関係はグラフィックスを用いて表示されるばかりでなく，ここからエディタやインスペクタなどのツールが起動できるように統合化されている．

先に，常駐システムはオブジェクト指向プログラミングの開発環境に適していると述べた．Lisp のみの場合，関数定義は1個しか存在しないので，編集・ロー

* これは Xerox Lisp が常駐システムで，Lisp Exec の中で関数定義も行うことに由来している．ソースファイルシステムである Symbolics Lisp では，ソースファイルのロードの際に再定義の警告が出る．

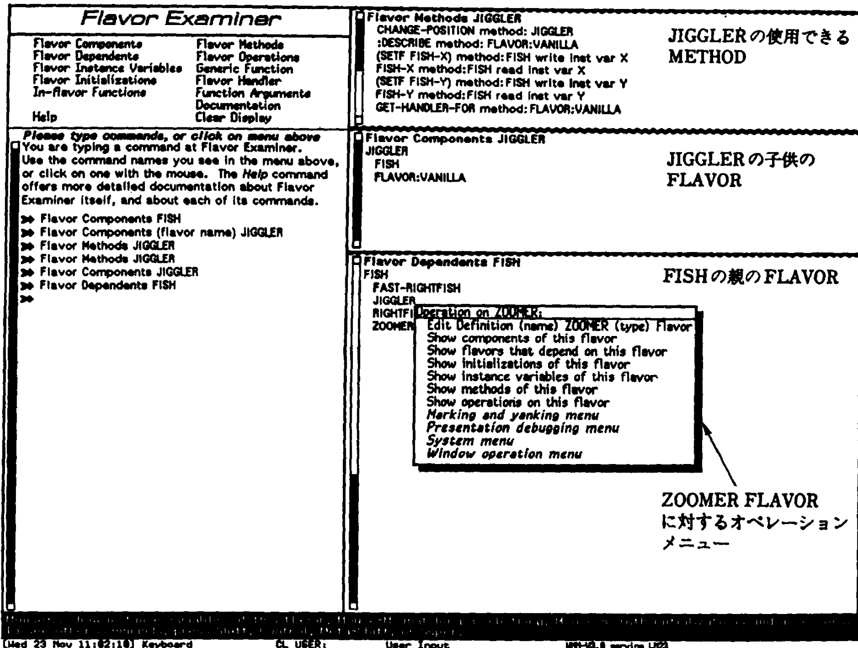
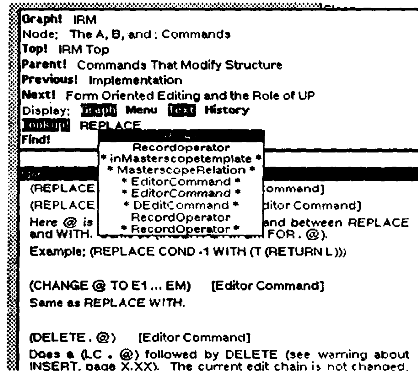



図-9 Symbolics Lisp の FLAVOR EXAMINER
JIGGLER, RIGHTFISH, ZOOMER, の各 FLAVOR はコンポーネント FLAVOR として, FISHを含んでいる.

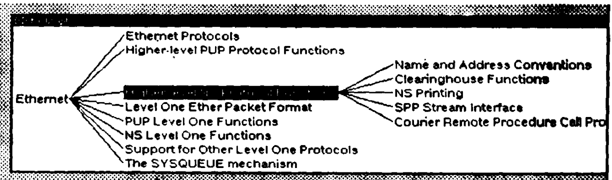
ドを繰り返して新しい定義を増やすか置き換えるかするだけでよく, 定義の削除などは考える必要がなかった. 要らなくなった定義はその関数を呼ばなければ済むのである. オブジェクト指向の場合, メソッドはクラスに対応して存在しており, 同じ名前のメソッドが複数個存在しうる. スーパクラスのメソッドの定義を使うため, あるクラスのメソッドの定義が不要になった場合, ソースファイルシステムで, ソースファイルからその定義を削除するだけでは不十分で, 実行環境で適切に削除しなければならない. このような理由から, 開発時にも実行時と同様の管理が行いやすい常駐システムのほうが, オブジェクト指向プログラミングの開発環境に適していると考えられる.

### 4.4 オンラインマニュアル

マニュアルに馴れていないと, 知りたいことがマニュアルのどこの個所に記述されているかを捜すだけでも時間がかかるし, 重要事項を見落とすこともありうる. プロ



(a) 上部がメニューで下部がドキュメント内容. "REPLACE" に対して Lookup! コマンドを発行したところである. Replace が種々の場所で現れるのでポップアップメニューが現れ, どれを選択するかユーザに問い合わせている.




(b) 章立てを木構造で表したもの. マウスで選択できる.

図-10 Xerox Lisp の DINFO

グラミングの際には，思考の流れを中断させずにマニュアルを参照したい．

このためにオンラインマニュアルが用意されている．オンラインマニュアルは，ハイパテキストの形態になり，製本されたものにはない特徴がある．たとえば，キーワード検索やマウスによる選択などが可能になる．また，マニュアルの例を Lisp Exec にコピーして実行してみることもできる．

Xerox Lisp では DINFO (図-10)，Symbolics Lisp ではドキュメントイグザミナと呼ばれている．キーワード検索では，ドキュメントの候補を選び出してくれるので，その中から必要なものを選択すればよい．表示されたドキュメント中のあるテキスト項目をマウスで選択することができ，その選択したテキスト項目をさらに検索することができる．そのほかに，ドキュメント中のプログラム例の実行，ドキュメントの概要の表示，プリントアウト，個人的なドキュメントの分類などさまざまな機能がもりこまれている．

オンラインドキュメントは，確かに紙でつくられた本にない良さがあるが，現在のビットマップディスプレイの解像度では読みやすさではとうてい紙の本にはかなわない．そのほかにも紙の本には捨てがたい魅力があり，共存する状態は当分は続くだろう．

## 5. 今後のプログラミング環境への期待

以上，現在のプログラミング環境に関して考察をしてきた．これまでに Lisp 環境についてはほぼ十分なものになったと認められたのか，現在のところは環境の大きな変化はなくなったように思える．しかし，Lisp 言語は標準化のなかでも発展を続けているし，計算機アーキテクチャや周辺機器も発展を続けている．プログラミング環境も，それに応じて発展していくものと期待される．ここでは，今後のプログラミング環境に対する筆者の期待をいくつかあげる．

### 5.1 並列処理プログラム開発サポート

計算機の処理能力を高めるアーキテクチャとして，並列処理が有効な方法の一つである．実際，最近では，BBN の BUTTERFLY, INTEL の iPSC, THINKING MACHINES のコネクションマシンなどの商用の並列処理計算機上で並列 Lisp 言語が使用できるようになってきている．並列処理環境でのプログラム開発のデバッグは，Lisp 言語の優れた統合化された環境を用いたとしても，実行時の時間的要因や空間的要因などの扱いにくい複雑な環境に大きく依存するため，非常に困難である．残念ながら，これらの計算機上には，このような問題を扱う効果的な並列 Lisp 言語の開発ツールは，用意されていないようである．

並列プログラム用の開発ツールの例としては，GHC におけるアルゴリズミック・デバッギングがある[竹内 1987]．これは，並列プログラムにおけるデッドロックのような新種のエラーも扱えるようになっており，効果を発揮するものと考えられる．

### 5.2 マルチランゲージ環境

統合環境の項では，プログラミングツールだけでなく，種々のツールの統合の必要性にふれた．しかし，このような環境を成立させるために，OS のみならず，文書エディタもメイルシステムもすべて Lisp で書かれているのである．文書エディタは今後もマルチメディアなどの発展をとげていくことが期待されているが，この発展を Lisp が享受するためには，Lisp でも実装されなければならないのである．その他のアプリケーションにしても同様である．

このため，マルチランゲージ OS ともいえる OS で，どのような言語で実装されたプログラムでも容易に統合できるものが欲しい．この OS は，広い意味でのプログラミング環境を言語によらずサポートする．これには，まず，オブジェクト管理が必要である．ヒープをまとめて割り付けて，まとめて開放するのではなく，ガーベジコレクションによる回収も行う．このためには一つ一つのオブジェクトに対する知識が必要である．

もう一つは，プログラム間の結合のサポートである．現在の Unix のように文字列だけでプログラム間をつなぐのではなく，Lisp オブジェクトを直接やりとりしたい．また，結合のしかたにも Lisp と同様の自由度をもたせたい．さらに，ウィンドウマネージャとの連携により，ウィンドウを通じたオブジェクトのアクセスも，プログラム間の結合と同様にオブジェクトの種類に限定がないことが望まれる．

**謝辞** この解説の作成に当たって，Symbolics Lisp 環境について教えてくださったうえ，プログラミング環境について意見を寄せてくださいました ATR 自動翻訳電話研究所の諸氏に感謝いたします．特に，この解説の中の Symbolics Lisp に関する記述は全面的に加藤進氏の力をお借りしました．感謝いたします．

## 参考文献


[Bobrow et al. 1988] Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G. and Moon, D. A.: Common Lisp Object System Specification, X 3 J 13 Document 88-002 R (1988).

[Goldberg and Robson 1983] Goldberg, A. and Robson, D.: Smalltalk-80: The Language and its Implementation, Addison-Wesley (1983).

[Sandewall 1978] Sandewall, E.: Programming in an Interactive Environment: The LISP Experience, in Interactive Programming Environment, (Barstow, D. R., Shrobe, H. E. and Sandewall, E. eds.) McGraw Hill (1984), 邦訳：岡田正志（監訳），ソフトウェア開発支援環境—Unix から AI アプローチまで，TBS 出版会 (1986).

[Seil 1983] Seil, B. A.: Power Tools for Programmers, DATAMATION (1983. 2), 邦訳：プログラマ期待の強力なツールが登場，日経コンピュータ (1983. 4. 18).

[Steele 1984] Steele, G. L. Jr.: Common Lisp the Language, Digital Press (1984), 邦訳：後藤英一（監訳），井田昌之（訳）：bit 別冊，COMMON LISP，共立出版 (1985).

[Teitelman and Masinter 1981] Teitelman, W. and Masinter, L.: The Interlisp Programming Environment, IEEE Computer, pp. 25-33 (1981).

[上谷 1987] 上谷晃弘（編著）：ワークステーションシリーズ 統合化プログラミング環境—Smalltalk-80 と Interlisp-D，丸善 (1987).

[鈴木 1988 a] 鈴木則久：プログラミング・エンバイロンメント，研究者のテイスト第 9 回，bit, Vol. 20, No. 9 (1988).

[鈴木 1988 b] 鈴木則久：パーソナルコンピュータの将来を予測する，研究者のテイスト第 10 回，bit, Vol. 20, No. 10 (1988).

[竹内 1987] 竹内彰一：GHC のプログラミング環境，淵　一博監修，古川康一・溝口文雄共編，並列論理型言語 GHC とその応用，第 10 章，共立出版 (1987).

[富士ゼロックス 1987] LOOPS 概説書，富士ゼロックス (1987).


（昭和 63 年 12 月 6 日受付）