記 号 処 理 54－4
（1990. 3. 12）

S式のための文字端末用インターフェース

湯浅 敬
松下電器産業情報通信東京研究所
yuasa@trl.mei.co.jp

本論文は，著者がUtiLisp上で開発したS式用インターフェースList Windowについて述べる．当システムは文字端末上でリストを表示，調査するためのものである．リスト内の任意の要素あるいはヒープ内の任意のオブジェクトは，参照関数を繰り返し実行すればアクセス可能である．しかしそのためにユーザは，トップレベルやエラーハンドラにおいて，関数呼び出しを何度も入力しなければならない．List Windowでは画面上に表示されているオブジェクトをユーザが指定したり，参照関数を簡単に実行できるようにすることでこの操作を軽減した．

Character Terminal Interface for S-expression

Kei Yuasa
Tokyo Research Laboratory
Matsushita Electric Industrial Co. Ltd.
3-10-1 Higashimita Tama-ku Kawasaki 214 Japan

This paper presents List Window, a new interface for S-expression which was developed on UtiLisp. This system is for displaying and inspecting a list on a character terminals. A typical behavior of a Lisp programmer in debugging is to inspect alive objects. The operations for inspecting are preformed in toplevel loops or error handlers and they are repetitions of similar calling of primitive functions. The new system provides a screen oriented interface to simplify the operations of specifying objects on a screen and applying referring functions to them.

1 背景

Lispは，プログラミングを全般的に支援する「環境提供型」言語である．Lispはインタープリタとコンパイラの両方を持ち，プログラムの開発はインタープリタ上で行い，完成したプログラムの実行はコンパイルして使用される．

環境提供型言語では，プログラムを入力し，実行，デバッグするというプログラミングの過程を支援する．ユーザはこの間処理系から出る必要はない．Lispは元来トップレベルというread-eval-printループをインターフェースの核として持ち，プログラムの入力と実行を支援している．

プログラムはテキストエディタを用いてファイルに作られて管理される．UNIX上のLispシステムでは，Xウィンドウやvi，emacsなどのツールや環境をオペレーティングシステムが提供しているので，これを応用することが多い．すなわちウィンドウをLisp用とエディタ用に開いたり，Lispの中から外部のエディタを呼び出す機能を付けるなどすれば，Lispを使いながらファイル管理も可能となる．

デバッグはエラーハンドラで行われる．これはトップレベルと同じread-eval-printループで各種関数を呼び出すことで，エラーの原因を調べる環境を提供している．しかしエラーハンドラで呼び出される関数は一部の参照関数に限られており，引数も同じ場合が多い．ユーザにとって同種の関数呼び出しを何度もキーボードから入力するのは苦痛であり，このエラーハンドラのインターフェースを改良すれば，デバッグ全体が楽になるはずである．

Lisp Machine Lispなどの専用機上のシステムではウィンドウシステムを用いてデバッグを支援するが，汎用機上のシステムではエラーハンドラのユーザインターフェースは昔のままである．UNIXが提供するライブラリなどを利用することによってよりよい環境が提供できるはずである．

著者が開発したList Windowは，エラーハンドラにおける繁雑な関数呼び出しをキーボードから入力しないですむように開発されたもので，デバッグを簡略化するためのものである．当システムでは画面上でリストを表示し，任意の要素を探索できる．さらに画面上の要素に対する参照関数の呼び出しを簡略化することでデバッグを支援する．

本論文は，List Windowの実装とその応用について述べる．2章では，エラーハンドラにおけるデバッグ操作の問題点を抽出する．3章では，画面上にリストを表示する上での問題点を述べる．4章では，3章で述べた問題点をList Windowでどう解決したか，5章では，List Windowの応用例を示す．

2 エラーハンドラにおけるデバッグ操作

Lispシステムにおいて，デバッグを支援するのがエラーハンドラである．これはトップレベルと同じread-eval-printループであり，ユーザは任意の関数を実行することができる．しかし，実際にはここで行われる作業は限定されている．この章では，エラーハンドラで行われているデバッグ作業を分析して，これを簡略化するのにどのようなツールが必要になるかを考える．

Lisp処理系の中では，プログラマはヒープ内にLispオブジェクトを作成したり修正することによってプログラムとそのための環境を作っていく．例えば，関数定義はその名前のシンボルをヒープに作り，その関数スロットにラムダ式，あるいはクロージャを入れることである．変数に値を束縛するのは，シンボルの値スロットに値を入れる，もしくはシンボルと値の連想リストを作ることである．

プログラム実行中にエラーが起きた場合，その原因はヒープの中に残されている．それは例えば関数定義に誤りがあるか，変数に正しい値が束縛されていないなどである．ヒープ内のどこにエラーの原因があるかを探す手掛りになるのがスタックである．エラーハンドラはエラーが起こった時点でのスタックや変数の束縛状態をそのまま保存してユーザの入力を受け付ける．

デバッグ操作の一例を示す．エラーが起こると，エラーの種類とその原因となった引数などは表示される．これだけの情報で原因がつかめた場合は，以後のデバッグは不要であり，`toplevel`関数などでトップレベルに戻ればよい．関数を別の引数で呼び出す場合も，スタックを保存する必要はないので，通常トップレベルに戻って再実行する．

原因が分からない場合は，たいてい`backtrace`関数を実行し，どのような経緯で関数呼び出しが行われて来たかという履歴を見る．これによって，エラーを起こした関数を呼び出した関数がわかる．そこで呼び出し側の関数の定義を`getd`などで見ると呼び出し形式がわかる．この関数定義を見ても原因をつかめない場合は引数となった変数の値を見る．また場合によってはシンボルのプロパティリストや，ベクタの要素，シンボルの束縛の様子（`oldvalue`）なども見なければならない．

このようにエラーハンドラではヒープ内のオブジェクトの調査（inspection）が行われる．オブジェクト調査に用いられる関数は「参照」関数がほとんどである．例えばシンボルに対しては`getd`，`get`（プ

ロパティ），symbol-value，リストに対してはcar，secondなどである．ユーザはデバッグ中同種の参照関数を何度も入力しなければならない．

また参照関数の引数には同じものが与えられたり，前回入力したものの結果もしくはその一部であることが多い．なぜならスタック中にあるオブジェクトからポインタを繰り返したどることによって，デバッグが進められるからである．画面に表示されているものを繰り返し引数に入力することは，ユーザにとって苦痛である．

従ってシステムが下のようなインターフェースを提供すると，デバッグ作業は簡略化できる．

- 画面に表示されているオブジェクトをユーザが簡単な形式で指定できる
- それに対する参照関数が簡単な形式で呼び出せる

3 リストの表示方法

前章でも述べたが，デバッグの過程においてリストの一部を取り出す関数が呼び出されることが多い．これは大きなリストを画面に表示する際，スクロールが起こるのを避けるために，一部の要素が省略されて表示されることが多いからである [6]．

表示のための省略 (holophrasting) は，リストの先頭からの距離によって行われる．先頭からのcarポインタの数はリストの深さ，先頭からのcdrポインタの数は長さを表す．システムのプリンタはリストを表示する際，深さもしくは長さがある限界値を越えた要素を，"?" などの短いシンボルに置き換えて省略する．限界値はUtiLispの場合，printlevel, printlengthというLisp変数であり，ユーザがこれらに値を束縛することによって，リストの出力を制御することができる．

エラーハンドラの中で大きなリストを調査する際，このように省略された要素を見るには，second, nth, nthcdrなどのリスト参照関数を用いる．これを繰り返し用いれば，リスト中の任意の要素にたどりつくことが可能である．省略を決める距離の起点となるのは常にリストの先頭なので，先頭から遠い要素を取り出すためには参照関数を何度か実行しなければならない．この場合，2章で述べたのと同じように，同種の参照関数を画面上に出ているリストに対して繰り返し適用される．2つの限界値に非常に大きな値を束縛すると，すべての要素が表示されるが，この場合はスクロールが起きる可能性がある．

リストの出力には清書 (pretty printing) という問題も生じる．リストをprind関数によって出力すると，適当にインデントが挿入されて構造が分かり易くなる．しかし，インデントや改行が多くなる分表示に必要な行数が増え，スクロールも頻繁に起こる．prindにはオプション引数があり，printlevel, printlengthに当たるものを指定すればリストの一部を省略することも可能であるが，実際にはユーザがこれら限界値を制御しながらリストを出力するのは困難である．

4 List Windowのインターフェース

前章で述べたリストの表示方法の問題点を解決するのがList Windowの目的である．当システムで提供するものは下のとおりである．

- リストの任意の要素に焦点（ユーザの注目点）をおくことができる [1]
- 焦点からの距離によって要素の省略を決め，リスト全体が画面に入るようにする
- リストは清書して構造が分かるようにする

同じ問題を解決する方法として，テキストを扱うページャやウィンドウシステムのようなインターフェースも考えられる．リストをファイルまたは文字列に清書しておき，画面に表示可能な部分だけを表示し，上下にスクロール可能にするものである．しかし，リストが大きくなるとこの方法では構造が分かりやすくなるとは言い難い．なぜなら構造的に近くにある要素が，清書したときに何行も離れて表示されてしまうことがあるからである．

4.1 焦点と距離

焦点はリストの中でユーザが注目している要素で，任意のサブリストまたはアトムに置くことができる．表示の際の省略は，焦点からの距離によって決められるが，その距離はcdrポインタの数，すなわち長さ方向の距離だけである．深さ方向はどれだけ離れていても省略には関係ない．これは一般的にはリストはcdr方向に要素を連結していることが多いからである．

リスト全体がスクロールが起こることなく1画面に表示されるように，適当な限界値が決められ，それよりも遠い要素は省略シンボル"?"に置き換えられる．焦点から前後両方向にある要素が省略の対象となる．これによって焦点の近傍だけが画面に表示される．

4.2 焦点移動によるリストの調査

List Window によって画面上にリストが表示されると，その先頭要素にカーソルが置かれる．このカーソルはスクリーンエディタのように，簡単なキー操作（vi と同じ h/j/k/l）によって画面上を動かすことができる．画面上に表示されている任意の要素の上にカーソルを移動して，「焦点移動」コマンドを実行すると，その要素に焦点が移り，省略が再計算されて表示される．焦点移動を繰り返すことによって，ユーザは任意の要素を参照できる．

図1にリスト内を探索する様子を示す．test:prind というシンボルの関数定義リストを調査すると，ユーザはまず (a) を得る．焦点は最初先頭の defun にある．ここでカーソルを動かして funcall に焦点を動かすと (b) を得る．ここでは焦点の近傍が表示され，リストの先頭は省略されている．さらに p:symbolp に焦点を動かしたものが (c) である．

4.3 List Window の実装

ここでは，前節で述べたような List Window のインターフェースをどう実現したかを説明する．

List Window は，まず与えられたリストと焦点からどの要素を省略するかを決める．そのために List Window では，距離リスト，属性リスト，省略リストという3つの作業リストを使用している．図2に例を示す．関数 foo の定義のうち，仮引数となった最初の x に焦点が置かれている．

距離リストと属性リストは，元のリストと同じ構造を持つ．前者の各ノードは，対応するアトムと焦点との間の距離である．後者は対応するアトムのタイプ，atomlength（出力したときの文字数），それに実際のオブジェクトへのポインタを組にして持っている（図2において，丸囲みのアルファベットがポインタである）．次にこの2つを元に，焦点から遠い要素を省略して省略リストを作る．ある限界値 Weight よりも距離が大きい要素は省略される．このリストには，属性リストからとったノードのタイプや実体へのポインタなども含まれる．

画面には省略リストを清書するが，実際に出力する前に，出力に必要な行数を計算する．この際省略リストにある文字数を用いる．行数が画面の行数を越えた場合は，出力が不可能なので，Weight を1減らし，ふたたび省略リストを作る．

画面に収まる省略リストが得られたら，これを実際に出力する．この際画面に文字を出力するのと同時に，省略リストから情報を取って screen-attribute というベクタに書き込む（図3）．画面上の文字それぞれに1つのスロットがあり，その内容はオブジェクトのタイプと実体へのポインタである．これによってユーザが画面上で指定したオブジェクトを拾うことができる．

限界値 Weight は，距離リストを作ったときの最大距離を初期値として与えてやれば，必ず最適の解が得られる．しかし出力するリストが大きくなると，焦点を動かすたびにとても出力できないような大きな初期値から試行を繰り返すことになる．こうなると焦点の移動に時間が掛かり，ゴミを大量に作ってしまう．これをふせぐために，焦点移動があった時は，前回の Weight に2を加えたものを初期値として与えている．このため，焦点を動かしたときに省略が多過ぎることもあり得る．その場合は，同じ要素に対して焦点移動のコマンドを繰り返し実行すれば最良の解が得られる．

5 List Window の応用

この章では，List Window をインターフェースとして用いる応用例を示す．最初は関数 inspect で，シングルプロセスの UtiLisp のエラーハンドラの中で利用される．次に，mUtiLisp における並列プロセス用のインターフェース，最後に構造エディタについて述べる．

5.1 関数 inspect によるオブジェクト調査

List Window により1つのリストを画面上で調査することが可能になり，リストの参照関数を明示的にユーザが入力する必要はなくなった．関数 inspect はこれを応用し，さらにシンボルなどアトムの中から出るポインタを参照する関数の実行を簡略化する．これによって2章で上げた調査を支援するインターフェースを実現する．

この関数は，引数として与えられたオブジェクトを List Window によって画面に出力する．ユーザが画面上のリスト中に現れたシンボルの値を取りたい場合は，そのシンボル上にカーソルを移動し，キーコマンド v(alue) を実行する．するとシステムは，その値を List Window によって同じ画面上で扱う．この時，古いリストは捨てずにスタックに保存し，新しいオブジェクトはその上にプッシュされる．ユーザは，このオブジェクトスタックをポップすることによって元のオブジェクトを回復することができる．シンボルの値や関数定義を取ったり，スタックをポップする操作はキータイプによって実行できる．

シンボル以外のアトムからもさまざまな情報が抽出できる．例えば，ストリングからは文字数，ベク

```
(defun test:prind (x indent close (asblock)) (test:tab indent)
 (cond ((p:insertingp x) (test:prin1 x))
       ((p:atom x) (incr column (fourth x)))
       ((p:printable x (min (/- leng column close) 80)) (test:prin1 x))
       ((eq (car x) (quote *DOTTED*)) (test:dotted x indent close))
       ((eq (car x) (quote *FOCUS*)) (test:focus x indent close))
       ((eq (car x) (quote *ELLIPSIS1*)) (incr column 1))
       ((eq (car x) (quote *ELLIPSIS2*)) (incr column 2))
       (t (incr column 1) (funcall (cond (?? ?) ?) ??) (incr ??)))))
```

(a) 焦点はdefun上

```
((?? ((eq (car ?) ??) (incr ??)) ((eq (car x) (quote ?)) (incr column ?))
  (t (incr column 1)
     (funcall
      (cond ((or asblock (< (/- ??) ?)) (function test:block))
            ((p:symbolp (fourth x)) (or (and ?? ?) ?? ?))
            ((or (atom ??) ??)) (function ?))
            (t ??))
      x (/1+ indent) (/1+ close))
         (incr column 1))))
```

(b) 焦点はfuncall上

```
(((?
   (funcall (cond ((or asblock (< ?? ?)) (function test:block))
                  ((p:symbolp (fourth x))
                   (or (and (every ?? ?) ??) (get ?? ?) ??))
                  ((or (atom (fourth ?)) (and ??)) (function test:block))
                  (t (function ?)))
      x
      ??
      ?)
   ?)))
```

(c) 焦点はp:symbolp上

図1　リストの調査

タからは各要素ならびに要素数，コードビースからは関数名，最大引数，最小引数，ストリームからは入出力モードなどである．このうちベクタの要素の場合，ベクタ全体をリストに展開してやれば，List Window で調査が可能になる．それ以外の情報については，数や文字列など，それ以上の調査が必要となるものではない．これらについては，describe コマンドを実行して画面の最下行にまとめて情報を出力するようにしている．

inspect を用いることにより，ユーザは最初に引数として与えたオブジェクトからたどれるオブジェクトはすべてたどれる．引数の既定値は，(backtrace) である．これは，多くのユーザがエラーハンドラの最初にスタックの履歴を見ることが多いからである．また関数 inspect はエラーハンドラの中だけでなく，トップレベルからでも呼び出すことが可能である．

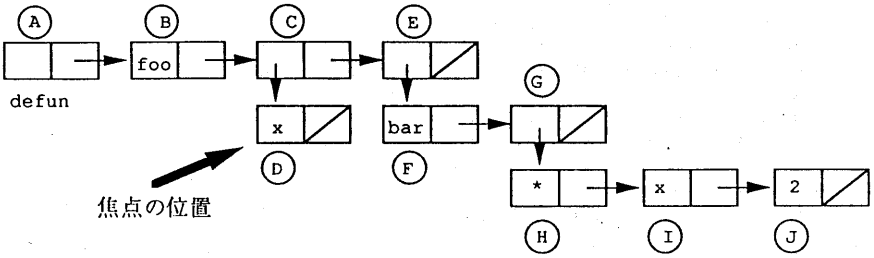
5.2　並列 Lisp におけるプロセスモニター

List Window のもう 1 つの応用例は，mUtiLisp Shell [5] である．マルチプロセス Lisp システム mUtiLisp [2, 3] は，オブジェクトを共有しない並列性を提供する．このシステムの中では，動的に作られるプロセスがすべて固有のオブジェクトを持ち，メッセージによりプロセス間通信を行っている．このような環境のもとでユーザプログラムをデバッグするのが mUtiLisp Shell である．

各プロセスの中でエラーが起きた場合，通常のエラーハンドラが動いてしまうと複数の read-eval-print ループが並列に動いてしまうことがあり，望ましくない．そこで，ユーザとのインターフェースを行う専門のプロセスを 1 つ作り，一般のプロセスはエラーが起きたときにエラーハンドラを起動せず，中断（suspend）するようにしておく．

あるプロセスのオブジェクトを外から参照するに

リスト　(defun foo (x) (bar (* x 2)))



距離リスト　(2 1 (0) (1 (2 3 4)))

属性リスト

((SYM (A) 5) (SYM (B) 3) ((SYM (D) 1))
((SYM (F) 3)) ((SYM (H) 1) (SYM (I) 1) (FIX (J) 1)))

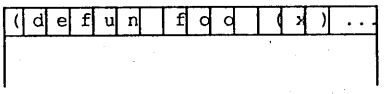
タイプ　ポインタ　文字数

省略リスト(Weightが２の場合)

(LIST (TOP) (SYM (A) 5) (SYM (B) 3)
(LIST (C) (SYM (D) 1))
(LIST (E) (SYM (F) 3)
(LIST (G) (SYM (H) 1) (ELLIPSYS (I) 1))))

図２　作業用リスト

画面への出力



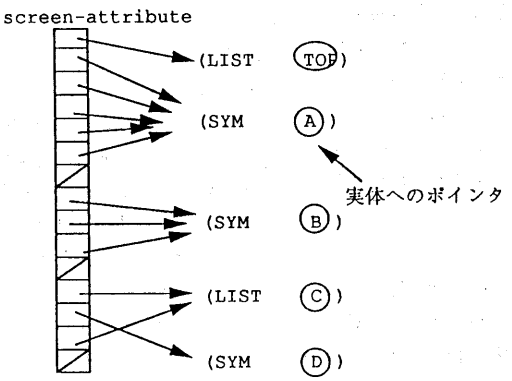


図３　画面情報

は，関数 remote-eval を用いる．これは，引数と任意の Lisp オブジェクトを引数にとり，対象となるプロセスでそのオブジェクトを評価して，その結果をメッセージで返す．

```
(remote-eval 'proc00001 '(backtrace))
```

他プロセスの状態を見るのに毎回このような関数を入力するのはたいへんである．そこで，このためのインターフェースに List Window を用いている．

図 4 に mUtilisp Shell の画面を示す．画面は 3 分割され，最下のものが通常のトップレベルの入力を行う．最上のウィンドウは現存するプロセスのリストを表示するもの，中央は各プロセス内のオブジェクトを出力するもので，両方とも List Window を動かしている．

エラーが起きたとき，もしくはユーザが明示的に指定したとき，全ユーザプロセスのリストが上のウィンドウに表示される．ここでユーザは調査したいプロセスを選択し，スタックやメッセージ履歴などとりたい種目を選択する．するとそのオブジェクトが 2 段目のウィンドウに表示される．これ以後は，関数 inspect と同様にポインタをたどって，そのプロセスの中の状態を見ることが可能である．

5.3 構造エディタ

List Window のインターフェースを用いて，リストを調査するだけでなく修正できるように拡張したものが構造エディタである．OS が提供するウィンドウシステムの元で Lisp を使用している場合には，プログラムの編集は，外部のスクリーンエディタでプログラムファイルを修正するのが一般的である．ここではオブジェクトの調査中にエラーの原因が分かったときなどに，一時的にオブジェクトの状態を変更するのが目的である．

編集は，キーに編集コマンドを割り付けることによって行う．要素の削除，コピー，ペーストなどはキーコマンドによって実行可能である．ただし，挿入や置換などで Lisp オブジェクトをキーボードから入力するときは，ライン入力を行う．これは，リストなどを入力した場合に，最後の右カッコを入力するまでリスト構造にすることができず，入力中のリストと焦点との距離を求めることが不可能なためである．このため画面の下 2 行を入力用に用いている．

6 まとめ

List Window は，大きなリストを限られた文字画面の上で見る手段を提供した．リスト中の任意の場所に焦点を置き，そこから構造的に遠い要素を省略し，リスト全体を清書する．ユーザは画面上でカーソルを動かし，キーコマンドによって焦点を動かしてリスト全体を調査することができる．さらに List Window を応用した関数 inspect ではリスト中のシンボルの値や関数定義をとれるようにし，あるオブジェクトから出るポインタをすべてたどることを可能にした．inspect によって，エラーハンドラで行われていたデバッグ操作は，大幅に簡略化された．

6.1 問題点

List Window は，ループ構造を含むリストには対応できない．これは，リスト中に焦点を置いているため，2 つのポインタが同じリストを指している場合などには，焦点が複数できてしまうことがありうるからである．また，ポインタが自分自身や自分の祖先を指しているような無限ループを含むリストは処理できない．いずれの場合も，最初に対象リストを作業用リストにコピーする時点で構造が壊れてしまったり，スタックがあふれてコピーできなくなるかのいずれかになってしまう．

6.2 今後

現在，構造エディタは関数 inspect とは個別に開発しており，組み込まれてはいない．これは，ヒープ調査中にオブジェクトを変更することにより，関数 inspect の持つオブジェクトスタックの一貫性が失われることが起こり得るからである．これについては，変更したリストの書き戻しに何らかの制限を加えることで，解決できると考えられる．

オブジェクトの調査と修正，およびファイル操作を提供することにより，デバッグだけでなく開発やファイル管理など，プログラミングのすべての段階を援助することが可能になると思われる．

もう 1 つの拡張は，X ウィンドウなどの上で List Window 用のウィンドウを作ることである．この場合，表示方法は現在のものと変わらないが，現在キーボードで行っている操作のほとんどが，マウスのキークリックやメニューによって行われることが可能になる．

7 Acknowledgement

本研究の主要部分は，著者が東京大学在学中に博士課程の研究の一部として行っていたものである．東京大学工学部計数工学科の和田英一教授ならびに和田研究室の方々に感謝いたします．

参考文献

[1] George W. Furnas. *Generalized Fisheye Views*, Human Factors in Computing Systems, CHI '89 Conference Proceedings, ACM April 1986

[2] Hideya Iwasaki. *mUtiLisp Manual*, METR 88-11, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, August 1988

[3] 岩崎, 寺田, 湯浅: UNIX 上で動作する mUtiLisp システム, 記号処理研究会, 情報処理学会, 48-3, 1988

[4] Wada Laboratory. *UtiLisp Manual Revision 2.0*, Department of Mathematical Eng. University of Tokyo, January 1988

[5] Kei Yuasa. *Process Monitoring Interface for Multiprocess Lisp*, Doctoral Thesis, Faculty of Engineering, University of Tokyo, December 1988

[6] 和田英一, *Lisp* ウィンドウ（または *S* ウィンドウ）, 第30回プログラミングシンポジウム, 情報処理学会, 1989, 1

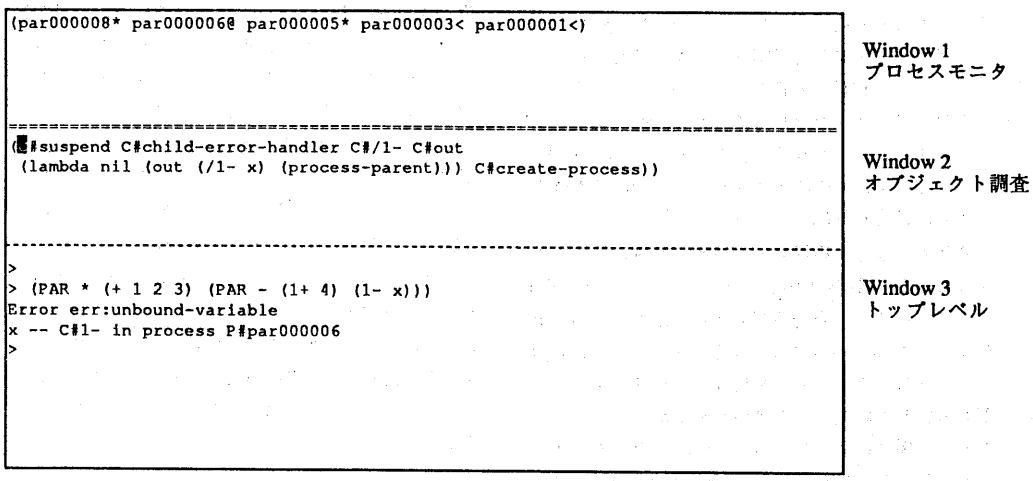


図4　mUtilisp Shellの画面