グラフィック・ディスプレイを多用した Lisp プログラミング環境

斎藤康己
（日本電信電話公社　武蔵野電気通信研究所）

0. はじめに

筆者はプログラミングを総合的に援助するシステムとして"アプレンティス"（apprentice，弟子という意味）の提案を行なってきた。[1][2]　本稿ではアプレンティスの基本的な考え方を述べるとともに，その実現に向けてLispによるプログラミングを対象として試作しているシステムについて報告する。　具体的にはLisp プログラムの動きをグラフィック・ディスプレイ上にわかりよく表示するシステムについて，その機能，インプリメント法などを説明し，本システムの利用に際して得られた経験について考察を加え，今後の課題にも言及する。

1. アプレンティスとは何か

1.1. アプレンティスの目標

プログラミングは実に多くの活動を含んでいる。　例えば，アルゴリズムの考案，コーディング，プログラムの入力，テストラン，机上デバッグ，プログラムの修正などである。　そこではプログラマを取巻くすべてのもの～机，マニュアル，リスティング，システムを熟知している人…～がプログラミングの環境となる。　その環境の役割を計算機システムで完全に代行しようというのがアプレンティスのねらいである。　そのためにはプログラマとアプレンティスの間に豊かなインタラクションが可能でなければならないし，そのインタラクションを意味あるものとするためにアプレンティスはプログラミングやプログラマに関する多くの知識を持っていなければならない。　さらに将来的には，一般的な知識を使った知的な会話機能等も必要になってくる。　ここではまず豊かなインタラクションを可能にするための第一歩としてアプレンティスの側の表現力を増すことを考える。　従来の計算機システムはプログラマが指定した出力以外ほとんど何も外に示さず，何をしているのかがプログラマに見えないことが多かった。　この点を改めて計算機がより適切に自分の活動を報告するようになれば，必然的にプログラマにとって有意義なインタラクションの機会が増えるはずである。

1.2. グラフィック・ディスプレイの利用

アプレンティスの表現力を実現するには，計算機と人間の接点になっているターミナルに多くの情報を適切に表示しなければならない。　そのためにはできるだけ大きなスクリーンを持つディスプレイ・ターミナルが必要である。　さらに人間にとっては図形的表示が文字よりもはるかに理解しやすいものであることを考えれば[3][4]，グラフィック・ディスプレイの利用は不可欠である。

1.3. これまでに行なわれた研究

アプレンティスの提案は[5][6]にみられる。　Winograd の[7][8]もアプレンティスの必要性を強調したペーパーである。　ディスプレイの利用に関しては Smalltalk－72[9]や Teitelman のプログラマズ・アシスタント[10] にすばらしい例がみられる。　O/S のコマンドプロセッサも含めた全システムを EMACS という強力なスクリーン・エディタのもとに統合してしまった[11]の例も示唆に富んでいる。

2. アプレンティスの画面に要求される機能

前節で述べたアプレンティスの構想は，O/Sまでも含めて考えなければならない厖大なもので，そう簡単に実現できるものではない。 そこでまずLispという言語を取りあげ，その言語プロセッサ（インタプリタ）をアプレンティスに見立てて，Lispプログラムの動きをうまく画面上に表示する方法を追求することにした。 その際，以下の機能が基本的であると考えた。

2.1. プログラムとその効果が同時に見えること

EMACS[12]のようなスクリーン・エディタが従来のエディタに比べて格段に使いよいのは，ユーザーの打ち込む各コマンドの効果がすぐに画面上に見えるからである。 同じことがプログラムについても言えるはずである。 プログラムテキストと実際のプログラムの効果や結果との対応がうまくつかないために我々はデバッグに多くの時間を費す。 プリント文の挿入がデバッグの常套手段になっているのもこの事実のあらわれである。 そこでプログラム・テキスト，その上でのコントロールの位置，その部分の実行の効果（データの変形，出力など）を常に同一画面上に表示しておくことにする。

2.2. 画面上に見えているものは画面上で編集できること

計算機が行う多くの仕事がエディタというコンテクストの中で実行できることは以前から指摘されていた。[13] ひとたび画面上に表示され，プログラマの目に見える状態になっているデータはいつでも容易に変更できるようになっているべきである。 Lispの中でミスタイプをした時に思わずエディタの削除コマンドを打ち込んでしまう所謂モード・エラーの頻出が逆にこの機能の有効性を裏付けてくれる。 この機能によってプログラマはエディタと言語プロセッサの間を往復する煩わしさから解放されて，あたかも一枚岩のアプレンティスと対話をしながらプログラムを作っていくという感触を得ることができる。

2.3. 省略表示

いかに大きなディスプレイを使おうとも一画面に表示できる内容には限りがある。 また大きなプログラムを一度に細部まで画面上に表示するのは，かえって人間の理解をさまたげることになる。 そこで，省略表示が重要になってくる。

2.4. 複数ウィンドウ（ボックス）の表示

画面上に表示すべき情報は多岐にわたり，その量も多い。 したがって，それぞれ別の領域に分けて表示するのが得策である。 その便利さはSmalltalk[9]やTeitelmanのシステム[10]で実証ずみである。

3. 実現したシステムの使用例

本システムではグラフィック・ディスプレイとしてTektronix 4016を利用している。 現在の所，上記2.1.，2.2.の一部と2.3.，2.4.の機能が実現している。

まずLispを起動し，システムをロードしたあと，ユーザーは自分の関数を定義するかまたはファイルから読み込んでくる。 ユーザーの関数名をSIX（図1参照）とすると，

```
(SHOW SIX)
```

と入力することによって，まずSIXのプリティ・プリントされたテキストが図1のように表示される。 これで準備がすべて整ったことになる。 これ以降，ユーザーが評価した関数の中にSIXが現われると，システムは自動的にプログラム・テキスト上のコントロールの表示を開始する。

図1. グラフィック・プログラムの表示 (Tektronix 4016のハードコピー)

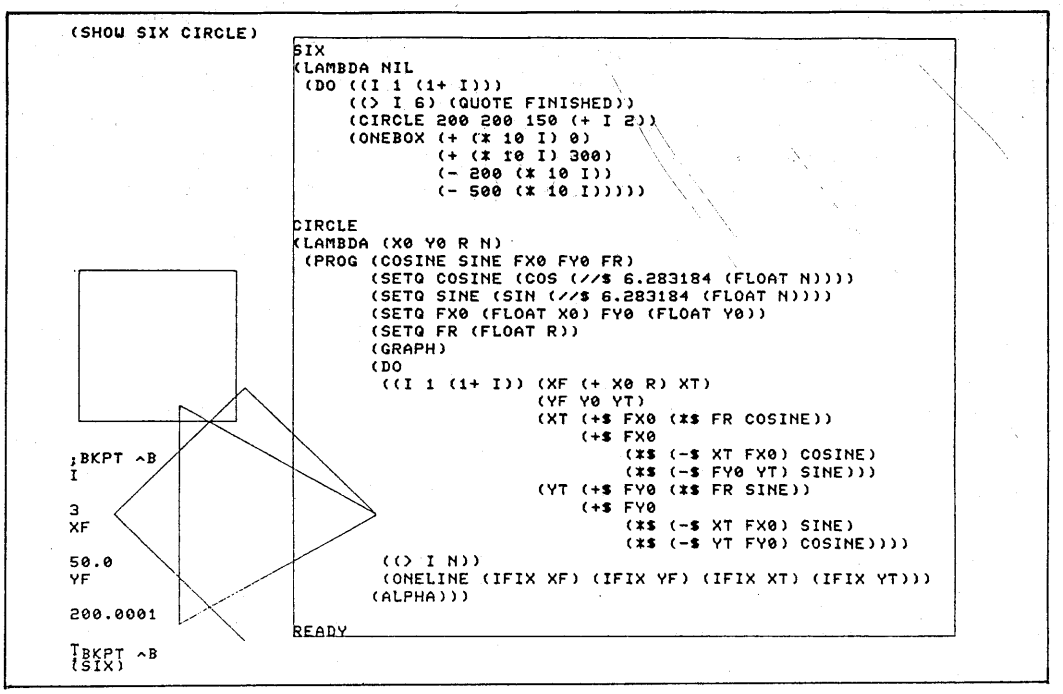
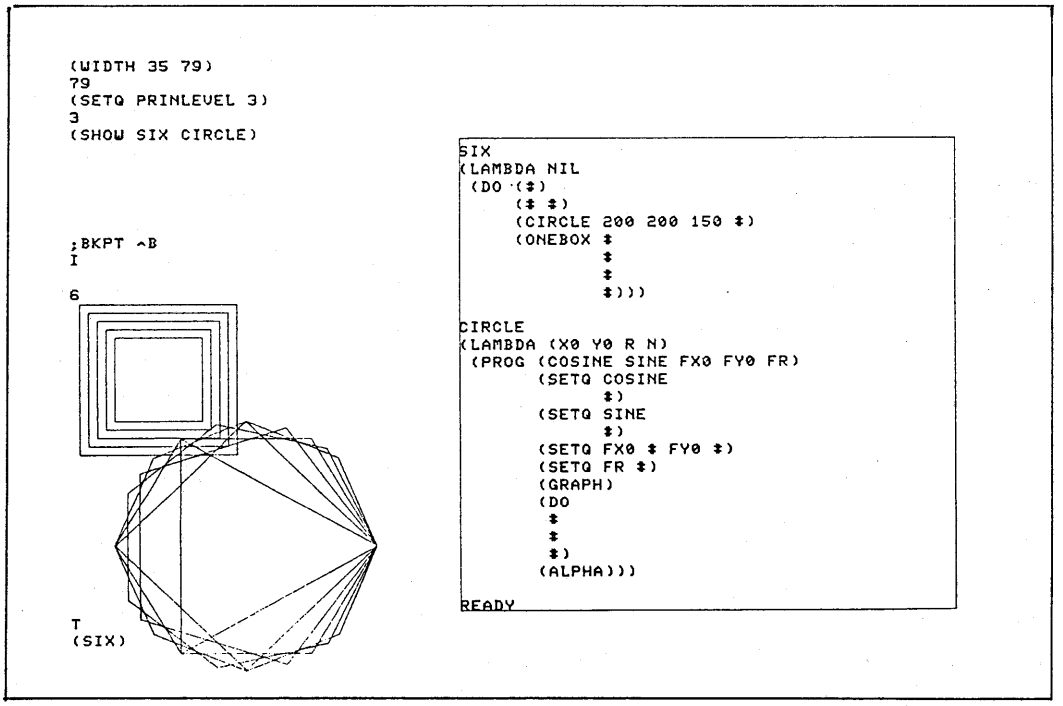


図2. 上のプログラムの省略表示



3.1. グラフィック・プログラムの表示（例1）

図1に表示されたプログラム six は circle と onebox を呼んで画面上に多角形と正方形を6個ずつ書くプログラムである。（six）と入力してこのプログラムを起動するとカーサーが DO の直前の括弧の位置に移動して1秒間点滅し，コントロールが今まさに DO に入ろうとしていることを示す。次に1の上にカーサーがきて，まずIの初期値を評価していることを示す。評価が終了すると評価ずみのS式（今の場合は1）をうすい四角い枠で囲んでそのことを知らせる。（この枠は Tektronix の write-thru というモードで表示しているので管面には蓄積されない。）以下同様にして現在評価しているS式の先頭にカーサーが移動することによってコントロールの所在を常に明示しておく。six の中で circle が呼ばれると当然コントロールは circle に移るのでその時点で，カーサーが circle 内の PROG の直前の括弧の位置へ移るわけである。

図1の例ではプログラマがキーボードからのインタラプトで計算を中断して，内部状態を調べている。この Break の機能は Lisp に元々そなわっている機能であるが，このシステムと同時に使うことによって格段にその使いよさを発揮するようになった。なぜならばプログラマはコントロールの所在を常に見ながら自分の望みの位置でインタラプトをかけられるからである。もちろん必要な変更を施したり，内部変数の値を見たりしたあとで，計算を続行できる。

図2は同じプログラムを省略表示でながめている所である。テキストを表示するボックスの大きさは内容に応じて小さくなっている。ここでの省略は単純にリストのレベルによって行なわれていて，あらかじめセットしたレベルより深いリストの要素はすべて "#" に置きかえて表示している。コントロールの表示は図1の場合と同じように進行するが，このような省略表示の上でながめるとプログラムのグローバルな動きをはっきりとつかむことができる。さらに省略を強くしてレベルを1にし，多くの関数を同時に表示すると関数間の呼びの状況を追うこともできる。

3.2. 入出力を含むプログラムの表示（例2）

図3に示したプログラム fortread は FORTRAN から Lisp への変換プログラム用の読み込みルーチンで，ファイルから FORTRAN の一行を読み込んでリストとして返すプログラムである。このように入出力を含む場合，読み込んでいるもの，あるいは書き出しているデータが，プログラムの動きにつれて見えることが肝要である。そこで本システムでは図3のように入出力ボックスを画面上に用意し，入出力はこれらのボックスを通して行うようにしてある。すなわち前記と同様に中央のプログラム・テキスト上でコントロールが表示されつつ，それが入出力の関数の所にきた時にはカーサーが入出力ボックス内の現在の読み込みヌは書き出し位置に移動し，入力の場合は入力関数に応じてそこにある文字，行ヌはS式などを取り出して来る。その結果カーサーは次の読み出し位置に移動し，コントロールは元のプログラム・テキスト上にもどる。出力も同様で，出力ボックス内に次々と出力されていく。入出力ボックスはそれぞれEMACSでサポートされているので，計算の途中や計算が始まる前に入力ボックス内にテスト用のデータを準備したり，カーサーを移動しておくことによって通常とは違う位置から入力を開始したりすることも容易にできる。

図3では各S式の評価が終了した時点で write-thru モードではなく，実際に

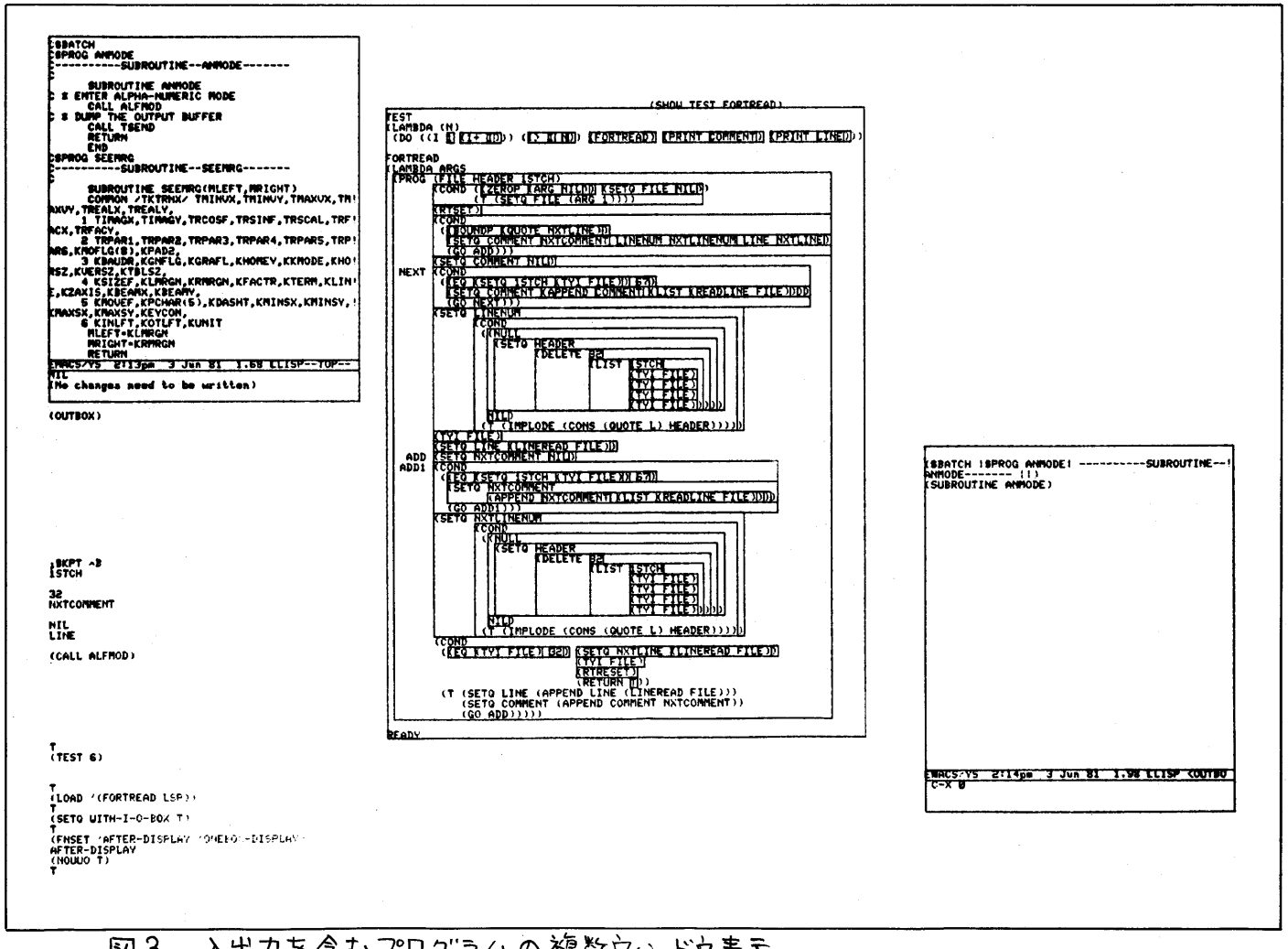


図3. 入出力を含むプログラムの複数ウィンドウ表示

四角い枠を管面に書き込んでしまってプログラム・テキスト上に軌跡が残るようなモードで動かしたもののハードコピーなので中央のプログラムテキスト内には多くの枠が書き込まれている。

4. インプリメント

4.1. コントロールのプログラム・テキスト上での表示

SHOW に渡されたユーザ関数は画面上にプリティプリントされるとともに，各部分の画面上での位置情報を含むように変形される。 通常の eval（Lisp のS式を評価する関数）は変形した関数を受けつけないが，次に述べる evalhook という機能を使うことによって，この位置情報を有効に使うことができるようになる。

4.1.1. Evalhook

Lisp の eval に代わって，eval にS式が渡されるたびに，そのS式をもらってユーザが任意の関数を実行できる機能である。 ユーザは eval の代行をさせたい関数を evalhook というグローバル変数にセットしておけばよい。 この関数として，受け取ったS式に位置情報が含まれている場合には，値の評価の他に，画面上のカーサーをその位置に移動したり，枠で囲んだりするルーチンを使う。

4.1.2. 関数定義への位置情報の埋め込み

ユーザの関数定義が与えられた時に，それをプリティプリントしつつ，各部分の画面上での位置情報を埋め込む変形は，次の2段階に分けて行なわれる。

1) 元の定義式のうち評価される部分（evalに渡される可能性がある部分）にマークを付ける。 たとえば，

```
(Lambda (X) (setq Y 3) (print (list X Y)))
```

という式は，'*' でマークを表わすことにすると，

```
(Lambda (X) (*. (setq Y (*. 3)))
                (*. (print (*. (list (*. X) (*. Y))))))
```

のように変形される。

2) 元の定義式とマークされた式とを受けて，定義をプリティプリントしつつ，マークの付いている式のみ，位置情報を埋め込んでいく。
たとえば，上の (*. (setq Y (*. 3))) の部分は，

```
((26.34) (setq Y ((26.42) 3 (26.43))) (26.44))
```

のように変形される。 ここで (26.34) というドッテドペアは，(setq Y 3) の先頭の括弧の位置が，26行目の34文字目であることを示している。 (26.44) は最後の括弧の次の位置である。（実際は行とコラムではなく絶対座標を使っているが説明上こうした。）

実際のインプリメントでは，この他に省略表示のために，レベルの設定に応じて，あるレベル以下はマークを省略し，プリティプリントする時に，設定されたレベルより深いリストは "#" で置きかえて省略表示するようにしてある。

コントロール位置の表示そのものは，以下のどれかを自由に選べる。

- カーサーの移動によるもの．
- write-thru モードで四角い枠で囲うもの．
- 管面に蓄積するモードで四角い枠で囲ってしまうもの．

カーサーや枠を1つの位置にどの程度の時間とどめておくのがよいかは実験的に求めたが，その時間は約1秒である。

4.2. 入出力ボックスの表示

図3のような複数のボックスをそれぞれEMACS というスクリーン・エディタを利用して表示するためにシステム全体は図4のような構成をとる。 ここで矢印はデータ及び制御の流れを示す。

図4で，曲線で囲んだ部分が現在インプリメントが完了している部分であるが，まずEMACS をTektronix 4016上で動くようにするために必要だった準備について述べ，次にファイルを利用した各プロセス間の交信について述べる。

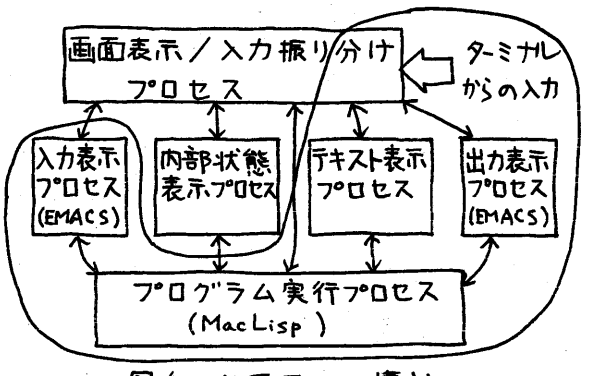


図4. システムの構成

4.2.1. EMACS の改良

EMACS は大変すぐれたスクリーン・エディタで，入出力ボックス等をサポートするエディタとしては申し分のないものである。 ところが，EMACSは20数種のターミナルをサポートしているにもかかわらず，Tektronix 4016（蓄積型）はサポートしていない。 蓄積型のディスプレイはスクリーン・エディタ向きでない[14]ということらしい。そこでTektronix 4016用に，以下のような改良をEMACS に加えた。

1) スクリーン・クリア，カーサーの移動等の基本ルーチンをつけ加えてまず一応Tektronix 4016上で動くようにすること。
2) Tektronix4016上に4種類ある文字の大きさのどれででも動くようにすること。
3) 画面に表示するバッファの大きさを可変にし，また画面上の任意の位置にバッファを表示できるようにすること。
4) Tektronix 4016では部分的な消去は全くできないので，全画面の消去が頻繁におこることになる。 それを極力おさえるようにすること。

4.2.2. プロセス間のファイル経由の交信

Lisp（Maclisp）とEMACS との間にはもともとファイルを介しての交信手段が確立していて，EMACS の中で作ったプログラムをLispの環境の中へ持ち込むことができるようになっている。 実はちょっとした改良でLisp の中で作ったものをEMACS の中へ持ち込むことも可能である。 そこで本システムでは入力するものをEMACS のバッファ内からLisp に取り込んだり，出力結果をLispからEMACS のバッファへ送り込んだりするのに（すなわち図4の矢印にあたる部分）ファイルを利用している。

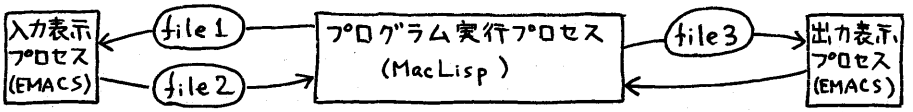


図5

図5でfile1には入力関数（TYI, LINEREAD, READCH など）に応じて，EMACS内でカーサーをどのように動かして，バッファ内のどの部分をどういう形で入力として切り出してくるべきかが，EMACS で実行できるように書かれていて，返事

として返ってくる file2 内には読み込まれたものが，

(setq *return-value* ～～)

という形で Lisp に評価されるべく入っている。

出力の方の file3 には出力すべき文字列を EMACS のバッファ内に挿入した後 Lispへもどれという命令が書き込まれているので，出力ボックスをサポートしている EMACS はこれを読み込むと，出力文字列をバッファに挿入し，すぐに Lispへもどるという動作をする。

ファイルを使うこの方法は効率という観点からはよい方法ではないが，既存のソフトウェア間のデータのやりとりには便利な方法である。

5. 既存のLispデバッギング・ユーティリティとの比較

5.1. Traceとの比較

Trace は指定した関数への入口でのパラメータの値と出口での return value を次々に出力する機能である。 いわば，関数をブラックボックスと見立てているわけで，どのコールでエラーまたは期待に反した結果がおこったかをみるのには適しているが，ある関数の中のどこが悪くてそういう結果になってしまったのかを見るのには適していない。 それに比べるとここで紹介したシステムを使うと，関数の定義の中までが表示されて，より詳しいレベルでプログラムを追いかけることができる。

5.2. Breakとの比較

システムの動きを説明した所で述べたが Break 単独よりも本システムと Breakの両方を使うことによって，より豊かなインタラクションが可能になる。従来はソーステキストを編集して，プリント文などを挿入しなければ見ることのできなかったプログラム内部のローカル変数や，種々の内部状態を，まさにある条件が成り立った瞬間に Breakして見ることができる。 従来からも Break にはある条件が成立したときにのみ動作するというような機能もあったが，それは条件を正確にプログラムテキストの中に書き込んで走らせなければならなかった訳で，それに比べるとターミナルからのインタラプトで直ぐに Breakできるというのは大きなメリットである。 これによってプログラムあるいは言語システムとの対話的なやりとりの機会も大いに増えることになる。

5.3. Step との比較

Stepはアセンブラーのステッパーのように Lispの S 式を 1 つずつ評価しては結果を打ち出すユーティリティーである。

本システムとの一番大きな違いは，プログラムテキスト全体の中でどこを評価しているのかが必ずしも明らかにはならないこと，及びユーザーの側に完全にイニシアティブがあって，言いかえると本システムのように黙ただ走らせておいて眺めているというわけにはいかないという点があげられる。

目的の（すなわちエラーが発生する近くの）所へたどりつくまでに手間がかかることがあるのは両者に共通の欠点で，本システムでは省略表示によってそこを補っている。 逆に本システムの側の弱点として各 S 式が評価された時の値が表示されないという点があげられる。 単に値を出力するだけならば簡単ですぐにできるが，それを画面上で編集も可能にし，さらに図形として表示しようという目標があるので，まだ実現していない。

6. 考察と今後の課題

6.1. 考察

前節の比較検討からも明らかなように，本システムに使われているプログラムの表示方式はアプレンティスのごく一部とは言え，従来あまり見えなかったプログラムの動きを見せるという観点から大きな可能性を含んでいる。 いくつかのプログラムを本システムを使って眺めた経験から以下のようなことが言える。

1.) 常にプログラム・テキスト上にコントロールの所在が見えることによってプログラムが期待どうりに動いているという実感と安心感を得ることができる。

2.) 何かエラーが発生したときに，エラー発生までの履歴を見ていたことによって，あらぬ部分を疑うことがなくなり，エラーの原因となっている部分にすぐに注意を集中できる。

3.) 省略表示は強力で大まかな動きをとらえるのに便利であるが，単純なレベルによる省略なので，見たい部分が見えなかったり，どうでもいいものが見えたりしてしまう。 テキストとしてはできるだけ多くを表示しておいて，カーサーの移動をあるレベル以下は省略するモードが良いのではないかと思われる。

4.) コントロールのみが見えている状態は，まだプログラムが何をしているかがよく理解できるという感じからはほど遠い。 画面に図を書くプログラムや，ファイルからの入力に対して単純な変形を施すようなプログラムでは書かれた図や出力を眺めていれば何をしているかは一目瞭然であるが，内部に保持したデータを次々と変形していくようなプログラム（たとえば sort ）では，内部データの変形していく様をコントロールの所在と同時に画面上に見せれば，よりわかりやすくなると思われる。

6.2. 今後の課題

6.2.1. データ・内部状態の表示

前節 4)でも述べたように，内部データの表示は是非とも必要な機能で，これが加わるとアプレンティスの機能は何倍にもふくれあがるはずである。

リフレッシュ型で大型のディスプレイを利用すれば，減少する方向の変化も楽に表示できるので，内部状態，データ等の表示用には適している。 またリフレッシュ型を使えばスクリーン・エディタとの親和性もよい。 したがってリフレッシュ型のグラフィック・ディスプレイを使用するのがよいと思われる。

Lispのデータはすべてリストであるから標準的には括弧で囲んだリストとして表示することができる。 しかし複雑な構造になってくると括弧は邪魔で，ユーザーがトリーとして扱っていればトリーのように2次元図形として表示したい。その際，ユーザーに理解しやすい図形はどのようなものかとか，一画面に2次元的に表示した内容をどのようにして編集するかなど興味ある問題がたくさんある。

6.2.2. プログラム・テキストの画面上での編集

現在のシステムでは，プログラム・テキストの編集はLisp から一度EMACS に入って行っているが，理想的にはコントロールを表示したテキストそのものをインタラプトなどをかけた状態ですぐ画面上で編集できることが望ましい。 それにはいくつかの方法が考えられる。

1.) EMACS内でプログラム・テキスト上のコントロールの表示をしてしまう方法。 ファイル経由の通信がネックになる。

2.) Lispの structure editor を改良して，スクリーン・エディタとし，それを使って Lisp内で編集を行うようにする。

3.) Lisp machine [15]上のエディタ（EINE, ZWEI）はLispで書かれているし，Multics EMACS [11]はMacLispで書かれている。 このようにLisp そのものでスクリーン・エディタを書いてしまえば，編集も実行もすべてLisp の中でできる。 この方向が一番よいと思われる。

6.2.3. モニタ・ボックス

プログラミングの作業は言語プロセッサの中だけでは完結せず，しばしばモニタ・レベルのコマンドを使う必要が生ずる。 これもモードの切換なしに，画面上の1つのボックス内でコマンドを打つと，そこに必要な情報が表われるようになっていると便利である。

謝辞

日頃研究の機会を与え，見守ってくださる山下第一研究室長，野村調査役に感謝します。 また色々と意見を述べて下さる第一研究室の皆様にも感謝します。

References

[1] 斉藤康己，"プログラミング・アプレンティス ―視覚的プログラミングへの試み―" 昭和56年度信学会総合全国大会講演論文集（第6分冊）P.1495.

[2] 斉藤康己，"ディスプレイ上の複数領域を利用した知的プログラミング環境" 情処学会第22回全国大会講演論文集 4C-1，pp.343-344.

[3] M. Fitter, T.R.G. Green, "When do diagrams make good computer languages?", Int. J. Man-Machine Studies, vol.11, pp.235-261, (1979).

[4] M. Fitter, "Towards more 'natural' interactive systems", Int. J. Man-Machine Studies, vol.11, pp.339-350, (1979).

[5] C.E. Hewitt, B.Smith, "Towards a Programming Apprentice", IEEE Trans. on SE, vol.SE-1, no.1, pp.26-45, (Mar. 1975).

[6] C. Rich, H.E. Shrobe, "Initial Report on a Lisp Programmer's Apprentice", IEEE Trans. on SE, vol.SE-4, no.6, pp.456-467, (Nov. 1978).

[7] T. Winograd, "Breaking the Complexity Barrier again", ACM SIGPLAN Notices, vol.10, no.1, pp.13-30, (Jan. 1975).

[8] T. Winograd, "Beyond Programming Languages", CACM, vol.22, no.7, pp.391-401, (Jul. 1979).

[9] A. Goldberg, A. Kay, "SMALLTALK-72 Instruction Manual", The L.R.G,, XEROX Palo Alto Research Center, (Mar. 1976).

[10] W. Teitelman, "A display oriented programmer's assistant", Int. J. Man-Machine Studies, vol.11, pp.157-187, (1979).

[11] B.S. Greenberg, "Prose and CONS (Multics Emacs: a commercial text-processing system in Lisp)", Conference Record of the 1980 LISP Conference, pp.6-12, (1980).

[12] R.M. Stallman, "EMACS manual for TWENEX users", AI Memo 556, AI Lab. MIT (Aug 1980).

[13] C.W. Fraser, "A Generalized Text Editor", CACM, vol.23, no.3, pp.154-158, (Mar. 1980).

[14] B.S. Greenberg, "Desirable terminal features for support of EMACS", Internal memo on ARPA network. (available as a file in EMACS.)

[15] D. Weinreb, D. Moon, "Lisp Machine Manual", AI Lab. MIT.