Common Lisp とそのアプリケーション = ART

大阪大学 工学部 船舶海洋工学科
服部 真人
(x60630a@ccsun01.center.osaka-u.junet)

1．はじめに

Lisp の歴史は以外と古く、FORTRAN とほぼ同時期、１９６０年に開発されたプログラミング言語です。 プログラムというと我々理工学者はすぐに、足したり引いたり繰り返したりといった数値演算を思い浮かべますが、そういったなんとかの一つ覚えはこの際捨て去りましょう。

コンピューターなるものがこの世に出現したとき、ロケットやミサイルを飛ばして喜んでいた多くの理工学者は、その計算の速さと正確さを愚かにも自分の頭と比べてしまったのです。 その結果コンピューターは今日あるような姿、すなわちただの計算機になってしまいました。 幸いなことに、一部の賢明な学者はコンピューターに知能の存在の可能性を見いだし、人類の良き伴侶としての姿を夢みて、コンピューターに記号の意味を理解させようと試みました。彼らにとって不幸だったことは彼等がマイノリティだったことで、コンピューターの技術は当時のマジョリティによって、それを電子頭脳としてではなく、電子計算機として発展させてしまい、マイノリティは多大な努力を強いられました。その努力の一つの成果が Lisp です。

今日では、データの管理、表やグラフの処理、ワードプロセッサー、そしてコンピューターをベースとする種々のシステムなど、元来の数値計算ではない高度で知的な処理をコンピューターにさせることが多く、昔とは逆に記号処理が主流の時代です。 「人工知能」と言う言葉も最近では聞き慣れた言葉になりました。 昔のように力ずくで方程式を解く時代は終わったと言えるでしょう。 Lisp では連続値を離散化して数値計算をするまでもなく、数式中の記号を直接操作して数式（＝記号）を答えとして出すことが、他の言語に比べて簡単にできます。 Lisp を使えばエディターやコンパイラーを作ることはいとも簡単なことです。 あなたも Lisp の世界に旅立ちませんか？

本稿では、Lisp をこれから始めようという殊勝な方のために、大阪大学計算機センターの Sun-3 ワークステーションで利用可能な "Common Lisp" を例題に、その使い方、プログラムの書き方、リスト処理の例、その他 Lisp にまつわる話を、また、Lisp をベースにするアプリケーションが多数世に発表されていますが、その中の一つ "ART" を、御紹介したいと思います。

ここで一つお断りしておかねばならないことは、私の専攻は船舶工学であって情報工学は専門外であること、つまり私も皆さんと同じ、コンピューターの１ユーザーに過ぎないということです。 ユーザーの立場を活かすことができましたら、幸いです。

2．Lisp を始めよう

まず、センターの Sun-3 の利用申請をして下さい。 大型計算機を利用できる方なら WSTR というコマンドで申請することができます。 詳しくはセンター速報の１６７号を御覧になるか、当センター１階正面の共同利用掛に御相談下さい。

次にやるべきことは、UNIX に馴染むことです。 Sun-3 などのネットワーク環境の優れたワークステーションは、理学・工学などの研究機関に限らず、一般企業でも開発・設計・生産・管理、さらには金融・経営・通信など、あらゆる方面で導入・活用されています。 UNIX はワークステーションの標準オペレーティングシステムとして、業界では常識となっています。もう既に御存知の方は言うまでもありませんが、そうでない方は、将来必ずやお役に立つでしょうから、この機会に覚えることをお奨めします。 UNIX についての解説は本ニュース前号、前々号を御覧下さい。

センターに利用申請をして、少しの間待ちますと、ログインネームとパスワードを知らせてくれます。 大抵は利用者番号に支払いコードをつけたものがログインネーム、あなたの姓名が初期パスワードとして与えられます。 早速ログインしてみましょう。 センターからのお知らせや種々のメッセージがディスプレイに出て、その後、% が出るでしょう。 これが UNIX のプロンプトで、コマンド待ちの状態を示します。 その % の後に、 lisp と打ってリターンを押してみて下さい。 lisp と小文字でなければいけません。図１のように、Common Lisp が起動できたでしょうか？

Command not found と言われて起動できなかった場合、検索パスが不十分なのです。Lisp システムは /mnt2/lisp というサブディクトリーに格納されています。 シェル変数 path の値にこれをセットしてやる必要があります。 vi（エディター）等で .login というファイルの中のどこかに、次の一文を加えて下さい。

```
set  path = ( $path   /mnt2/lisp )
```

.login というような名前がピリオドで始まるファイルは、俗にいう隠しファイルで、普段は表に現われません。

.login に上記の一文を書き加えた後、一度ログアウトして再びログインし直すか、あるいは source .login と入力します。 これで Common Lisp が起動できるようになるはずです。 この操作は一度行なっておけば、恒久的に残ります。 ログインするたびに .login というファイルが評価され、必要な検索パスがセットされます。

エッ！まだ駄目！？ あ、言い忘れましたが、５台ある Sun-3 のうち ccsun01 と名付けられた Sun-3/260 以外では使うことが出来ません。 センターには Sun-3/260 が１台、Sun-3/50 が４台あって５台全てが NFS でつながれていて、あたかも１台のように振舞いますが、Sun-3/50 では主記憶容量が不足するため Common Lisp は使えません。 残念ですが、Sun-3/260 すなわち ccsun01 でお使い下さい。

せっかく吹田キャンパスの辺境の地、計算機センターまで足を運んだのに、ccsun01 は占領されている、そういう時は、ちょっとテクニックを使えば ccsun02 からでも使えます。 つまり、ccsun02 を一つのターミナルにしてしまって、ccsun01 を使えばよいのです。 ccsun01 以外から Common Lisp をお使いになりたい方は、rlogin というコマン

ドを使って ccsun01 にリモートログインしてください。 そうすると、目の前にあるのは ccsun02 だけれども、使っているのは ccsun01 という妙なことが起こります。

```
          ************************************************
          ****  Computation Center, Osaka University  ****
          ************************************************

  *** News from Center (1989.05.26) ***

   1. Telnet is now available for ACOS-2000. Host name of ACOS-2000 is
      acos. ftp and telnet is available.

   2. Japanese TeX and Latex is now available in /usr/local/bin. But
      printing document is little bit complex procedure. If you want
      to print document in Tex please contact to shimojo@mars.ics.osaka-u.

   3. X, gnu emacs, wnn is available. Try login by guest. If you copy .*
      file from /usr1/guest, you can use it.

   !!! CAUTION !!!

   1. You should use YPPASSWD instead of PASSWD to update your password.

(You can contact S.shimojo (talk shimojo@saturn or
mail shimojo@mars.ics.osaka-u) if you have any questions.)

ccsun03% lisp

Sorry, lisp is not configured to run on this CPU.

ccsun03% rlogin ccsun01

Last login: Fri Jun 23 16:41:13 from ccsun03
Sun UNIX 4.2 Release 3.4EXPORT (GENERIC) #1: Thu Apr 30 09:36:18 PDT 1987

          ************************************************
          ****  Computation Center, Osaka University  ****
          ************************************************

  *** News from Center (1989.05.26) ***

   1. Telnet is now available for ACOS-2000. Host name of ACOS-2000 is
      acos. ftp and telnet is available.

   2. Japanese TeX and Latex is now available in /usr/local/bin. But
      printing document is little bit complex procedure. If you want
      to print document in Tex please contact to shimojo@mars.ics.osaka-u.

   3. X, gnu emacs, wnn is available. Try login by guest. If you copy .*
      file from /usr1/guest, you can use it.

   !!! CAUTION !!!

   1. You should use YPPASSWD instead of PASSWD to update your password.

(You can contact S.shimojo (talk shimojo@saturn or
mail shimojo@mars.ics.osaka-u) if you have any questions.)

ccsun01% lisp
lisp: Command not found.
ccsun01% echo 'set path=($path /mnt2/lisp)' >> .login
ccsun01% source .login

ccsun01% lisp
;;; Sun Common Lisp, Development Environment 2.1.1, 19-Sep-88
;;;
;;; Copyright (c) 1987 by Sun Microsystems, Inc.  All Rights Reserved
;;; Copyright (c) 1985, 1986, 1987 by Lucid Inc.,  All Rights Reserved
;;;
;;; This software product contains confidential and trade secret
;;; information belonging to Sun Microsystems.  It may not be copied
;;; for any reason other than for archival and backup purposes.

> (quit)

ccsun01%
ccsun01%
```

ログインすると...

センターからのメッセージが表示される

UNIX のプロンプト

% の前はホストの名前

ccsun03 では使えない

ccsun01 にリモートログインする

再びメッセージがでる

検索パス不十分のため起動できない

エディターを使わずリダイレクトでファイルに書き加えることもできる

Common Lisp のメッセージ

(quit) で終了

図 1　Common Lisp の起動

3．Lisp に触れよう

さて、Lisp はいわゆるインタープリターですから、入力したことの結果がすぐ出力されます。 入力が間違っていようが何だろうが、それなりの反応が返ってくるのです。これはちょうど BASIC のようなもの、とお考えになって良いと思いますが、BASIC と少し違うのは、Lisp は、BASIC や FORTRAN のような「ステートメント翻訳型」の言語ではなく「関数型」の言語である点です。

百聞は一見にしかず。 一度使ってみましょう。 図２のように、(+ 1 2) と入力します。 そうするとすぐその下に 3 と印字されるでしょう。 これは + という和を計算する関数を 1 と 2 の二つの引数を渡して呼び出したのです。 その結果の 3 が反応として返ってきたわけです。 同様に (* 2 3) は 6 を、(/ 12 3 2) は 2 を、(- 9 3 2 1) は 3 を返します。 賢明な読者諸氏はもうお気付きですね。 そうです。 これら四則演算の関数は二つ以上の引数を受け取ることが可能なのです。 このように、関数呼び出しの際の引数を自由に定義できることは Lisp の特徴の一つです。

さらに、次の面白い例をお見せしましょう。 (/ 12 2 3 4) と、四つの引数をもって商を計算する関数を呼び出します。 12 を 2*3*4=24 で割るから 0.5 かな？ とお思いになったあなた、すっかり「数値計算屋」になっていますね。 Lisp システムはそんなバカな答えはだしません。 1/2 と分数の答えを出してくれるのです。 このことの重要性は、先ほどの例で、四つ目の引数 4 を 3 にすることを考えると、おわかり頂けると思います。 12 を 18 で割ると 0.66666667 ではなく、2/3 なのです。

その他、Common Lisp には Lisp らしい数値計算の関数が多数、組み込み関数として定義されています。 詳しくは Common Lisp のマニュアルを御覧下さい。

```
> (+ 1 2) ……… + という関数を呼び出す
3 ……… 答えが反応として表示される
> (* 2 3) ……… 同様に．．．
6 ……… ２＊３ は ６
> (+ 1 2 3) ……… 引数は３っつでもよい
6 ……… １＋２＋３ は ６
> (/ 12 3 2)
2 ……… １２／３／２ は ２
> (- 9 3 2 1)
3 ……… ９－３－２－１ は ３
> (* 2 (+ 1 2)) ……… こんな風に入れ子にもできる
6 ……… ２＊（１＋２） は ６
> (/ 12 2 3 4) ……… １２／２４ は ０．５ ではなく
1/2 ……… １／２
> (/ 12 2 3 3) ……… １２／１８ は ０．６６６・・・ ではなく
2/3 ……… ２／３
> (quit)
ccsun01%
```

図２　Lisp による数値計算

どんな言語でも、プログラムを組む際には「変数」なるものを多用しますが、Lisp で「変数」を使う場合、それなりの覚悟が必要です。

Lisp で普通に変数の使用を宣言すると、それは、どこでも参照可能な「大域変数」となります。 もちろん、プログラムの構造化のために、変数の局所的な使い方もできますが、それには特別な「宣言」が必要です。 これについては後述します。 普通の大域的に通用する「変数」は、特に宣言することなしに使うことができます。 このことは逆にいうと、変数名を間違えて代入してもエラーにならない、ということですから、注意しなければなりません。 さらに、「値を持たない変数の参照」も注意が必要です。 プログラム内で、ある変数を参照したとき、その時点で過去になんら値を代入されたことのない、まったく新しい変数であれば、致命的なエラーとなります。 Lisp では「変数」はなるべく使わないようにしましょう。 プログラム内で局所的に使う変数ならば、問題は少ないので、どうしても必要なときは、後述の「局所変数」を使用しましょう。

値を変数へ「代入」するために、BASIC や FORTRAN では = という演算子を使いますが、Lisp では setq という関数を使います。 「代入」するのに「関数」を使うというと奇異に感じられるかも知れませんが、Lisp には演算子なるものがありません。 先程の四則演算でもわかるように、すべて「関数」なのです。

```
> (setq a 123) ------------------------------------ aという変数に１２３という整数の値を代入する
123
> a ------------------------------------------------ 変数aの持つ値を調べるには単に a と入力する
123
> (setq b a) --------------------------------------- 変数aの持つ値をそのままbに代入する
123
> b
123
> (setq c (quote a))------------------------------- 変数cにaという変数の名前をセットする
A
> c ------------------------------------------------ 変数cの値はAという一つの変数の名前
A
> (set c 456)-------------------------------------- set という関数を使うと
456
> c ------------------------------------------------ 変数cの持つ値を変数の名前として、
A
> a ------------------------------------------------ 間接的にaに値を代入することができる
456
```

図３　Lisp の変数

Lisp でよく使う「関数」に quote というのがあります。 quote という関数は、引数を一つ取り、何もしないで引数をそのまま返します。 「何もしない」ということが非常に重要で、かつ、有益なのです。

例えば、図３のように、(setq a 123) と入力して、変数 a に 123 という整数の値をセットしました。 続いて、変数 b に変数 a の持つ値を代入したいとき、(setq b a) とすればよいことは、すぐおわかりになるでしょう。 では、変数 c に a という変数の名

前を値として代入したいときは、どうしたらよいのでしょうか？ (setq c a) としたのでは c の値は 123 という変数 a の持つ値になってしまいます。 このようなことは、なんらかの計算をして、その答えを与えられた変数に代入したいときに必要になります。 そういうとき、変数の評価をマスクするために、quote があるのです。

(setq c (quote a)) とすれば、変数 a はその値 123 に評価されず、a という一つの「変数の名前」を c という変数にセットすることができます。 quote はあまりに多用するため、シングルクオーテーション ' にマクロ定義されています。

4．Lisp で創ろう

Common Lisp のシステムが提供している組み込み関数だけでも、覚えきれないほどたくさんあって、使っていて飽きることがありませんが、実用レベルのプログラムとなると、それらを複数組み合わせて作り上げることになることは言うまでもありません。 もう少し実践的な Lisp のプログラムを、もとい、関数を作ってみましょう。

月並みの例題で恐縮ですが、「階乗」を計算する関数を作ってみましょう。 図3に示すように factorial-a と名付けた関数を定義します。 関数を定義するには defun という関数を使います。 defun は一度行えば (quit) と打ってシステムを終了するまで有効です。 同じ名前の関数を defun すれば、前に defun した古い定義が捨てられ、一番新しい定義にすげ換えられます。 defun は "define function (関数を定義する)" の略称で、マニュアルでは「関数」とは呼ばず「特殊形式」と呼んでいますが、これも一種の関数と考えることができます。

factorial-a の定義では、カッコ ( ) で囲まれた argument は、それがこの関数の仮引数であることを意味します。 let もまた関数の一種で、局所変数 answer の使用を宣言すると同時に、それを初期値 1 にセットします。 局所変数 answer の有効範囲は、関数 let を囲む丸カッコの中だけです。 dotimes は反復処理のための関数で、ここでは局所変数 number が 0 から始まって 1 ずつ増えて次の実行部を繰り返し、 argument に達したらループを終え、その時の answer の値を「答え」とするように定義されています。 「答え」とは「この関数が返す値」ということです。 反復処理の実行部は setq を使って answer の値を (* answer (+ number 1)) にセットします。 つまり answer に answer*(number+1) を代入する、ということです。 ループカウンターの number は 0 から始まって argument-1 までであることに注意して下さい。 これは dotimes の定義がそうなっているからで、他意はございません。

このように定義された関数 factorial-a は (factorial-a 数) と入力することで、数の階乗を計算します。 (factorial-a 5) は 120 を、(factorial-a 10) は 3628800 を、「答え」として返します。 ここで注目すべきことは (factorial-a 100) と 100 の階乗を計算させると、ちゃんと158桁フルデジットの答えを示してくれることです。 たとえ数が 1000 でも 10000 でも、時間はかかりますが、ちゃんと真面目に計算してくれます。

次に、階乗計算の関数のもう一つの定義をお見せします。 factorial-b と名付けた関数は、factorial-a と同様、一つの引数を取りその階乗を計算しますが、定義が随分す

っきりしています。　局所変数を一つも使っていません。　センスの問題ですが、こういうのが Lisp らしいプログラム、と言えるのかもしれません。

factorial-b では、まず引数 argument が 1 以下の数かどうか、if という関数を使ってチェックします。　1 以下であれば「答え」を 1 とします。　1 より大きければ、argument に argument-1 の階乗を掛けた物を「答え」とします。　自分自身を再帰的に呼び出すように定義されているのです。

このように、Lisp のプログラム（＝関数）は全て、関数を演繹的に、あるいは再帰的に呼び出して定義されます。　関数は関数の入れ子になっているのです。

```
> (defun factorial-a (argument) ------------------------------------------ factorial-a を defunする
    (let ((answer 1))
         (dotimes (number argument answer)
         (setq answer (* answer (+ number 1))))))                          定義した関数の名前が
FACTORIAL-A ------------------------------------------------------------- 反応として表示される
> (factorial-a 5)
120 ---------------------------------------------------------------------- ５の階上は１２０
> (factorial-a 10)
3628800 ----------------------------------------------------------------- １０の階上は３６２８８００
> (factorial-a 100)
93326215443944152681699238856266700490715968264381621468592963895
217599993229915608941463976156518286253697920827223758251185210 91
6864000000000000000000000000                                              １００の階上は　！！
>
(defun factorial-b (argument) ------------------------------------------ factorial-b を defunする
   (if (<= argument 1) 1
       (* argument (factorial-b (- argument 1)))))
FACTORIAL-B
> (factorial-b 5)                                                         以下同様
120
> (factorial-b 10)
3628800
> (factorial-b 100)
93326215443944152681699238856266700490715968264381621468592963895
217599993229915608941463976156518286253697920827223758251185210 91
6864000000000000000000000000
> (quit)
ccsun01%
```

図４　Lisp 関数の定義

５．Lisp を使いこなそう

さぁ、だいぶ Lisp に馴染めたことと思いますが、このままでは、Lisp ってなんて不便なんだろう、と思われそうなので、Common Lisp の便利な使い方をご説明します。

実際に何かの目的に Lisp を使うとなりますと、相当多くの関数を defun することになりますから、どうしてもソースコードをファイルに書いておいて、それを Lisp システムを起ち上げる都度読み込むことになります。　ソースコードは vi などのエディター

で書くとよいでしょう。 Lisp のソースは C言語などと同じようにフリーフォーマットですから、FORTRAN のように空白をいくつ一行が何文字、と気にする必要はありません。ただ、丸カッコの閉じ忘れにだけ注意してください。 丸カッコには Lisp を使い始めてからしばらくの間苦労させられますが、すぐに慣れるでしょう。

書き上げたソースファイルは、Common Lisp システム上に load という関数で読み込みます。 > のプロンプトの下で、(load "ファイル名") と打てばよいのです。 ファイル名 はダブルクォーテーション " " で囲ってください。 ファイル名 の拡張子は lisp とするのが一般的です。 必ずしも必要があるわけではありませんが、アプリケーションによっては問題が起こる場合があり得ます。 xxxx.lisp というような名前なら問題はまず起こりません。

defun された関数は、正しく動くことがわかれば、スピードアップのためコンパイルするとよいでしょう。 Common Lisp のコンパイラーは compile という関数で呼び出されます。 (compile '関数名) と 関数名 を quote して compile 関数に渡しますと、その関数の最新の定義はバイナリーコードにコンパイルされ、以後、その呼び出しがバイナリーで行われるようになり、実行速度が数倍速くなります。 ただし、quit してシステムを抜けてしまうと、そのバイナリーコードは失われてしまいます。 再びシステムを起ち上げたとき、もう一度テキストファイル(vi などで書いたファイル)を load して、コンパイルし直すことになります。 compile 関数で生成したバイナリーコードをファイルにセーブする方法は、標準の Common Lisp にはありませんが、compile-file という関数を使うと、ファイル単位でコンパイルして、バイナリーファイルを作成することができます。 図5の操作を御覧ください。 (compile-file "ファイル名") と入力すると、ファイル名 の拡張子が lbin になったものが生成されます。 これがバイナリーファイルで、普通のテキストファイル同様、load 関数で読み込むことができます。 このファイル単位のコンパイルは、lc という UNIX のコマンドを使って Lisp システムを起ち上げずに行うこともできます。

また、Common Lisp を使う上で、あると便利な関数をいくつか作っておきましょう。例えば、Common Lisp システムは、立ち上げるのには幾分時間がかかりますし、せっかく定義した関数なども失われてしまいますから、なるべく quit したくないものです。 しかし、vi などのエディターは UNIX に戻らなければ使えません。 Common Lisp システムを抜け出さずに UNIX のコマンドを使うにはどうしたらよいでしょうか? いちばん簡単な方法は X-window や suntools などのマルチウィンドーシステムを使うことですが、これらのツールはセンターにある高解像度のビットマップディスプレーでないと使えません。 そこで、私は次のような関数を定義しています。

```
(defun  unix  ()  (run-unix-program  "csh") )
```

これは Common Lisp に組み込みの run-unix-program という関数を使って UNIX のシェルを走らせてしまうものです。 この関数を定義しておくと > のプロンプトの下で、(unix) と入力するだけで、UNIX に戻ることが出来ます。 そして、ファイル操作やエディットをした後、CTRLキーと D を同時に押します。 すると、Common Lisp システムに戻ることができます。

```
ccsun01% cat factorial.lisp ···· cat というコマンドで
                                ファイルの内容を表示させる
(defun factorial-a (argument)
  (let ((answer 1))
       (dotimes (number argument answer)
                (setq answer (* answer (+ number 1))))))
                                テキスト形式で書かれている
(defun factorial-b (argument)
  (if (<= argument 1) 1
      (* argument (factorial-b (- argument 1)))))

ccsun01% lisp ···· Common Lisp を起動して
;;; Sun Common Lisp, Development Environment 2.1.1, 19-Sep-88
;;;
;;; Copyright (c) 1987 by Sun Microsystems, Inc.  All Rights Reserved
;;; Copyright (c) 1985, 1986, 1987 by Lucid Inc.,  All Rights Reserved
;;;
;;; This software product contains confidential and trade secret
;;; information belonging to Sun Microsystems.  It may not be copied
;;; for any reason other than for archival and backup purposes.
                                loadという関数でテキストファイ
> (load "factorial.lisp") ···· ルを読み込む
;;; Loading source file "factorial.lisp" ···· 若干のメッセージが表示される
#P"/usr1/x60630a/hattori/factorial.lisp"
                                これで factorial-a と
> (factorial-a 10)              factorial-b の二つの関数が
3628800                         定義された
> (factorial-b 20)
2432902008176640000             function という関数で関数の定
> (function factorial-a) ···· 義を見ることができる
#<Interpreted-Function (NAMED-LAMBDA FACTORIAL-A (ARGUMENT) (BLOCK FACTORIA
L-A (LET ((ANSWER 1)) (DOTIMES (NUMBER ARGUMENT ANSWER) (SETQ ANSWER (* ANS
WER (+ NUMBER 1))))))) 79D8D7>
> (compile 'factorial-a) ···· factorial-a をコンパイルする
;;; Compiling function FACTORIAL-A...assembling...emitting...done.
FACTORIAL-A
> #'factorial-a ···· #' は function を呼び出すよう
#<Compiled-Function FACTORIAL-A 7A3D9F>
                                マクロ定義されている
> (factorial-a 100) ···· コンパイルされているので速い！
93326215443944152681699238856266700490715968264381621468592963895217599993 2
2991560894146397615651828625369792082722375825118521091686400000000000000000
00000000
                                compile-file でファイル単位の
> (compile-file "factorial.lisp") ···· コンパイルを行なう
;;; Reading input file #P"/usr1/x60630a/hattori/factorial.lisp"
;;; Compiling function FACTORIAL-A...assembling...emitting...done.
;;; Compiling function FACTORIAL-B...tail merging...assembling...sharing li
st...emitting...done.
;;; Wrote output file #P"/usr1/x60630a/hattori/factorial.lbin"
#P"/usr1/x60630a/hattori/factorial.lbin"
> (quit)
                                ls というコマンドでファイルを
ccsun01% ls -l ···· リストアップすると
-rw-r--r--  1 x60630a      913 Jun 24 12:21 factorial.lbin ··· 拡張子が lbin になったものが
-rw-r--r--  1 x60630a      242 Jun 24 11:59 factorial.lisp    生成されている
ccsun01% □
```

図５　Common Lisp の便利な使い方

６．Lisp をもっと知ろう

これまで数値計算ばかりを扱ってきました。「リスト」というものを避けてきたのですが、「リスト」こそ Lisp の本質なのです。「リスト」とは Lisp プログラムでとられるデーター構造のことで、アルファベットや数字で構成されたシンボルを、いくつか数珠つなぎにしたものです。「配列」とは区別され「リスト」と呼ばれますが、基本的には同じようなものと考えてよいと思います。Lisp にはリストを操作するための関数が多数用意されているので、データーを配列にするよりリストにした方がずっと便利であるわけです。このように、データー構造はそのドメイン（領域、あるいは環境）によってかなり左右されることは皆さん御承知でしょう。では、「リスト」というデーター構造がいかに便利か、Common Lisp のもつリスト操作関数をいくつか御紹介しましょう。

リスト操作の基本は car、cdr、cons の三つの関数です。カー、クダー、コンスと発音します。この三つの組み合わせでいろいろな操作ができるのですが、Common Lisp では代表的な操作を組み込み関数として用意してくれています。

リストを作るには list という関数を使います。リストを継ぎ足すには append という関数を使います。これらリスト操作関数は、大別して二つのグループに分けることができます。引数を保存するものと、引数を破壊的に操作するものと、目的によって使い分けるべきです。

mapcar などのマッピング関数も大量のデーターの統計的数値計算に便利です。一連の数値データーをまとめて一つのリストにしておいて、mapcar という関数で各要素を一つずつ取り出してある関数にかける、ということもできます。

「連想リスト」あるいは「ａ－リスト」と呼ばれるリスト構造は、「リスト」には違いないのですが普通のリストとは違い、操作する関数も違います。これはデーターベースなど多量のデーターを分類管理するのに便利でしょう。

主なリスト操作関数の使用例を図６にお見せしますが、この他にもたくさんのリスト操作関数が Common Lisp に組み込みの関数として提供されていて、とても書ききれるものではありません。やはり、マニュアルの類なしでは Lisp プログラムの開発は無理でしょう。それら解説書をいくつか御紹介しておきます。

Common LISP　Guy L. Steele Jr. 著　後藤英一 監訳　井田昌之 訳　（共立出版社）

本書は Common Lisp 処理系の開発に携わった著者が自らとりまとめた文法書です。Common Lisp の文法の定義を行なっているのであり、解説書ではないので、少々読みにくいかも知れませんが、プログラミングの際には是非とも必要になるでしょう。

Common Lisp プログラミング　Rodney A. Brooks 著　井田昌之 訳　（丸善）

本書はスタンフォード大学における Lisp 入門コースの講義ノートから生まれた解説書です。Common Lisp を紹介し、効率よく美しいプログラムを書く方法を解説した本です。初めはこの本から入るのが最短距離でしょう

```
> (setq a 123)                                  aという変数に１２３という整数の値を代入する
123
> (setq b 456)                                  同様に bに４５６を
456
> (setq c 789)                                  cに７８９ を代入しておく
789
> (list a b c)                                  list という関数でリストを作る
(123 456 789)                                   (list a b c) は (123 456 789) というリストを作る
> (list 'a 'b 'c)                               (list 'a 'b 'c) と名々を quote したら、
(A B C)                                         (A B C) という変数の名前のリストができる
> (setq x (list a b c))                         これを変数にセットするには、setq を使う
(123 456 789)
> x
(123 456 789)                                   xという変数がリスト (123 456 789) にセットされた
> (reverse x)                                   reverse はリストをひっくり返す
(789 456 123)
> (car x)                                       car はリストの頭をとる
123
> (cdr x)                                       cdr は car の残り
(456 789)
> (append x (list 'a 'b))                       append を使ってxに (a b) というリストをつなぎ足す
(123 456 789 A B)
> (revappend x (list 'a 'b))                    revappend という関数もある
(789 456 123 A B)
> x                                             いずれも変数xは保存されている
(123 456 789)
> (rplaca x 321)                                rplaca はリストの car を置き換える
(321 456 789)
> x                                             ただし，破壊的である
(321 456 789)                                   変数xの内容が変わってしまった
> (nconc x (list 'a 'b))                        nconc は append と同じだが．．．
(321 456 789 A B)
> x                                             破壊的である
(321 456 789 A B)                               またしても変数xの内容が変わってしまった
> (nth 0 x)                                     nth はn＋１番目の要素を取り出す
321                                             ０番目
> (nth 1 x)
456                                             １番目
> (nth 3 x)
A                                               ３番目
> (list-length x)                               list-length はリストの長さをはかる
5
> (last x)                                      last は最後の要素だけを残したリスト
(B)
> (butlast x)                                   butlast はその逆 をつくる
(321 456 789 A)

> (setq z (list (car x) (last x) (nth 2 x) (list x x)))   ）もっと複雑なリストを作る
(321 (B) 789 ((321 456 789 A B) (321 456 789 A B)))
> (subst '0 '789 z)                             そのリストの中から要素 789 を探し
(321 (B) 0 ((321 456 0 A B) (321 456 0 A B)))      0 に置き換える

> (setq y '(-1 3 2 -5 7 -6 9 0 -4))             たくさんの数値データーを一つのリストにしておく
(-1 3 2 -5 7 -6 9 0 -4)
> (mapcar #'abs y)                              そのリストの各要素を一つずつ abs という関数にかける
(1 3 2 5 7 6 9 0 4)
> y
(-1 3 2 -5 7 -6 9 0 -4)                         mapcar は破壊的ではない
> (setq y (mapcar #'abs y))                     結果を残すには setq を使う
(1 3 2 5 7 6 9 0 4)
> (setq y (mapcar #'(lambda (n) (* n 2)) y))    それを各要素を２倍する
(2 6 4 10 14 12 18 0 8)
> (mapcar #'(lambda (n m) (+ n m)) y (cdr y))   さらに隣どうしを加える
(8 10 14 24 26 30 18 8)
```

図６　リスト操作の例

7．ART を使おう

このように柔軟なデーター構造をとりうる Lisp を用いていろいろなアプリケーションが開発されています。 センターには Common Lisp のアプリケーションとして ART というエキスパートシステム構築用のツールがあります。ART は米国 *Inference* という会社の製品で、ちょっと目玉が飛び出るほど高価なツールですが、そのパフォーマンスは多くの類似ツールの中で抜きんでています。 エキスパートシステムという一つのソフトウェア・アーキテクチャは人工知能（ＡＩ）の分野で最も実用的なシステムとして注目されていますが、その解説は他の文献にゆずるとして、ここでは ART の基本的な紹介だけにとどめたいと思います。

エキスパートシステムとは、故障・病気の診断や機械・構造物の設計など、経験豊かな専門家が行っている知的な作業をコンピューターでシミュレートするもの、つまり専門家の持つ経験則を集めコンピューター上で表現しルールベースとし、与えられたデーターとそれらのルールとから今すべきことを推論し、実行するシステムです。 ART はそういったシステムを構築するためのツールです。 実践なさった経験のある方はおわかりのことと思いますが、エキスパートシステムを構築しようとするとき、常につきまとう問題は

① 知識表現の方法
② グラフィック等の結果表示の方法
③ 開発時のトレース・デバッグの方法
④ 他言語とのインターフェース

の４点が挙げられます。 *Inference* 社はこれらの問題点を一気に解決しようと試みました。

ART は if-then 形式のプロダクションルールを基本とし、その前向き推論を相当のスピードでやってのけるだけでなく、後向き推論も可能です。 また、最近その必要性がささやかれている仮説推論、すなわち仮説を立てながら問題空間を探索し行き詰まった時点でその仮説全体を棄却し別の仮説を立てるというような、コンピューターらしい、しらみ潰し的問題解法も高速に実行することができます。 さらに、それらルールの参照する対象オブジェクトは、フレーム理論（Prof.Minsky 1974）に基づく構造化表現が可能です。 このような柔軟な知識表現方法によって上記①の問題に対処しています。

②は人工知能の研究レベルではあまり取り沙汰されませんが、実用的システムを考えるとやはりある程度の物は必要になります。 せっかく素晴らしい答えを導き出せても、その表示ができずただ記号・数字の羅列が出力されたのでは何のことだかさっぱりです。 ART は独自のグラッフィク・ルーチンを内蔵しており、マルチウィンドーを操り、高度に集積された情報をわかりやすく「絵」にする事が可能です。

エキスパートシステムに限らず、それなりに規模の大きなシステムを構築するとなると、開発の時間的効率が問題となります。 開発とはエディット・トレース・デバッグの繰り返しであることはいうまでもありません。 ART はその開発サイクルの手間を少しでも軽減するため、スクリーンエディターから推論エンシンのトレース、ルールベースやデーターベースのトレースなど、開発に必要な機能をすべて含んだひとまとまりの開発環境として提供しています。 さらに、開発を終え完成されたシステムは、もはやそういった

環境は必要ないのですから、ART から切り放し単独で稼働するアプリケーションシステムにチューンアップすることもできます。

構造物や機械の設計など、ある程度工学的に解析のできる分野では、数値計算による容量計算・構造計画・要目決定というような作業を行います。 その際問題になるのはそういった数値計算を行う既存のプログラムとの親和性です。 有限要素法などのプログラムはいちいち作らなくても既存の物があって、ユーザーはブラックボックスとして利用することが多いと思いますが、それらはたいてい FORTRAN で書かれています。 ART は C や FORTRAN などの既存のプログラムとの親和性をある程度持っています。 一番よいのは Lisp で、Common Lisp で書かれた言語ならソースレベルで深い親和性を持ち、いつでもどこでも利用可能です。 C や FORTRAN も全くないわけではないのですが、少し苦労することになると思います。 この点だけは ART の欠点です。

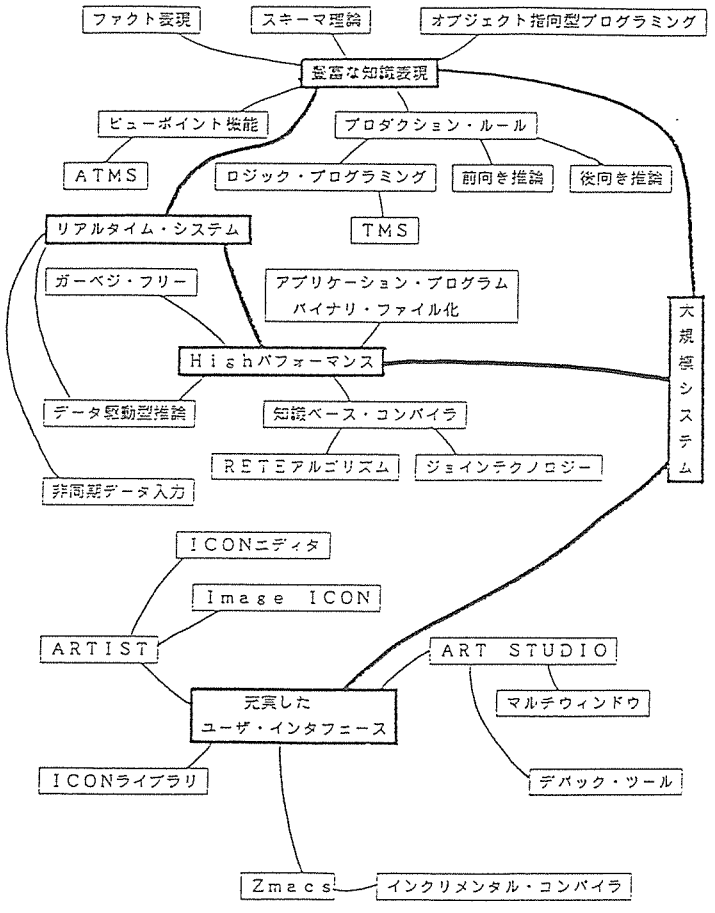

図7 ART 機能マップ

さて実際の ART の使い方ですが、その前にはやはり準備が必要です。 検索パスなど ART の環境を整えるため、いくつかのコマンドを実行しなければなりません。 それらコマンドは少々複雑ですし、使うたびに入力していたのでは面倒で仕方ありませんから art-world というファイルにまとめて書き込んであります。 このファイルの内容を使おうとするたびに評価してやればよいのです。 .cshrc というファイルの中に次の一文を加えて下さい。

```
source /mnt2/art3.1.1/art-world
```

art-world の前の /mnt2/art3.1.1 というのはこのファイルを含めて ART システム全てのファイルが格納されたサブディレクトリーです。 この後 Common Lisp の時と同じように、一度ログアウトするか、source .cshrc とします。 これで一応の前準備はできましたので、UNIX のプロンプト % の下で art と入力すれば ART が起動されます。

ART の開発環境には、グラフィックとスクロールの二つのモードがあります。 ART のグラフィック機能を使いたいときは、センターまで足を運んで ccsun01 のグラフィックディスプレーの前に座る必要があります。 そして、suntools と入力してマルチウィンドーシステムを起動して下さい。 その上で、ART を起動し最初の > のプロンプトに (graphics-studio) と応えて下さい。 すると図８のように、ART のグラフィックモードの環境が起動します。 あとはマウスを使っての操作となります。 マウスには三つのボタンがあって、それぞれにその時の状態に応じた役割がありますから、はじめは少しまごついてしまいますが、下の方にその時のマウスのボタンの意味が表示されています。 右・中・左を１回・２回、と全部で６通りの操作がマウスで行われます。

電話回線経由、あるいは ccsun01 以外からのリモートログインで使う場合、グラフィック機能は使えません。 ART 起動後の > のプロンプトに (scrolling-studio) と入力して下さい。 これを間違えてグラフィックモードを起ち上げようとすると大変なことになる可能性がありますので、十分ご注意下さい。 スクロールモードでは操作はすべてキーボードからの入力となります。 が、スペルコレクターが非常に親切な環境を作ってくれています。 コマンドを少々間違ったスペルで入力しても、正しく意図したとうりに訂正してくれるのです。 もちろん「これでいいのか」と聞いてくれます。

いずれのモードでも起動したあと ROOT というメニューが出ているはずです。 すべてオンラインヘルプの機能がありますから、わからなくなったところで help と入力すれば若干の手助けをしてくれます。 ROOT のメニューには examples というコマンドがあります。 名前の通り「例題」なのですが、examples とキーボードから入力するか、あるいはマウスでクリックするとさらに別のメニューが出てきて、どの例題にするか選択することができます。 このように、メニューの下にはサブメニューが入れ子になっています。 以下、大まかな操作の例を図で示します。

ART は Common Lisp の上で構築されたツールですが、独自の文法を持ち、柔軟な知識表現・高度なグラフィック操作などを実現しています。 それら ART 固有の文法についてはマニュアルを参照下さい。 "ART Tutorial" という解説書が４分冊と "Reference Manual" が２分冊、結構読みごたえのある本ですが、センター２階の第４端末室に備え付けてあります。 また、『ART 概説』という日本語の解説書もあります。

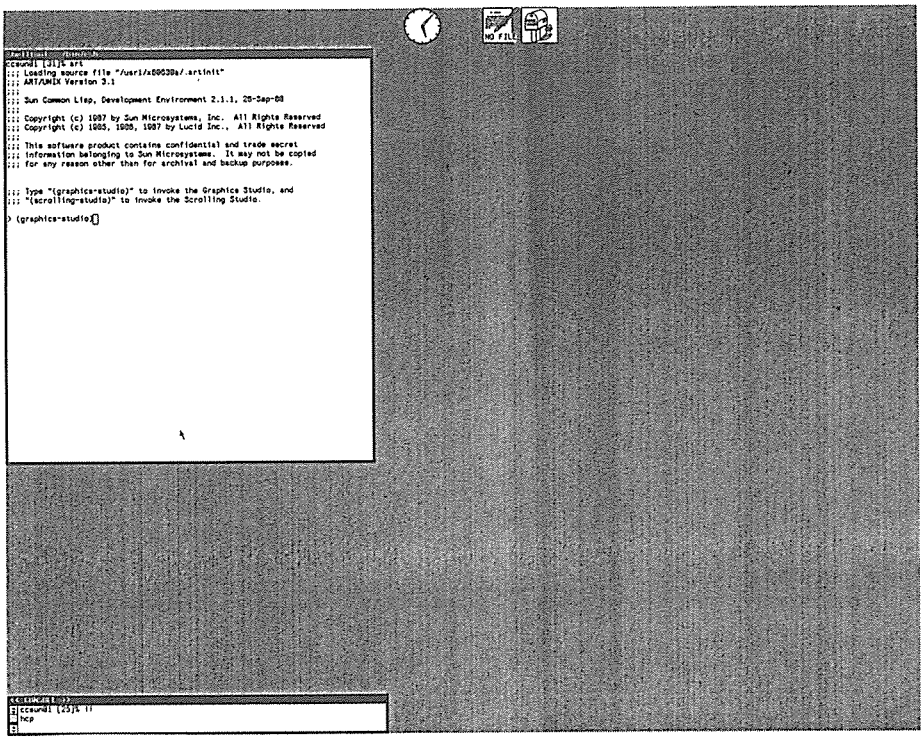

図８ａ　suntools 上で ART のグラフィックモードを起動する

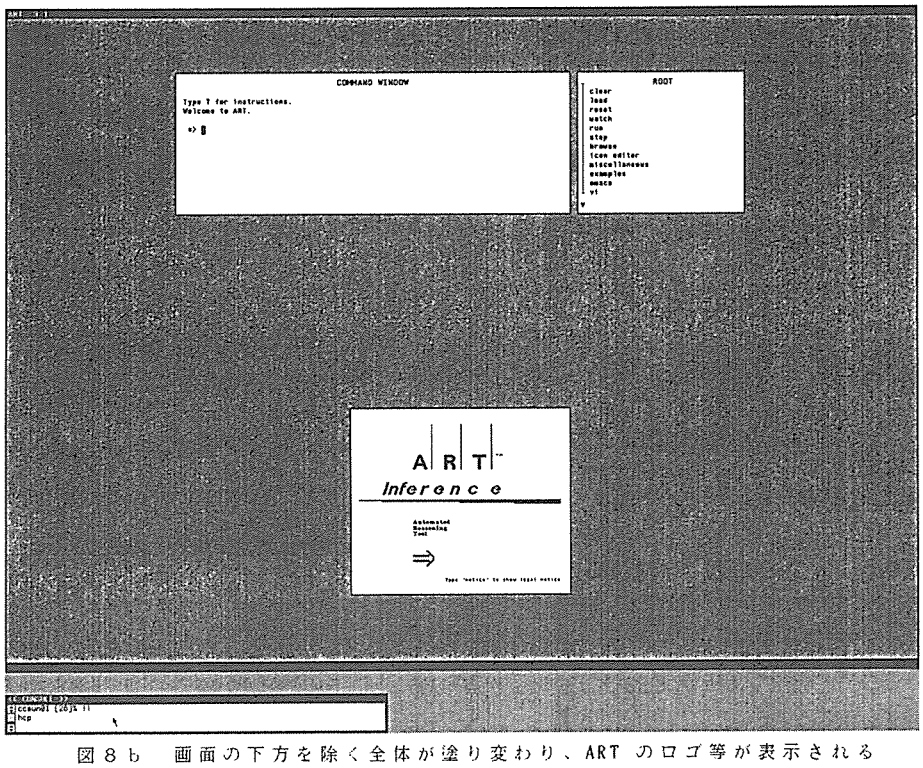

図８ｂ　画面の下方を除く全体が塗り変わり、ART のロゴ等が表示される

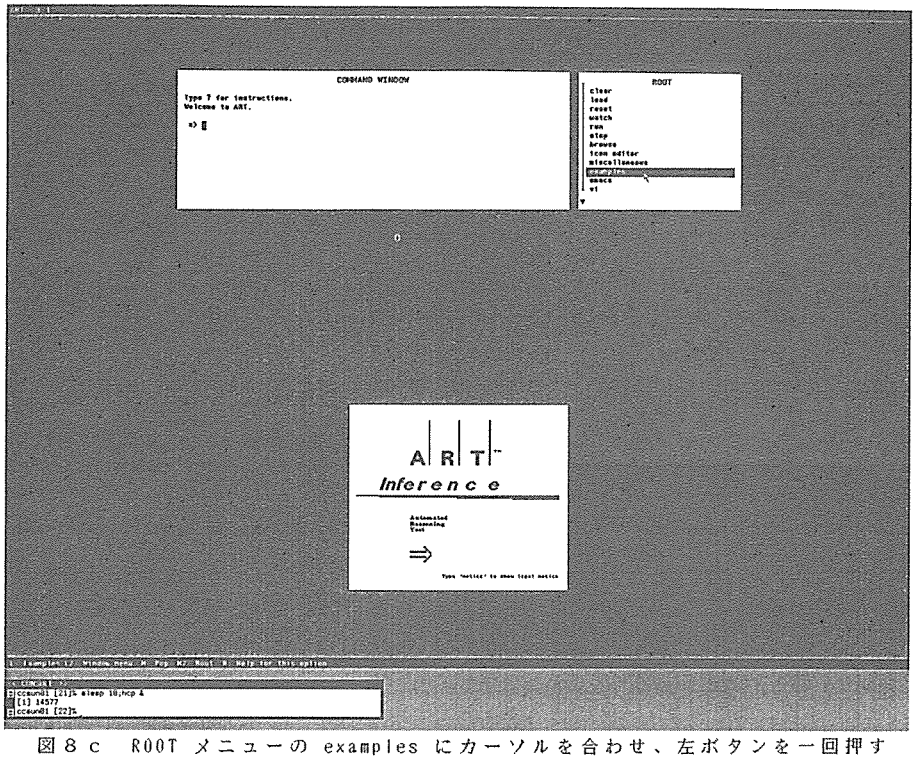

図８ｃ　ROOT メニューの examples にカーソルを合わせ、左ボタンを一回押す

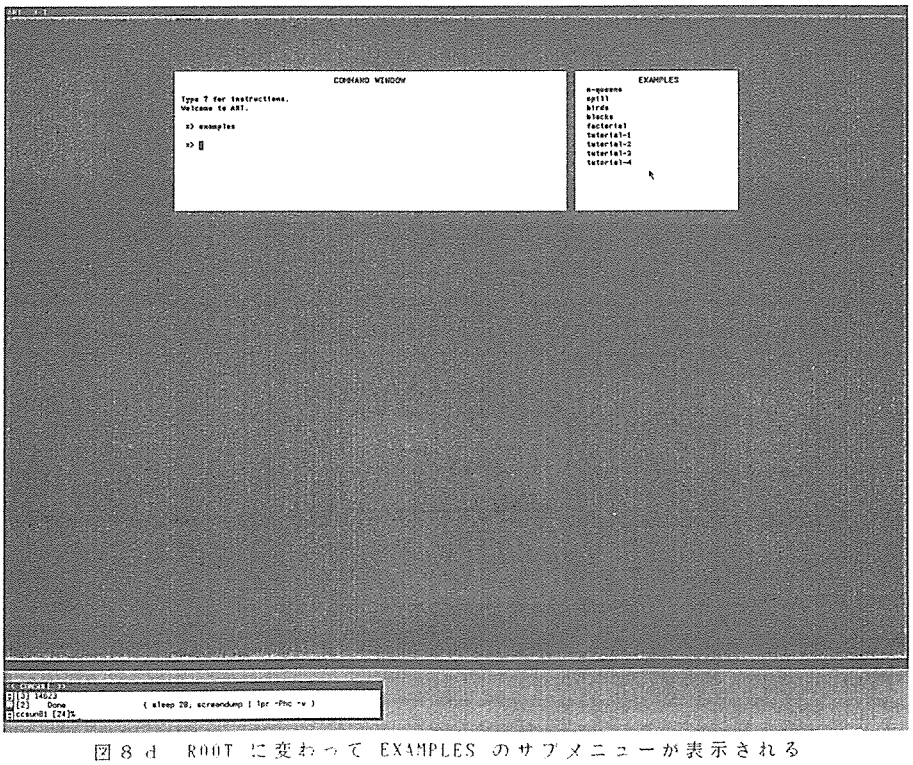

図８ｄ　ROOT に変わって EXAMPLES のサブメニューが表示される

```
ccsun01 [36]% art
;;; Loading source file "/usr1/x60630a/.artinit"
;;; ART/UNIX Version 3.1
;;;
;;; Sun Common Lisp, Development Environment 2.1.1, 26-Sep-88
;;;
;;; Copyright (c) 1987 by Sun Microsystems, Inc.  All Rights Reserved
;;; Copyright (c) 1985, 1986, 1987 by Lucid Inc.,  All Rights Reserved
;;;
;;; This software product contains confidential and trade secret
;;; information belonging to Sun Microsystems.  It may not be copied
;;; for any reason other than for archival and backup purposes.


;;; Type "(graphics-studio)" to invoke the Graphics Studio, and
;;; "(scrolling-studio)" to invoke the Scrolling Studio.

> (scrolling-studio)

                   The Automated Reasoning Tool

                           ART Studio
               Knowledge Base Development Environment

                          => INFERENCE

   Copyright (c) 1987, 1986, 1985, 1984 by Inference Corporation.
                        All Rights Reserved.

               Version 3.1 for SUN CommonLISP 2.1.1
                Type `notice' to show legal notice.



Press return to begin...

Type ? for instructions.


 ROOT

 clear
 load
 reset
 atch
```

図９　ART のスクロールモード

８．おわりに

Common Lisp とそのアプリケーション ART について御紹介したつもりですが、ほんのうわべだけをなぞったような中身のない文章になったようで、書き上げた後読み直して少し反省しています。　ただ、これを契機にワークステーションの利用者が増えることを願ってやみません。

最後にセンター研究室の村田先生、下條先生、後藤先生に、御迷惑をおかけしたことと私にこういう文章を書く機会を与えて下さったことに、謝意を表します。