A Lisp Environment at IBM TJ Watson Research

M. Mikelsons, C. N. Alberga and C. F. Skutt
Computer Science Department
IBM T. J. Watson Research Center,
Yorktown Heights, New York

Abstract: We give a brief description of a Common Lisp programming environment developed at the IBM TJ Watson Research Center over the past two years. The two most novel features of this environment are a program analysis tool and a visual stepper. These are described in more detail.

Overview

Over the past two years, we have built a prototype of a portable Common Lisp programming environment. The user interface is window oriented and based on the CLX interface to X Windows [5]. The code is currently running on the IBM RT PC under Lucid Common Lisp Release 2.

Our primary goal was to implement for the IBM RT PC an environment comparable to the leading current Lisp environments [3, 8] and to incorporate concepts developed in earlier systems [2]. We wanted both a tool for program development in Lisp, and also a foundation for future work in Lisp programming environments.

At the present time we have prototypes for:

- A data inspector that can display all built-in Lisp types, DEFSTRUCT instances, CLOS class instances, and a number of hidden types such as compiled functions, meta classes, and structure descriptors.
- A stack inspector that shows the stack as a scrollable window and pops up windows to display stack frames. Variable bindings in stack frames can be inspected with the data inspector.
- A program analyzer that extract call graph and special variable usage information from Lisp source files.
- An inspector for the analyzer database. This tool can show call graphs or variable def/use graphs. We have extended the analyzer and the display interface to show instances of structure and CLOS class slot accessors in an intelligent way.
- A defsys that goes considerably beyond the PCL defsys in several ways. In particular, it interacts with the analyzer to create an index of known symbols. This index is used to integrate the various separate tools.
- An interactive apropos tool for guided searches through the Lisp name space.
- A class browser that uses the same graph presentation tool as the analyzer.
- A source viewer that presents a program extracted from a file or from memory. The program is formatted and condensed to fit whatever window is available.
- A program stepper that uses the source viewer as the user interface.

All the tools are built on top of a PCL-based window support layer that interfaces to X-windows through CLX. Although we have not had time to do any speed hacking, the performance of the tools (except for startup) is acceptable.

In the remainder of this short note, we describe the two components that we feel contain some advances over the current state of similar tools. The program analysis and presentation methods attempt to provide more interesting information about a set of programs and try to provide tools for containing the information explosion inherent in cross reference tools. The stepper interface goes beyond teletype-oriented steppers by presenting the stepping information in a more readable and compact form.

The Program Analyzer and Graph Follower

The program analyzer collects control flow and data flow information from source files. The Graph Follower presents this information in interactive windows.

Control flow information consists of a call graph for an application. We treat CLOS methods as if they were called from their corresponding generic function.

The data flow information is more complex. It includes use and modify information for special variables. In addition, we treat structure and class accessors as data references. When an accessor is encountered in a program, we do not analyze the occurrence as a function call. Instead, we call it a reference or a modification of the corresponding slot in a specific data structure. As a result, the number of nodes in the call graph is significantly reduced and the use and modification of various data structures is clearly identified.

The Graph Follower attempts to contain and control the information explosion inherent in a program analysis tool. Instead of showing a mass of lines leading often to off-screen nodes, we limit each window to a focus node, a set of predecessor nodes, and and a set of successor modes. Simple mouse clicks are all that is needed to quickly scroll this view over any desired portion of the data graph.

When viewing the control flow information, each window is focused on a particular function. The callers of this focus function are shown on the left, while the callees are shown on the right. The user can open as many windows as desired in order to view an appropriate segment of the entire call graph. The first window is usually opened by selecting a focus function from a table of functions in the application. Additional windows are typically opened by selecting nodes in other windows.

When viewing data flow information, each window may be focussed on a data name or a function name. A data name may be the name of a special variable, the name of a structure, the name of a CLOS class, or the name of a slot in a structure or a class. When the focus is a function name, we show uses of data on the left and modifications on the right. When the focus is a data name, we show using functions on the right and modifying functions on the left.

The Visual Stepper

The Visual Stepper consists of a stepping evaluator and a program browser. The evaluator captures both the control flow and the environment of the evaluated expression in a data structure. This makes the evaluator well suited to a window environment. We chose an interpreter-based approach in contrast to annotations because of the greater flexibility in manipulating the control flow during stepping [4].

The program browser displays a condensed formatted view of the program in a given available area. It can highlight several sub-expressions simultaneously to point out various foci of attention [6].

When the program browser is showing a stepped expression, there are two distinguished sub-expressions in the window. Each is shown in a different font, or background/foreground combination. One sub-expression is the *Execution Pointer* that indicates where the stepper is currently positioned. The other sub-expression is the *Edit Pointer* that identifies some sub-expression of interest to the user.

The Execution Pointer must be further qualified with information displayed in a separate window above the program display. We need to specify whether the stepper is about to enter the Execution Pointer or whether that expression was just evaluated. We may also indicate that the stepper is somewhere within an expression but because of macro expansions, it is not possible to be more specific. When entering an expression there may be arguments and when leaving there are values. These are also displayed in one line of the information sub-window. If more detail is desired, the arguments or values can be inspected by pressing a mouse button on that line.

The Edit Pointer is used in a normal browser to move the focus of attention to various parts of an expression. When the Edit Pointer enters an elided sub-expression, the window is reformatted to show the hidden text. During stepping, the Edit Pointer serves another function; it is a convenient and powerful means of controlling the rate of stepping. By simply positioning the Edit Pointer and selecting a menu item, the user can step directly to some interesting part of the evaluation without viewing all the intermediate steps. Again, we have the choice of running until the Execution Pointer is about to enter the Edit Pointer or until the Execution Pointer is about to leave the Edit Pointer.

We allow a limited amount of modification to the normal order of evaluation. At any point, the user can bypass the evaluation of a sub-expression and provide a substitute value. This can be done even after the sub-expression
has been evaluated. We also allow the flow of control to be altered by positioning the Edit Pointer and asking for execution to continue at that point. Not all places in a program can be selected in this way. The execution Pointer can be repositioned to any place that could have been labelled in a TAGBODY form or to any expression in a PROGN that could be reached by the previous rule if the PROGN were a TAGBODY.

In all interactions between the user and the stepper, the visual representation of the program is the normal means of two-way communication.

Some Observations on Common Lisp

When we initially thought of building a state-based interpreter in and for Common Lisp, we expected difficulties that did not materialize [1]. The evaluator is written entirely in Common Lisp as recently extended by the standardization process [7] and allows arbitrary combinations of compiled code, stepped code, and natively interpreted code.

We have not tested the portability of our code yet, but the only uses of system-dependent code are in the access to internal data structures in the data inspector and to the run-time stack in the stack inspector.

Future Work

For the immediate future, our plan is to port our environment to the RS6000 workstation where the performance should be more than adequate for a production tool.

In more long range terms, we would like to refine the source-level stepper to deal intelligently with the effect of Lisp macros. In the current implementation, we simply open a new window to step through the code generated by a macro. A better approach would be to use this as a last resort, but in general to attempt to show the progress of evaluation in the original expression.

We must also extend the source code analyzer to provide more detailed information about an application. This includes tracking of taglist labels, catch points, non-local gos, types and constants. Another use for the analyzer could be to generate the inter-file dependencies that are normally expressed in a defsystem.

Acknowledgment

We would like to acknowledge the significant contribution of Scott Hauck who worked with us for six months while a Senior at UC Berkeley. He is now a graduate student at U. of Washington, Seattle.

References

1. Alberga, C. A., Bosman-Clarke, C., Mikelsons, M., and Van Deusen, M. Experience with an Uncommon Lisp. *Proceedings of the Lisp and Functional Programming Conference.*, August 1986.
2. Alberga, C. N., Brown, A. L., Leeman, G. B. Jr., Mikelsons, M., and Wegman, M. N.,. A Program Development Tool. *Eighth Annual ACM Symposium on Principles of Programming Languages*, January 1981.
3. Endelman, Aaron and Gadol, Steve. The Symboloc Programming Environment. *Lisp Pointers - Special Interest Publication on Lisp*, 2(2), 1988.
4. Haulsen, Ivo and Sodan, Angela. UnicStep - a Visual Stepper for Common Lisp. *Lisp Pointers - ACM SIGPLAN Special Interest Publication on Lisp*, 3(1), 1990.
5. Kimbrough, Kerry. Windows to the Future. *Lisp Pointers - Special Interest Publication on Lisp*, 1(4), 1987.
6. Mikelsons, M. Prettyprinting in an Interactive Programming Environment. *ACM Symposium on Text Manipulation*, Portland, Oregon,, June 1981.
7. Steele, Guy L., Jr. *Common LISP: The Language, Second Edition.* Digital Press, 1990.

8. Walker, Janet H., Moon, David A., Weinreb, Daniel L., and McMahon, Mike. The Symbolics Genera Programming Environment. *IEEE Software*, November 1987.

Appendix: The User Interface

We include two figures on the following pages to illustrate some salient features of our user interface.

Figure 1: Defsys and Stepper Interface

Window #1 is a control panel for the environment. It contains menu items that activate a variety of tools. It also shows the value of important variables such as *PACKAGE*.

Window #2 opens when the "Defsys" menu item on the control panel is selected. It shows a list of known sub-systems.

Window #3 opens when the system named "EPITOME" is selected in #2. It shows some general information about the sub-system as well as a list of modules.

Window #4 opens when the module "MAKEBOX" is selected in #3. It shows some general information about the module as well as a list of all the symbols defined in the module. For each symbol, we see the type of definition, the symbol, the lambda list and any documentation strings.

Window #5 opens when we select the definition of symbol LIST-TO-BOX. It shows the definition extracted from the source file. Sub-expressions are selected by pointing with the mouse or by using navigation commands on the menu bar. The mouse position is continuously echoed by drawing a box around the corresponding expression or by boxing a pair of matching parentheses. When a sub-expression is selected as the Edit Pointer, it is shown in a contrasting color.

Window #6 opens when we ask for a macro expansion of the Edit Pointer in #5. We do so by selecting from a sub-menu under "SubInspect".

Window #7 opens when we ask for a stepped evaluation of the Edit Pointer in #6, again by selecting from a sub-menu of "SubInspect".

Window #8 opens when the LET expression is expanded during stepping. In a stepper window, the navigation menu items are replaced with stepper commands. Navigation commands are still available under the "EditPointer" button. An additional sub-window shows the status of the stepper and arguments or values produced by stepping. The highlighted sub-expression shows the Execution Pointer.

Figure 2: Program Analyzer Interface In this example, we have invoked the program analyzer for a small set of functions in a file.

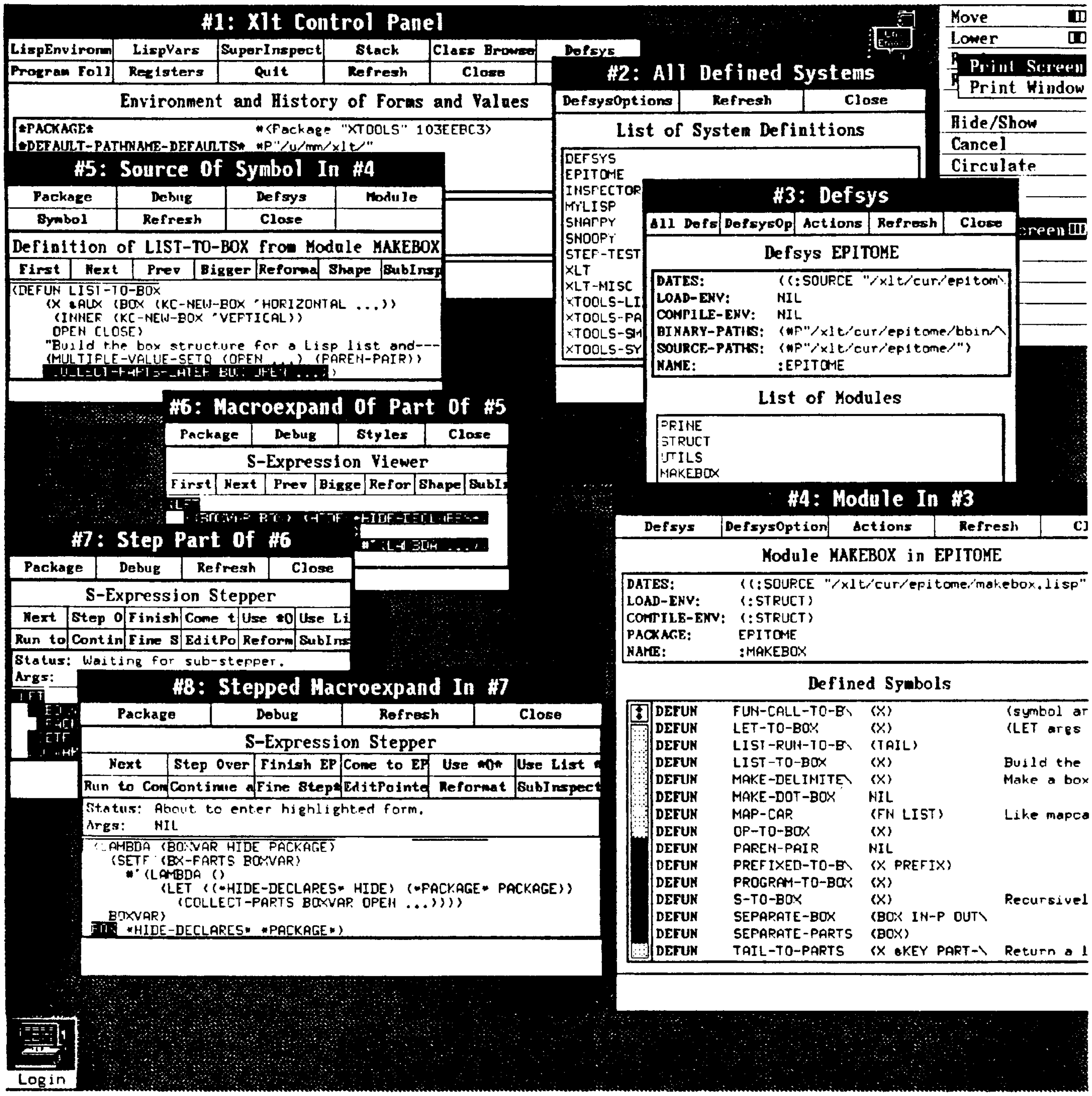
Window #2 is a table of all the special variables defined or used in the application.

Window #17 opens when the variable *COLOR-ARRAY* is selected in #2. It shows that the variable is input to the method INITIALIZE-INSTANCE for class DGNODE-DISPLAY. The boxes in the graph are always of a fixed size proportional to the size of the window, and therefore may show a truncated form of the variable or function name. The full name of a node in the graph appears in the window above the graph as the mouse passes through a node.

Window #3 is a table of all the functions defined in the application.

Window #4 opens when we select the method INSTALL for class DGNODE-DISPLAY. It shows that the method is invoked by way of the generic function INSTALL. It also shows that the method calls 5 functions.

Window #5 opens when we select the node UNHIGHLIGHT-CELL in #4 and ask for a new control graph window. this window shows that the function has three callers and calls four other functions.



Environment and History of Forms and Values

```
*PACKAGE*                        #<Package "XTOOLS" 103EEBC3>
*DEFAULT-PATHNAME-DEFAULTS*  #P"/u/mm/xlt/"
```

List of System Definitions

```
DEFSYS
EPITOME
INSPECTOR
MYLISP
SNAPPY
SNOOPY
STEP-TEST
XLT
XLT-MISC
XTOOLS-LI
XTOOLS-PA
XTOOLS-SM
XTOOLS-SY
```

Defsys EPITOME

```
DATES:          ((:SOURCE "/xlt/cur/epitom\
LOAD-ENV:       NIL
COMPILE-ENV:    NIL
BINARY-PATHS:   (#P"/xlt/cur/epitome/bbin/\
SOURCE-PATHS:   (#P"/xlt/cur/epitome/")
NAME:           :EPITOME
```

List of Modules

```
PRINE
STRUCT
UTILS
MAKEBOX
```

Definition of LIST-TO-BOX from Module MAKEBOX

```
(DEFUN LIST-TO-BOX
   (X &AUX (BOX (KC-NEW-BOX 'HORIZONTAL ...))
    (INNER (KC-NEW-BOX 'VERTICAL))
    OPEN CLOSE)
   "Build the box structure for a Lisp list and---
   (MULTIPLE-VALUE-SETQ (OPEN ...) (PAREN-PAIR))
   [illegible])
```

Module MAKEBOX in EPITOME

```
DATES:        ((:SOURCE "/xlt/cur/epitome/makebox.lisp"
LOAD-ENV:     (:STRUCT)
COMPILE-ENV:  (:STRUCT)
PACKAGE:      EPITOME
NAME:         :MAKEBOX
```

Defined Symbols

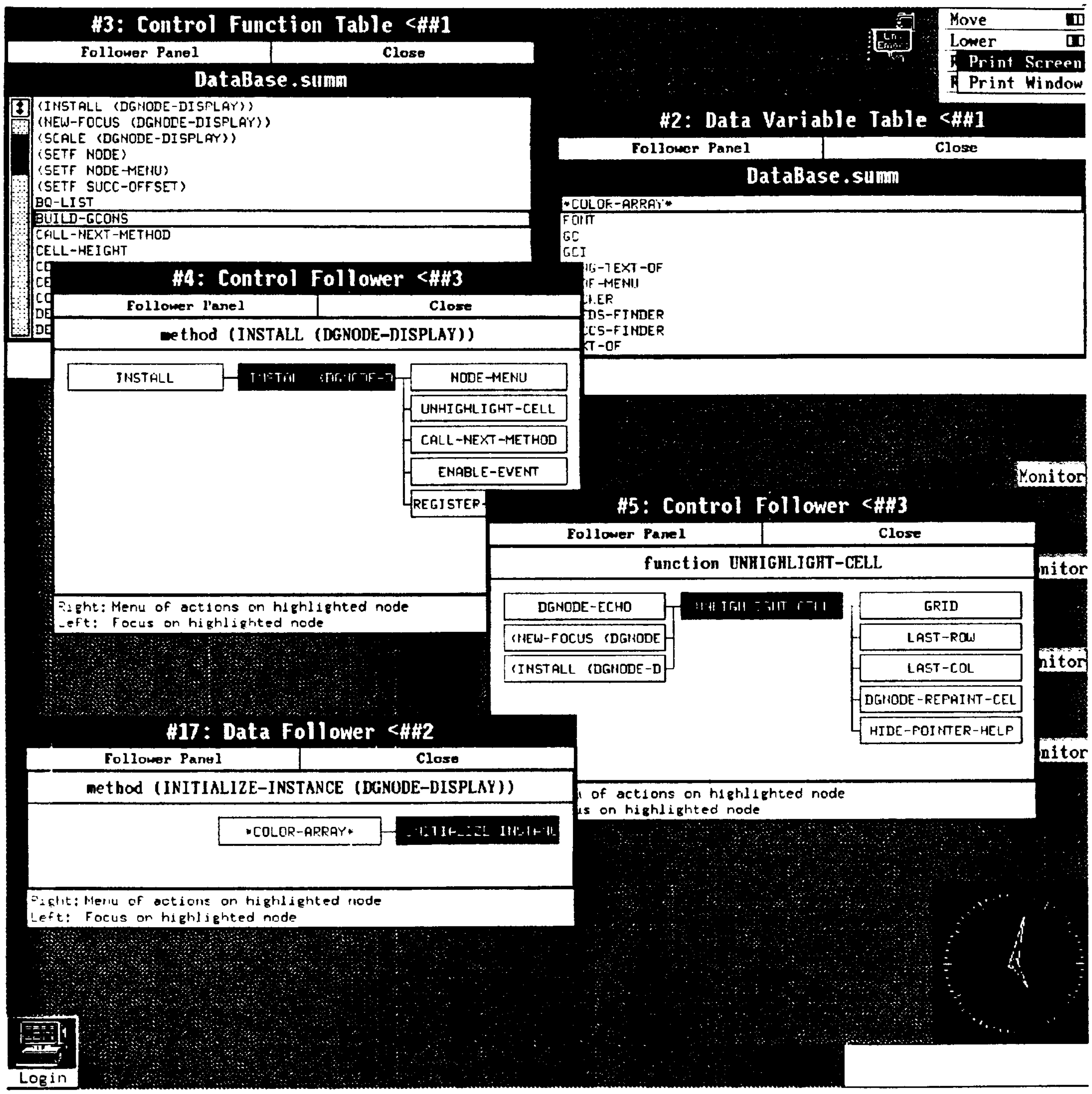
```
DEFUN   FUN-CALL-TO-B\   (X)               (symbol ar
DEFUN   LET-TO-BOX       (X)               (LET args
DEFUN   LIST-RUN-TO-B\   (TAIL)
DEFUN   LIST-TO-BOX      (X)               Build the
DEFUN   MAKE-DELIMITE\   (X)               Make a box
DEFUN   MAKE-DOT-BOX     NIL
DEFUN   MAP-CAR          (FN LIST)         Like mapca
DEFUN   OP-TO-BOX        (X)
DEFUN   PAREN-PAIR       NIL
DEFUN   PREFIXED-TO-B\   (X PREFIX)
DEFUN   PROGRAM-TO-BOX   (X)
DEFUN   S-TO-BOX         (X)               Recursivel
DEFUN   SEPARATE-BOX     (BOX IN-P OUT\
DEFUN   SEPARATE-PARTS   (BOX)
DEFUN   TAIL-TO-PARTS    (X &KEY PART-\    Return a l
```

```
Status: Waiting for sub-stepper.
Args:
```

```
Status: About to enter highlighted form.
Args:   NIL
```

```
(LAMBDA (BOXVAR HIDE PACKAGE)
  (SETF (BX-PARTS BOXVAR)
    #'(LAMBDA ()
        (LET ((*HIDE-DECLARES* HIDE) (*PACKAGE* PACKAGE))
          (COLLECT-PARTS BOXVAR OPEN ...))))
  BOXVAR)
[illegible] *HIDE-DECLARES* *PACKAGE*)
```

Figure 1: Defsys and Stepper Interface



```
#3: Control Function Table <##1
Follower Panel    Close
DataBase.summ
(INSTALL (DGNODE-DISPLAY))
(NEW-FOCUS (DGNODE-DISPLAY))
(SCALE (DGNODE-DISPLAY))
(SETF NODE)
(SETF NODE-MENU)
(SETF SUCC-OFFSET)
BQ-LIST
BUILD-GCONS
CALL-NEXT-METHOD
CELL-HEIGHT
CO
CE
CO
DE
DE
```

```
#2: Data Variable Table <##1
Follower Panel    Close
DataBase.summ
*COLOR-ARRAY*
FONT
GC
GCI
IG-TEXT-OF
IF-MENU
LER
DS-FINDER
CS-FINDER
T-OF
```

```
#4: Control Follower <##3
Follower Panel    Close
method (INSTALL (DGNODE-DISPLAY))
INSTALL — INSTAL (DGNODE-D — NODE-MENU
                           — UNHIGHLIGHT-CELL
                           — CALL-NEXT-METHOD
                           — ENABLE-EVENT
                           — REGISTER-
Right: Menu of actions on highlighted node
Left:  Focus on highlighted node
```

```
#5: Control Follower <##3
Follower Panel    Close
function UNHIGHLIGHT-CELL
DGNODE-ECHO          — UNHIGHLIGHT-CELL — GRID
(NEW-FOCUS (DGNODE   —                  — LAST-ROW
(INSTALL (DGNODE-D   —                  — LAST-COL
                                        — DGNODE-REPAINT-CEL
                                        — HIDE-POINTER-HELP
 of actions on highlighted node
s on highlighted node
```

```
#17: Data Follower <##2
Follower Panel    Close
method (INITIALIZE-INSTANCE (DGNODE-DISPLAY))
*COLOR-ARRAY* — INITIALIZE-INSTAN
Right: Menu of actions on highlighted node
Left:  Focus on highlighted node
```

Figure 2: Program Analyzer Interface