EMERALD 1: AN INTEGRATED SYSTEM OF MACHINE LEARNING AND DISCOVERY PROGRAMS FOR EDUCATION AND RESEARCH: PROGRAMMER'S GUIDE FOR THE VAXSTATION

by

K. Kaufman
R. S. Michalski

Reports of the Machine Learning and Inference Laboratory, MLI 90-14,
George Mason University, Fairfax, VA. Dec. 1990.

EMERALD 1:

An Integrated System of Machine Learning and Discovery Programs for Education and Research

Programmer's Guide for the DEC VaxStation

Kenneth A. Kaufman
Ryszard S. Michalski

MLI 90-14

EMERALD 1: An Integrated System of Machine Learning and Discovery Programs for Education and Research

PROGRAMMER'S GUIDE FOR THE DEC VAXSTATION

Kenneth A. Kaufman, Ryszard S. Michalski

Center for Artificial Intelligence
George Mason University
4400 University Drive
Fairfax, VA 22030

Abstract

EMERALD 1 is a large-scale system integrating several advanced programs exhibiting different forms of learning or discovery. The system is intended to support teaching and research in the area of machine learning. It enables a user to experiment with the individual programs, run them on various problems, and test the performance of the programs. The problems are defined by a user from a set of predefined visual objects, displayed through color graphics facilities. The current version of the system incorporates the following programs, each displaying the capacity for some simple form of learning or discovery:

AQ - learns general rules from examples of correct or incorrect decisions made by experts.

INDUCE - learns structural descriptions of groups of objects and determines important distinctions between the groups.

CLUSTER - creates meaningful categories and classifications of given objects, and formulates descriptions of created categories.

SPARC - predicts a possible continuation of a sequence of objects or events by discovering rules characterizing the sequence observed so far.

ABACUS - conducts experiments, collects data, formulates mathematical expressions characterizing the data, and discovers scientific laws.

Individual programs, presented as robots, communicate their results by natural language sentences displayed on the screen, and by voice. EMERALD 1 is an extension of ILLIAN, a smaller system developed for the exhibition "Robots and Beyond: The Age of Intelligent Machines". The exhibition was organized by a consortium of major US Museums of Science. Two versions of the system have been implemented, one for a DEC VaxStation (EMERALD/V), and another for a Sun workstation (EMERALD/S).

Acknowledgements

The EMERALD system integrates several machine learning and discovery programs that have been developed on the basis of research conducted by the research group of R.S. Michalski over the span of about fifteen years, first at the University of Illinois at Urbana-Champaign, and recently at George Mason University.

The support for the research that led to the development and continuing improvement of these programs was provided by a number of grants received over the years, specifically, by the National Science Foundation under grants No. MCS-74-03514, MCS-76-22940, MCS-79-06614., MCS-82-05166, MCS-83-06614, and NSF DCR 84-06801; by the Office of Naval Research under grants No. N00014-81-K-0186, N00014-82-K-0186, N00014-88-K-0186, N00014-88-K-0226, and N00014-88-K-0397; and by the Defense Advance Research Projects Agency under grants administered by the Office of Naval Research No. N00014-85-K0878 and N00014-87-K-0874.

The support for the development of ILLIAN, the initial smaller version of the system that was presented at the exhibit "Robots and Beyond: the Age of Intelligent Machines," was provided by the Digital Equipment Corporation, the Boston Museum of Science, and the University of Illinois at Urbana-Champaign. The development team of the initial version was led by R. S. Michalski, with assistance from Professors R. E. Stepp and D. Medin, and included A. Buriks, T. Channic, K. Chen, A. Gray, G. Greene, C. Kadie, K. Kaufman, H. Ko and P. Ong.

The EMERALD system was developed at the George Mason University Center for Artificial Intelligence under the direction of R. S. Michalski, in collaboration with K. DeJong, K. Kaufman, A. Schultz, P. Stefanski and J. Zhang. We particularly acknowledge A. Schultz, who was primarily responsible for the adaptation of the system to the SUN workstation, and the Naval Research Laboratory for supporting this effort. We also gratefully acknowledge the support of Professor D. Rine, Chairman of the GMU Department of Computer Science, and the help in testing and experimenting with the system from the members of the AI Center, including E. Bloedorn, J. Bala, K. Dontas, R. Hamakawa, J. Wnek and M. Wollowski.

(c) Copyright 1989 Artificial Intelligence Center
George Mason University

1. INTRODUCTION TO THE EMERALD SYSTEM

This report describes an integrated system of machine learning and discovery programs, called EMERALD, intended to demonstrate machine learning capabilities, and to serve as a tool for education and research in machine learning. To serve these needs, two systems have been developed:

ILLIAN - an initial, short version used specifically for demonstrating machine learning capabilities. This version was developed for the exhibition "Robots and Beyond: The Age of Intelligent Machines", organized by a consortium of eight Museums of Science (Boston, Philadelphia, Charlotte, Fort Worth, Los Angeles, St Paul, Chicago and Columbus.)

EMERALD (**E**xperimental **M**achine **E**xample-based **R**easoning **a**nd **L**earning **D**isciple) - an extended version for use as both an educational tool in machine learning and related areas, and as a laboratory for experimentation and research. Two versions of the system are currently available: EMERALD/V, which runs on the DEC VaxStation, and EMERALD/S, which runs on the Sun workstation.

The system is based on many years of research done by Michalski's research group in the area of machine learning. As this area has recently become very active, and many researchers have started to work in machine learning, we felt it would be of interest to the scientific community to integrate different programs into one system, and make it available for use in education and research. The integrated system makes it easy for a user to interact with and run individual programs, test them, and acquire experience in understanding their functions and their capabilities.

A complete, seamless integration of these programs is a very difficult task, requiring significant modification of the input and output modules of these programs and the development of a complex control mechanism. As the first step toward such a goal the programs were integrated at a user level, i.e., the system allows a user an easy access to each program through a menu and facilitates an application of each to problems defined by the user, employing various predefined objects.

The capabilities of the EMERALD include the ability to learn general concepts or decision rules from examples, create meaningful classifications of observations, predict sequences of objects, and discover unknown mathematical laws. A user may be surprised by some capabilities of the programs. Sometimes a user may do better than the machine, but sometimes it may be the machine

that does better.

The examples used in the demonstration deal with very simple objects -- pictures of imaginary robots or trains, geometrical figures, cards, etc. -- so that anyone can easily understand them. These learning programs, however, have already been applied to, and have a potential to be useful in many areas, such as medicine, agriculture, biology, chemistry, financial decision making, computer vision, database analysis, and, of course, intelligent robotics.

The current system, EMERALD 1, integrates the following five programs, presented as robots, each displaying a capability for some simple form of learning or discovery:

AQ -- learns general decision rules from examples of correct or incorrect decisions made by experts.

INDUCE -- learns descriptions of groups of objects and determines important distinctions between the groups.

CLUSTER -- creates meaningful categories and classifications of given objects.

SPARC -- predicts possible future objects in a sequence by discovering rules characterizing the sequence observed so far.

ABACUS -- formulates scientific laws and discovers mathematical patterns in data.

This report provides details of the functions used by EMERALD/S, in order to serve as a programmer's guide.

2. COMMON FUNCTIONS: THE SHARE PACKAGE

2.1. Introduction to the SHARE Package

The goal of the SHARE package is to provide users of VAXLISP access to external functions which are not conveniently implemented in LISP and to offer functions which will help to generate a standard environment for the demonstration of the different learning programs. Although some of the functions are hardware-dependent, the implemented graphics functions should be mostly portable, since they are implemented via an interface with the XWindow package.

This chapter will first describe the package for the user who desires to add interfaces with external functions to the SHARE package. A discussion of how key functions are typically used in the LISP code for the EMERALD exhibit and of the speed constraints which limit how they should be applied will follow. Finally, a list is provided of all the SHARE functions currently implemented and a description of the operations they perform. The last two sections will be most useful for a LISP programmer unfamiliar with C who needs to modify the existing EMERALD code.

2.2. Addition of Features to the SHARE Package

Although one can easily add features to the SHARE package by combining existing LISP code and adding this combination to the package, the following discussion will assume that it is necessary to utilize processes from outside the LISP environment. In addition to a description of the process of interfacing an external routine with LISP, two examples of such interfaces will be provided. The external code for the examples will be written in C, as is typical for the majority of the external code in the exhibit.

Two documents provided by DEC which will help when attempting to add an external routine are the following:

VAX LISP/ULTRIX System Access Programming Guide, Order Number: AA-EV40A-TE
and
ULTRIX-32w Xlib-C Language X Interface, Order Number: AA-HF10A-TN.

The first describes the details of interfacing LISP with external routines, while the second is necessary in order to obtain a complete understanding of how to interface C code with the XWindow utilities.

2.2.1. Outline of Procedure

The general process followed when adding an external routine can be described as follows:

1. Write and Compile the External Routine.
The compilation should be done with the -c option for C code. (Avoid passing structures into the external routine, as we have discovered in past experience that handling such interfaces is impractical, if not impossible.)

2. Run Tests on the External Routine for the purposes of verification.
These should be run in the external language. This is recommended because if tests are run from LISP, the entire environment must typically be reloaded before each test.

3. Write the LISP interface.

4. Enter the LISP environment and test the interface independently.

5. Load the share package, reload the interface and test the interface.

6. It is necessary to do some relocation of files to remain consistent with previous conventions:
Copy any C files to the directory .../share/c/filename.c
Copy any object files to the directory .../share/o/filename.o
Copy any LISP code to the directory .../share/lsp/filename.lsp.

From the shell, type the command shassemble which is an alias for a catp which takes all the lisp code in the "lsp" directory and creates the file .../share/share.lsp.

Finally, enter LISP, load and compile share.lsp, and retest the new routine. If no errors are encountered, a new function has been added to the SHARE package. The compiled version should be stored in .../share/share.fas.

2.2.2. Programming Examples

This section consists of 2 examples. The first is simple, and is provided to give an initial understanding of how parameters are passed between lisp and an external routine. It does not involve the XWindow utilities, thereby avoiding a number of complications. The second example is a graphics function taken directly from the EMERALD exhibit which provides more a realistic example of an interface with an external routine.

2.2.2.1. Example 1 -- Printing Lines of Text

This example is the output of a string of characters via a C routine. The string will be output on separate lines a number of times specified by the first parameter. Although this could also be done easily within LISP, it provides a simple first example.

The C code for the example appears as follows:

```
#include <stdio.h>

/*  Routine textout(x,str) prints the text in a given string  x times,
      each time on a separate line.   */

extern textout(x,str)
int *x;
char *str;

{int i;
if (*x>0)
for (i=0; i<*x; i++)
    printf("%s\n",str);
} /* END textout */
```

We observe in the above example that values are passed as pointers. We also observe that the C routine has been declared extern.

The LISP code which provides an interface to this routine is shown below:

```
;file: examp1.lsp
(export 'ExampleTextOut)

;;
;;(
Define-external-routine
        (TextOutDef
;; The following specifies the list of necessary files.
        :file                   ("/mnt/michalski/adrian/docshare/examp1.o")

 ;;The following specifies the name of the external function.
;; NOTE: Do not forget to insert the underline symbol
;; preceding the name.
                :entry-point
              "_textout"
    )
        (x  :lisp-type integer    :access :in-out)
            (str :lisp-type string    :access :in-out)
)
;;NOW DEFINE THE FUNCTION SEEN BY A USER OF THE SYSTEM
(DEFUN ExampleTextOut (NumberTimes String)
   (CALL-OUT TextOutDef NumberTimes String)
)
```

Observe that in above text, first the name "TextOutDef" used by the CALL-OUT facility is defined, followed by the definition of the LISP function "ExampleTextOut". This method has been used in

the share package to hide the CALL-OUT function from users of the package. Thus, users of the share utilities can treat them as if they are standard LISP functions. Thus documentation for share would only include a listing describing "ExampleTextOut", its parameters and use.

2.2.2.2. Example 2 -- The StringOut Function

This example illustrates how one of the functions in the share package is implemented. The function shown outputs a string to the screen in an arbitrary font. This function uses the XWindow routines and thus demonstrates how these can be used from C.

The C code for this routine is shown below:

```
#include <stdio.h>
#include <X/Xlib.h>

#define CR          13
#define NL          10
#define LINELENGTH    100
#define WORDLENGTH    100
#define SPACE       32

/*   Routine stringout() displays all the text in a given string and
     incorporates checks for spaces which are skipped over as some
     fonts print out a character for a space (i.e. cmr7)
*/
extern stringout (x, y, str, fontname, foreground, background, draw)
char *str;
char *fontname;
int *foreground, *background;
int *x, *y;
int *draw;      /* draw=0 =>don't output,  draw<>0 => output string*/

{
    int i, num_char_pre_sp, num_char;
     char word[WORDLENGTH];
    short fontheight,fontwidth;
        short widths[LINELENGTH];
        int xmax, ymax;
    FontInfo *font;
    Font fontid;
     WindowInfo *winfo;
        int ht,wd;
        wd = DisplayWidth()-1;
        ht = DisplayHeight()-1;

if (!(*x<0 || *y<0 || *x>wd || *y>ht))
```

```
{
    XRaiseWindow(RootWindow);
    if ((font = XOpenFont(fontname)) == NULL) return(-1);
    fontheight = font->height;
    fontid = XGetFont(fontname);

        XCharWidths(str,strlen(str),fontid,widths);
        xmax = 0;
        for (i=0 ; i < strlen(str) ; i++)
        {       if (str[i] == SPACE) fontwidth = widths[i];
                xmax += widths[i];
        }
        xmax += *x;
        ymax  = *y + fontheight;

    num_char = strlen(str);

    for (i=0, num_char_pre_sp=0;
        (word[num_char_pre_sp] = str[i]) != NL &&
          str[i] != CR  &&  i < num_char; i++)
        {
        if (word[num_char_pre_sp] == SPACE)
            {
            if (num_char_pre_sp > 0)
                {
                word[num_char_pre_sp] = NULL;
              if (*draw!=0)
                        XText(RootWindow, *x, *y, word,
                                        num_char_pre_sp,fontid,
                    *foreground, *background);
              *x += XQueryWidth(word,fontid) + fontwidth;
                num_char_pre_sp = 0;
                }
            else *x += fontwidth;
            }
        else ++num_char_pre_sp;
        }
    if (num_char_pre_sp > 0)
            {
            word[num_char_pre_sp] = NULL;
            if (*draw!=0)
                XText(RootWindow, *x, *y, word, num_char_pre_sp,
                                        fontid, *foreground, *background);
            }

    XCloseFont(font);
        XFlush();
        XLowerWindow(RootWindow);
        XFlush();
        *x = xmax;
        *y = ymax;
} /* stringout */
}
```

The Lisp code for this function is shown below:

```
;file: stringout.lsp(export 'StringOut)
;;
;; External Routine: stringout
;;       x and y will be updated to the maximum coordinates
;;       that the string occupies in the display.
;;
(eval `(Define-external-routine
        (stringout
       :file (,(Full-Path "share/o/stringout.o")
         "/usr/lib/libX.a")
                 :entry-point "_stringout")
                (x      :lisp-type integer      :access :in-out)
                (y      :lisp-type integer      :access :in-out)
                (string         :lisp-type string       :access :in-out)
                (fontname :lisp-type string     :access :in-out)
                (foregnd  :lisp-type integer    :access :in-out)
                (backgnd  :lisp-type integer    :access :in-out)
                (draw   :lisp-type integer)
))
;;
;; Function: (StringOut x y string [fontname [foreground [background]]])
;;       Outputs a string of text starting at (x,y).
;;       Returns the width and height of the string (line) in a list.
;;
(defun StringOut (x y string  &optional fontname foreground background
                              &aux  (x0 (+ x *x-org*))
                                    (y0 (+ y *y-org*))
                  )
          (if (not (stringp string))
    (list 0 0)                    ;;nothing drawn
    (progn
        (if (null fontname) (setq fontname (EchoFont)))
        (if (null foreground) (setq foreground (EchoForeground)))
        (if (null background) (setq background (EchoBackground)))
        (call-out stringout x0 y0 string fontname
                                  foreground background 1)
        (list (- x0 (+ x *x-org*)) (- y0 (+ y *y-org*)))
    )
  )
)
```

In the LISP code, the global variables *x-org* and *y-org* are modified in order to implement placement relative to a relocatable origin. In order to test the routine, it would be necessary to load share, then specify:

(in-package 'share)

and load the above code. Of course, this function is already included in the share package.

Also note in the above code the existence of a function **Full-Path**. A global variable, ***exhibit-path***, defines the complete path from the root directory to the exhibit directory, and Full-Path attaches that to the path of the desired file. This function greatly increases the portability of the EMERALD system. One must only redefine *exhibit-path* to move the system.

2.3. Use of the SHARE Package

Although the application of the routines in the SHARE package has not been limited to what is discussed below, the novice EMERALD programmer should find the programming tips given useful when trying to understand existing code. The goal of the following introduction is thus not to provide a complete description of all SHARE functions, but to give insights into how the most commonly used functions should be applied. After one has developed some simple menus using the introduction given in section 2.1, it is suggested that the new user examine existing code to observe how particular features beyond the scope of selecting icons and displaying text on the screen can be implemented. A description of all existing SHARE functions can be found in section 2.3.2, while a list of specific guidelines regarding their use can be found in section 2.3.3.

2.3.1. Introduction to Screen Creation Using SHARE

The most common use of the the functions in SHARE is the production of screens containing text, possibly a few simple figures, and a number of regions which can be selected using a mouse or track-ball. Therefore, a discussion of the functions necessary to produce such screens seems natural as a means of introducing the package. These functions will be introduced in the order they are typically used when creating such a menu.

The first operation which must be performed when creating a menu screen is the creation of the standard background. This can be done using the **ClearScreen** function. This clears the screen so that it is black with a light-colored work area in the middle. The size and color of the regions match the specifications of Section 2.3.1.2. and are thus the standard means of producing this background.

The next operation which must typically be performed is the output of text which will provide a visitor to the exhibit information about some facet of machine learning. Such text has been broken into four conceptual classes which were originally distinguished graphically by color. The original definition of these classes can be found in Section 2.3.1.2. The classes have the names "General", "Rule", "Command" and "Special". For each category, a function exists which produces text meeting the specifications given by the standards of Section 2.3.1.2. These functions have the names **CommandText**, **GeneralText**, **RuleText**, and **SpecialText**. Each routine returns a list containing the width and height of the rectangle of text produced so that such functions can be called in succession, each producing text beginning at the bottom of the previous

rectangle of text. Because it is frequently necessary to know the size of rectangular region of text in order to place graphic images relative to its boundaries, a function exists corresponding to each of the four text-generating functions which returns the width and height of the rectangle of text, but which produces no graphic output. Each such function has the name of the text-producing function it sizes followed by the suffix "Size". Thus we have **GeneralTextSize**, **CommandTextSize**, **RuleTextSize**, and **SpecialTextSize**.

In addition to the production of text, it is usually necessary to provide illustrations to clarify points discussed in the text. Although space can be left for these figures in through the use of blank lines in a file, it is suggested that space be provided explicitly for the diagrams through the use of the **GeneralBlock** function. This provides a blank region of the appropriate size and color which can then be used as a background on which illustrations can be placed. Again, this function returns the width and height of the region produced. In order to obtain the top or left basis coordinates of the main GeneralBlock with which text blocks except "Special" are aligned, the functions **GeneralBlockLeft** and **GeneralBlockTop** have been provided. For the GeneralBlock function, a "Size" function exists giving its width and height, appropriately called **GeneralBlockSize**.

In order to avoid the necessity of continuously forcing the user to change the colors or constants used in a menu to meet new specifications of the GeneralBlock, RuleBlock, etc., a number of functions have been provided which give the user access to these values directly. Each returns the value of the specified constant. These functions are:

CommandBlockColor
CommandCopyColor
GeneralBlockColor
GeneralCopyColor
RuleBlockColor
RuleCopyColor
SpecialBlockColor
SpecialCopyColor
GeneralBlockTop
GeneralBlockLeft
SelectBlockColor
SelectCopyColor

In each, "Copy" has been used as a synonym for "Text".

The next stage in the production of a screen is the placement of mouse-selectable icons on the screen. This can be viewed conceptually as the placement of a transparent overlay on the screen. The icons are all contained on this overlay. In order to create the overlay, one first creates a

graphical description of the icons using the functions in the SHARE package. Each icon is typically defined as if (0,0) were its upper left hand corner, and is drawn on a rectangular background created using the DrawRect or FillRect function. The overlay is then created using the function **MenuDraw** or **MenuDrawPlain**. These functions are passed three lists. These lists consist of a list of icon function names, a list of locations to which icons are to be translated, and the list of tokens which are returned when a particular icon is selected using the mouse. These menu functions return as their output a list known as a "Pick List" which must be stored in a variable for later use.

The final stage in the creation of a menu is the creation of an interface with the mouse. The function provided to effect this interface is the function **MenuMouseSelect**. The typical interface consists of a loop which first calls the function **MenuMouseSelect** with the "Pick List" produced by a menu function followed by a set of cases based upon the output of MenuMouseSelect. When the user depresses the select button, MenuMouseSelect returns either the coordinates of the cursor or one of the tokens specified when defining overlay. The coordinate pair is returned only if the select button was depressed while the cursor was not within the boundaries of an icon. In all other cases an action is typically taken based upon the token returned.

2.3.2. XWindow and Other I/O Routines Available in share.lsp

Beep
ChoiceText
CommandBlockColor
CommandTextSize
DrawCircle
DrawPoly
EchoFont
EchoGeneralBlockTop
EnterExhibit
ExitExhibit
FillCircle
FillRect
GeneralBlock
GeneralBlockTop
GeneralText
HelpScreen
MenuDrawPlain
MouseClickLoc
NiceStringOut
RuleBlockColor
RuleTextSize
SelectSquare
SetOrigin
SpecialBlock
StringOut
TitleOut
WaitTimeOut

BlankDisplay
ChoiceTextSize
CommandCopyColor
DisplayHeight
DrawCurPoly
DrawRect
EchoFontHeight
EchoLimits
ErrorClear
FileOut
FillCurPoly
Full-Path
GeneralBlockColor
GeneralBlockSize
GeneralTextSize
Menu
MenuMouseSelect
NiceCTextClear
OptionBox
RuleCopyColor
SelectBlockColor
SetBackground
SpecialBlockColor
SpecialText
StringSize
TitleClear
WaitUser

CallUnix
ClearScreen
CommandText
DisplayWidth
DrawLine
EchoBackground
EchoForeground
EchoOrigin
ErrorOut
FileSize
FillPoly
FullRect
GeneralBlockLeft
GeneralCopyColor
GetColor
MenuDraw
MenuPlain
NiceCTextOut
ResetLimits
RuleText
SelectCopyColor
SetForeground
SpecialCopyColor
SpecialTextSize
Talk
UpdateLimits

FUNCTIONS PRESENTLY AVAILABLE FROM THE SHARE.LSP PACKAGE:

NOTE: The first time one of these commands is executed after SHARE has been loaded, it will take several seconds, due to the fact that the 'C' graphics routines have to be loaded. Also, color indexes may change with each program run, thus all programs specifying color should execute the GetColor command to get each color index.

(Beep)
Beeps.

(BlankDisplay [color-index])
Clear screen with specified background color
Returns color index of the color of the display after clearing.
Differs from ClearScreen in that it will clear the entire display, not just the working area.

(CallUnix command-string)
Executes a unix command.
Eg: *Lisp>* (CallUnix "ls")
file1.lsp file2.lsp file3.lsp

(ChoiceText y textlist [fontname])
Outputs the specified text as a menu choice with its top at y
(relative to the origin set by SetOrigin).
Returns a list giving the width and height of the Text. (width height)
Defaults:
fontname - given by (EchoFont) (i.e. "helv12b")
Eg: *Lisp>* (ChoiceText 0
'("1. See an example of" "how SPARC works"))

NOTES:
1. You will probably want to replace all your OptionBox calls with ChoiceText.
2. It is suggested that menu choices be limited to ONE LINE.
3. Menu choices can be numbered: "1. See an...", "2. More adva...".
4. Leave an interval of y = *MenuChoiceBlockInterval* (a SHARE global) between each menu choice.
5. The x coordinates of the coordinate-list passed to the Menu function are ignored by ChoiceText. However, y-coordinates are affected by the SetOrigin function which is called by Menu. ChoiceText will therefore usually be called with y=0 and with the y-coordinate of the menu choice in the coordinate-list set and passed to Menu.

(ChoiceTextSize textlist [fontname])
Does everything ChoiceText does, except output the menu choice.

(ClearScreen [color-index])
Fills the screen with the specified color.
If the color-index is not specified, the default background color is used.
ClearScreen may eventually be modified to only clear the portion of the display not occupied by the standard options of Menu and the error messages.

This will eliminate the need to redraw the standard options every time a new menu is drawn. Note: ClearScreen does not change the default background color. In order to change this, use the SetBackground function.

(CommandBlockColor)
Returns the color which is presently defined as the background for Command Blocks (ie, the background behind text output by CommandText.)

(CommandCopyColor)
Returns the color which is presently defined as the color of Command Copy (ie, text output by the CommandText function)

(CommandText y textlist [fontname])
Outputs the specified text as a Rule block with its top at y.
Returns a list giving the width and height of the Command block produced.
Eg: *Lisp>* (CommandText 400
'("Please stop" "kicking the terminal."))

(CommandTextSize y textlist [fontname])
Returns a list giving the width and height of the Command block produced by CommandText when given the same inputs.
Eg: *Lisp>* (CommandTextSize 400
'("Please stop" "kicking the terminal."))

(DisplayHeight)
Returns the height of the display in pixels.

(DisplayWidth)
Returns the width of the display in pixels.

(DrawCircle color-index x y radius [thickness])
Draws the outline of a circle of specified radius with center at (x,y), drawn in the specified color, and being of the given thickness.
Default thickness = 1 pixel.
Eg: *Lisp>* (DrawCircle (GetColor "green") 100 100 50 2)
Lisp> (DrawCircle (GetColor "pink") 10 10 5)

(DrawCurPoly color vertex-list [thickness])
Same as DrawPoly except points are joined with slines (curves).

(DrawLine color-index x1 y1 x2 y2 [thickness])
Draws a line from (x1,y1) to (x2,y2) of specified color and the specified thickness.
Default thickness = 1 pixel.
Eg: *Lisp>* (DrawLine (GetColor "blue") 0 50 0 100 3)
Lisp> (DrawLine (GetColor "black") 0 0 100 100)

(DrawPoly color-index vertex-list [thickness])
Connects the points in a vertex list with straight line segments.
The vertex-list is specified as ((x1 y1) (x2 y2) (x3 y3) ... (xn yn))
where (xi yi) are the coordinates of the vertices of the polygon.
Default thickness = 1 pixel.
Eg: *Lisp>* (DrawPoly (GetColor "aquamarine") '((0 0) (0 100) (100 100)))

(DrawRect color-index x1 y1 x2 y2 [thickness])
Draws a rectangle with diagonal from (x1,y1) to (x2,y2) with lines of the specified color. Thickness defaults to 1.
Eg: *Lisp>* (DrawRect (GetColor "red") 0 50 50 0 3)

(EchoBackground)
Returns the color index of the current background color.

(EchoFont)
Returns the default font name (in a string).

(EchoFontHeight)
Returns the default font height.

(EchoForeground)
Returns the color index of the foreground color.
Defaults to black.

(EchoGeneralBlockTop)
Returns the y coordinate of the top of the standard general block.
Eg: *Lisp>* (EchoGeneralBlockTop)

(EchoLimits)
Returns a list containing two pairs of numbers. The first pair represents the coordinates of the upper left corner of the screen; the second pair represents the lower right.
SHARE functions will draw only if their x and y coordinates fall within these limits.
Eg: *Lisp>* (EchoLimits)
((5 5) (750 800))

(EchoOrigin)
Returns the coordinate of the origin (with respect to the display).
This is the origin for all Draw and Fill functions.
Eg: *Lisp>* (EchoOrigin)
(0 0)
If SetOrigin is not used, the default is (0 0).

(EnterExhibit)
Hides all but the main window from view. Windows can be returned by a call to ExitExhibit.
This should only be called from the top level controls of the exhibit.

(ErrorClear text [font])
Covers the message produced by ErrorOut with the background color. If no font is specified, the default fonts chosen by,ErrorClear are used.
Eg: *Lisp>* (ErrorClear '("Please correct" "your mistake"))

(ErrorOut text [font [foreground [background]]])
Prints the message text at a standard location on the screen. Text is a list of strings to be printed. If no font is specified, the font is chosen so that messages fit on the display.
Defaults:
font - Given by (EchoFont)
foreground - Given by (GetColor "Red")
background - Given by (EchoBackground)

NOTE: use as few lines as possible, generally one or two.
Eg: Lisp> (ErrorOut '("Please correct" "your mistake."))

(ExitExhibit)
Returns all windows to view. This is the inverse of EnterExhibit. It should only be called when exiting EMERALD and returning to LISP.

(FileOut x y filename [fontname [foreground [background]]])
Same as TextOutput, but the contents of the specified file is used as the string. The entire path of the file should be specified, using the Full-Path function (see below). Most text files are stored in txt subdirectories within the subdirectories for the individual modules. Fontnames can be any of those in the /usr/lib/Xfont directory. However, it is suggested that FileOut text be of one font, "helv12b", for a consistent appearance of text output.
The present maximum string length is 100 characters. Returns the list:
(width height)
Defaults:
fontname - Given by (EchoFont)
foreground - Given by (EchoForeground)
background - Given by (EchoBackground)
Eg: Lisp> (FileOut 100 100 (Full-Path "aq/txt/intro.txt")
"helv12b" (GetColor "white") (GetColor "black"))

(FileSize filename [fontname])
Returns a list containing the width and height a file will occupy when output using the font specified by fontname.
For example, the output might be the list (50 60)
NOTE: No output to the screen is produced.
The default fontname is given by (EchoFont).
Eg: *Lisp>* (FileSize (Full-Path "share/README.share"))

(FillCircle color-index x y radius)
Draws a circle of a specified radius with center at (x,y), filled with the specified color.
Eg: *Lisp>* (FillCircle (GetColor "green") 100 100 50)

(FillCurPoly color-index vertex-list)
Same as FillPoly except points are joined with curved lines.

(FillPoly color-index vertex-list)
Draws a polygon and fills the interior with the given color. The vertex-list takes the form: ((x1 y1) (x2 y2) (x3 y3) ... (xn yn)), where (xi yi) are the coordinates of the vertices of the polygon.
Eg: *Lisp>* (FillPoly (GetColor "orange") '((0 0) (0 100) (100 100)))

(FillRect color-index x1 y1 x2 y2)
Draws a rectangle with diagonal from (x1,y1) to (x2,y2) and fills it with the specified color.

(Full-Path path-suffix)
Appends the path suffix to the string representing the full path of the base directory in which EMERALD is stored.
NOTE: The definition of this function must be changed when moving the system to a new directory.
Eg: *Lisp>* (Full-Path "share/lsp/full-path.lsp")
"/mnt/michalski/demo/share/lsp/full-path.lsp"

(FullRect color-index x1 y1 x2 y2)

Draws a rectangle with diagonal from (x1,y1) to (x2,y2), filling it with the specified color. This function differs from FillRect in that it includes the outline of the rectangle. Thus, it is a call to FillRect with the upper left corner shifted upwards and to the left one pixel, and the lower right shifted downwards and to the right one pixel.
NOTE: FullRect should therefore not be used with x or y coordinates on the edge of the screen as the rectangle will not be drawn.

(GeneralBlock height [ytop])

Produces a general block with top at y-coordinate *ytop*, and height given by *height*.
Returns a list giving the width and height of the box.
Default:
ytop - Given by *menuGenBlockY0*.
Eg: *Lisp*> (GeneralBlock 49)

(GeneralBlockColor)

Returns the background color on which GeneralText is produced -- the color of the GeneralBlock)

(GeneralBlockLeft)

Returns the coordinate of the leftmost pixel of the GeneralBlock.

(GeneralBlockTop)

Returns the y coordinate of the top of the GeneralBlock. Used for positioning relative to the top of the general block.

(GeneralBlockSize height [ytop])

Returns the same list as GeneralBlock, but without producing any output to the screen.

(GeneralCopyColor)

Returns the default color of text in the GeneralBlock.

(GeneralText <filename | textlist> [ytop [MarginList [font [textcolor [background]]]]])

Given a list of strings or a filename, this outputs the text on a rectangle as standard copy. The text is produced on a rectangle with a left edge at x = 100 and a right edge at x = 800. The initial y coordinate of the rectangle is 25 below the top of the background region (ytop). The height is determined by the number of lines of text.
MarginList is a list of margins which specify the amount of space to leave around text. These margins are typically used to leave space for diagrams. They are specified in counterclockwise order around the text, starting from the top.
The top and left margins are set up exactly as specified. The bottom margin is set up to be one pixel smaller than the smallest multiple of 25 which provides at least BOTTOM pixels below the text.
No right margin is used by GeneralText.
The function returns a list containing the width and height of the box.
Defaults:
ytop - Given by *menuGenCopyY0*
MarginList - Given by '(*menuGenCopyTopMargin*
menuGenCopyLeftMargin
menuGenCopyBottomMargin)
Font - Given by (EchoFont)

Textcolor - *menuMediumGoldenRod*
Background - *menuViolet*
Eg: *Lisp*> (GeneralText '("This is a short" "example."))
Lisp> (GeneralText (Full-Path "induce/menu1.txt"))

(GeneralTextSize <filename | textlist> [BottomCopyMargin [ytop [MarginList [font [textcolor [background]]]]])

Returns the size of the General block which would be produced by the function GeneralText if given the same inputs.
NOTE: No screen output is produced.
Eg: *Lisp*> (GeneralTextSize '("This is a short" "example."))
Lisp> (GeneralTextSize (Full-Path "induce/menu1.txt"))

(GetColor string)

Returns a number which is the color index corresponding to the color described in string. It is important that GetColor is used to generate the color-index because different machines might use different indices.
A list of legal strings are found in /usr/lib/X11/rgb.txt
Eg: *Lisp*> (GetColor "Blue")
4

(HelpScreen package-name filename [font])

Clears display, outputs file and waits for user to finish reading before returning.
The parameter *package-name* can be one of:
"ABACUS", "AQ", "CLUSTER", "INDUCE", "SPARC", "EMERALD"
Capitalization of the package-names is necessary.
The parameter *filename* represents the pathname to the desired help file, using **Full-Path.**
It is strongly suggested that help files are put in txt subdirectories, as were described in the section on FileOut, such as in (Full-Path "aq/txt/...") for AQ.
The parameter *font* is an optional string specifying the font to be used (see StringOut).

(Menu display-list coord-list output-list package-name [title1 [title2]])

Menu acts like MenuPlain, except when given a list of choices to display, it will display all the choices plus the following options (referred to in the text as standard menu options):

a. Quit
b. Go to the Main-Menu
c. Go to the <package-name> menu
d. Help
e. Go to the previous screen
f. Go to the next screen

After the user selects an option, a return token is returned by Menu.

The inputs:

All the inputs except package-name, title1 and title2 are the same as those in MenuPlain.
package-name: The name of the package in use, eg, "INDUCE", "SPARC", etc. Once again, this must be a string in UPPER-CASE.
title1: This is the optional first line of the title of the menu. It is output in the top part of the screen, between y = 0 and y =149.
title2: This is the optional second line of the title of the menu. It is output between y = 0 and y = 149.
Title1 and title2 must be in the form of strings.
IMPORTANT: Options should always be kept out of this area of the screen.

The output:

The returned list will be one of the return tokens or one of the following: "QUIT", "MAINMENU", "SUBMENU", "HELP", "PREVIOUS", "NEXT" Notice that all letters in these strings are upper-case.

Eg: *Lisp>* (menu '(OneBox TwoBox ThreeBox FourBox)
(list '(0 350) '(0 395) '(0 440) '(0 485))
'("choice1" "choice2" "choice3" "choice4")
"AQ")

(MenuDraw display-list coord-list output-list package-name [title1 [title2]])

Functions like Menu except it returns a pick-list without asking the user for a mouse input. See MenuMouseSelect for an example of when this function might be desired.

(MenuDrawPlain display-list coord-list output-list)

Functions like MenuPlain except it returns a pick-list without asking the user for a mouse input. See MenuMouseSelect for an example of when this function might be desired.

(MenuMouseSelect pick-list)

If the user might have to pick more than one selection from a single menu (or screen), this function can be used to register one selection without redrawing the menu, using the pick-list as its index of coordinates and values. To do this:

a. Call MenuDraw or MenuDrawPlain to draw the menu and return the pick-list.
b. Call MenuMouseSelect using the pick-list each time the user is to select an item.

This function will return either one of the return tokens corresponding to the selected regions, or it will return the coordinates selected if the user clicks in a region that is not in the pick-list.

Coordinates are same as those returned by MouseClickLoc.

(MenuPlain display-list coord-list output-list)

Draws a menu and allows the user to select a choice. The menus drawn by MenuPlain do not include the standard default options.

The inputs:

display-list: A list of either strings or atoms. Strings are printed, and atoms are taken as function names. The function is used to draw an icon or some symbol that the user can see and 'click' on the display. It must be drawn with respect to (0,0), since MenuPlain executes SetOrigin to locate the icon before executing the function to draw the icon. Also, these functions must use the Draw, Fill, or StringOut functions, since MenuPlain depends on these functions to tell it the size and location of the images drawn on the display. All text output will have default foreground and background colors as specified by (EchoForeground) and (EchoBackground).

coord-list: Each entry in the display-list should have a corresponding coordinate pair. These coordinate pairs are relative to the origin. They specify the upper left-hand corner of the icon or string being output.

output-list: Each entry in the display-list should also have a corresponding return token in output-list. These tokens may be atoms, numbers or strings (strings are recommended). One of these return tokens will be returned if its corresponding icon/string is selected.

IMPORTANT: No return token can be a list.

```
Eg:   Lisp> (MenuPlain  '("See an example of what SPARC can do"
                         "Try problems given by SPARC"
                                "Challenge SPARC with a problem")
                '((100 0) (100 100) (100 200))
                                '("EXAMPLE" "TRY" "CHALLENGE"))
      ;; This might return "EXAMPLE", "TRY", or "CHALLENGE".
      ;; The first line of text is output starting at (100,0),
      ;; the second at (100,100), etc.
```

The Output:

The CAR of the output will be one of the specified return tokens. The second element of the returned list is a pick-list which can be used in a future function.

(MouseClickLoc)

Returns coordinates, (x y), of the location of the cursor on the screen when the mouse was clicked.

```
Eg:   Lisp> (MouseClickLoc)    ;;Click left mouse button now
      (540  302)
```

(NiceCTextClear y text [fontname])

Covers output produced by NiceCTextOut with background color.
Default font is the same as in NiceTextOutC

```
Eg:   Lisp> (NiceTextOutCClear 150 '("Are you there" "today."))
```

(NiceCTextOut y text [fontname [foreground [background]]])

Produces centered text enclosed in a frame. The location of the top of the box is given by y. The text is double spaced.
Defaults:

- font - given by (EchoFont)
- foreground - given by (EchoForeground)
- background - given by (EchoBackground)

```
Eg:   Lisp> (NicetextOutC 150 '("Are you there" "today."))
```

(NiceStringOut x y string [fontname [foreground [background]]])

Same as StringOut, except it draws a rectangle around the string.

(OpenDisplay)

NOT PRESENTLY EXPORTED!
Connects lisp to the X server (i.e. the X windows package)

(OptionBox x y text-list [foreground [background]])

Draws a selection square with the top left-hand corner at (x, y).
Both coordinate parameters should be multiples of 100.

```
Eg:   Lisp> (OptionBox 300 300 '("TRY" "TUTORIAL"))
```

(ResetLimits)

Resets the screen boundaries so that they can seek the maximum values of *x-min* and *y-min* and the minimum values of *x-max* and *y-max*.

(RuleBlockColor)

Returns the default color index for Rule Blocks, ie, the background on which Rule Copy

(text) is output.

(RuleCopyColor)
Returns the default color index for Rule text.

(RuleText y textlist [fontname])
Outputs the specified text as a Rule block with its top at y. Returns a list giving the width and height of the Text.
Default fontname is given by (EchoFont) (currently "helv12b")
Eg: *Lisp>* (RuleText '("Life is what happens while you're"
"busy making other plans."))

(RuleTextSize y textlist [fontname])
Returns a list giving the width and height of the Rule block produced by RuleText if given the same text and fontname.
Default fontname is given by (EchoFont) (currently "helv12b")
Eg: *Lisp>* (RuleTextSize '("Life is what happens while you're"
"busy making other plans."))

(SelectBlockColor)
Returns the background color index of Select squares.

(SelectCopyColor)
Returns the color index of the text in Select Squares.

(SelectSquare x y text-list)
This function corresponds to the Select Square given in the menu layout specs. Its only difference from OptionBox is that it does not have an optional parameter for colors.
Eg: *Lisp>* (SelectSquare 50 50 '("Choose" "Me"))

(SetBackground color-index)
Sets the background color to the specified color. Does not affect the existing display.

(SetFont fontname)
Sets the default font type. Fontname must be a string.

(SetForeground color-index)
Sets the foreground color to the specified color. Does not affect the existing display.

(SetOrigin x y)
Allows the user to set the origin of the Cartesian coordinates of all the Draw, Fill, FileOut, and TextOut functions to a point on the screen.
NOTES: It is good practice to call SetOrigin before using a Draw or a Fill function because other routines might have set it to some other value.
Remember to execute (SetOrigin 0 0) before exiting a routine that uses SetOrigin.
The default origin position is (0,0).

(SpecialBlockColor)
Returns the default color index of Special Blocks, ie, the background on which SpecialText is printed.

(SpecialCopyColor)
Returns the default color index of the text output by the SpecialText function.

(SpecialText x y <filename | textlist> [MarginList [fontname [foreground [background]]]])

Given the upper left hand corner of a rectangle, this function will output the text contained in the file specified by filename, or the text in the list of strings, textlist. The text is produced on a filled rectangle enclosing the text. MarginList specifies a set of Margins counterclockwise from the top. The top and left are are drawn precisely as specified. The right and bottom values specify a minimum width around the text. The right and bottom sides are aligned inside the nearest multiples of 25 which provide a margin of at least RIGHT and BOTTOM pixels around the text.

SpecialText returns a list containing the width and height of the box output, such as (50 100).

Defaults:

marginlist - Given by (*menuSpecialCopyTopMargin*
menuSpecialCopyLeftMargin
menuSpecialCopyBottomMargin
menuSpecialRightMargin)

fontname - Given by (EchoFont)

foreground - *MenuMediumGoldenRod*

background - *MenuViolet*

Eg: *Lisp>* (SpecialText 200 500 '("Hi," "how's the job?"))

Lisp> (SpecialText 200 200
(Full-Path "induce/menu1.txt") '(50 50 30 30))

NOTE: It is suggested that x and y be multiples of 25.

(SpecialTextSize x y <filename | textlist> [fontname [foreground [background]]])

Returns a list containing the width and height of the box output by SpecialText when given the same parameters, such as (50 100).

NOTE: No output to screen is produced.

Defaults:

fontname - Given by (EchoFont)

foreground - Given by (EchoForeground)

background - *menuViolet*

Eg: *Lisp>* (SpecialTextSize 200 500 '("Hi, how's the job?"))

Lisp> (SpecialTextSize 200 200 (Full-Path "induce/menu1.txt"))

(StringOut x y string [fontname [foreground [background]]])

Outputs the string of text starting at the location (x, y) away from the point set by SetOrigin. Foreground is the color index of the color of the text to be printed.

Defaults:

fontname - Given by (EchoFont)

foreground - Given by (EchoForeground)

background - Given by (EchoBackground)

It is suggested that the default font be used, so that there will be a consistent output format. The present maximum string length is 100 characters.

(StringSize string [fontname])

Returns the width and height (x and y) of the string in the form of a list. The default fontname is given by (EchoFont).

Eg: *Lisp>* (StringSize "I")

(10 25)

(Talk string [package [device [voice]]])

Outputs a text string to the default device tty00 which is assumed to be a DECtalk voice synthesis unit (produced by Digital Equipment Corporation.) Package specifies one of the following:
"EMERALD", "ABACUS", "AQ", "CLUSTER", "INDUCE", or "SPARC".
Device defines the output device.
If voice is specified, it must be a valid voice specification for the DECtalk unit. Otherwise, the voice will default to the voice assigned the selected package. See the DECtalk manuals for a complete description of available voice specifications.

Eg: *Lisp>* (Talk "Hello" "EMERALD")
Lisp> (Talk "Hello" NIL NIL "[:nb]")

(TitleOut text [fontname [foreground [background]]])

Outputs a title at a standard location on the screen.
Defaults:
fontname - Given by (EchoFont)
foreground - Given by (EchoForeground)
background - Given by (EchoBackground)

Eg: *Lisp>* (TitleOut '("This is" "a title"))
Lisp> (TitleOut '("This is a" "small title" "instead") "troman10"
(Getcolor "Red"))

(TitleClear text [fontname])

Covers the title produced by TitleOut with the background color.
Default font is the default from TitleOut

Eg: *Lisp>* (TitleClear '("This is" "a title"))

(UpdateLimits x y)

Updates the values of *x-min*, *y-min*, *x-max*, and *y-max* if they are exceeded by the pair (x,y).

(WaitTimeOut seconds)

Sets the time in seconds between automatic system resets.
NOTE: This function may have to be rewritten for EMERALD to be run on different machines, as different systems may have different timekeeping methods.

(WaitUser [Pick-list])

Waits for the user to select the NEXT SCREEN box. If a Pick-list is provided, it waits for the user to select any of the included items. Used in conjunction with HelpScreen.

2.3.3. Exhibit Guidelines

The following is a list of specifications which were made relatively early in the development of the EMERALD system. Functions have been written implementing each of the descriptions below. Various changes were made during the development of EMERALD, particularly the replacement of option boxes by text dubbed "ChoiceText", which is simply text which has been placed on a yellow rectangular background. In order to find a function which creates the specified object, look for a function of the same name in the list of available share functions (e.g. GeneralBlock ..... GeneralText).

(NOTE: functions which output "copy" (ie, produce text) have been given the suffix text to avoid confusion with routines which duplicate data.)

The Guidelines are rather loose. Thus, each portion of the system, such as AQ, ABACUS and so forth, has taken on an appearance which is somewhat unique in terms of its color scheme, but continues to have the unifying property of consisting of the Main and Mid backgrounds overlaid with additional rectangles. Inside these rectangles, text and diagrams can be found.

Before adding menus to any particular portion of the system, it is wise to examine a few of the screens already designed and to create new screens with a similar layout. System standards are as follows:

Main background	Backdrop for everything.
Usage:	(fillrect *MenuBlack* 0 0 1000 850)
Mid background	Most of what happens should actually appear here. ClearScreen will draw this. EchoBackground will give the color index for the region, currently defaulting to pink.
Usage:	(fillrect *MenuPink* 100 150 900 850)
Menu squares	These stay on the bottom of the screen at all times; if a square isn't used on a screen, the words can be left off the square, but the square itself should stay. Menu and MenuDraw take care of the squares at the bottom of the screen.

Usage:

```
(fillrect *MenuMaroon* 200 750 298 850)
(fillrect *MenuMaroon* 300 750 398 850)
(fillrect *MenuMaroon* 400 750 498 850)
(fillrect *MenuMaroon* 500 750 598 850)
(fillrect *MenuMaroon* 600 750 698 850)
(fillrect *MenuMaroon* 700 750 798 850)
```

Program title

Such as "ABACUS", "SPARC", etc. These remain on the screen at all times but may eventually be digitized. Therefore leave this space for them. Menu and MenuDraw will that care of displaying the title.

Usage:

```
(fillrect *MenuPlum* 350 0 800 150)
```

Robot

There is no need to bother with robots, unless it is desired that their designs be changed, or if you are creating a new module that will have its own robot. The Menu and MenuDraw functions will draw them when called.

Usage:

```
(fillrect *MenuWhite* 800 0 900 150)
```

Title

This describes your page. Titles S H O U L D B E L E T T E R S P A C E D, I N A L L C A P S, L I K E T H I S. Three spaces between words. Menu and MenuDraw take care of this if you specify the title1 option. (Of course you must specify the spaces between letters.) Colors and locations are specified as follows:

```
Foreground: *MenuCoral*
Background: *MenuBlack*
*MenuTitleX0*: 100
*MenuTitleY0*: 130
```

General blocks

These are the rectangles that surround most of the written information on the screen. In fact they may be thought of as frames for the information that is

specific to each screen.
Every page will probably begin with one. It is good practice to give each general block a little room at the end. For example, if a block of text will end on y=449, end the rectangle on y=475 or 500, rather than at y = 450. These rectangles should begin at x=50 and end at x=800.

Usage: (fillrect *MenuViolet* 50 175 800 ___)

General copy

This describes the majority of the written text on the screen. The low y-coordinate boundary should be 25 units down from the top of the "Copy block" and the copy itself should fall between x=100 and x=750. Parameters are set as follows:

Foreground: *MenuMediumGoldenrod*
Background: *MenuViolet*
Font: (EchoFont) = Helv12b

Command blocks

Every time the user is instructed to do something specific, it should appear in a medium goldenrod block, to highlight it. Overlap your "general block" with this, beginning at y coordinate equal to the next multiple of 25. Regardless of the length of the command, the rectangle should run from x=50 to x=800.

Usage: (fillrect *MenuMediumGoldenrod* 50 y1 800 y2)

Command copy

The words should always begin at x=100, and go no further than x=750. If your command block begins at y=25, the copy should begin at y=30. Colors and fonts are as follows:

Foreground: *MenuViolet*
Background: *MenuMediumGoldenrod*
Font: (EchoFont) = Helv12b

Rule blocks

Like command blocks, these appear to highlight

	information, in this case, the rule or answer that a particular robot has discovered. Overlap the "general block" with this, with the beginning y-coordinate at the next multiple of 25.
Usage:	(fillrect *MenuPaleGreen* 50 y1 800 y2)
Rule copy	The words should always begin at x=100, and go no further than x=750. The following colors and font are used: Foreground: *MediumGoldenrod* Background: *MenuBlack* Font: (EchoFont) = Helv12b
Special copy	Whenever text is output at an arbitrary location on the screen, it shall be termed Special copy. Its upper left corner should be aligned on a boundary which is a multiple of 25, and its lower right corner should be aligned one unit to the left and above a boundary which is a multiple of 25.
Select Squares	These are the options for which a viewer has to use the mouse or track-ball. Ideally, they are 100 units square (or 98 units square if they are adjacent) and appear in the spaces above the menu squares (x = 200 to 298) ... (x=700 to 798). The standard color is *MenuBlueViolet*
Icon Blocks	Where trains, shapes, cards, etc. appear. Again, use the grid where possible. Don't feel that all the horizontal or vertical space must be taken up. Line the blocks up from the left at x=200 if possible.

We will now present examples of several SHARE function calls. These functions help generate menus as specified above.

NOTE: "text" is used in function names as a synonym for the word "copy" in the specs (in README.menu).

```
Lisp>    (GeneralText '("This is a short" "example."))
Lisp>    (GeneralText (Full-Path "share/txt/pg1.txt"))
Lisp>    (GeneralBlock height)
Lisp>    (CommandText 50 '("Please select an option"))
Lisp>    (RuleText 50  '("Blue birds have wings" "and Red birds don't"))
Lisp>    (SelectSquare x y '("ABACUS"))
Lisp>    (SelectSquare x y '("CHALLENGE" "SPARC"))
Lisp>    (SpecialText 200 500 '("Hello," "how is the weather?"))
Lisp>    (SpecialText 200 500 (full-path "share/txt/pg1.txt"))
```

Look in section 2.2.1 for more detailed descriptions.

2.4. Index of SHARE Functions

The following index contains a list of all functions defined in the SHARE package. For those which are used by the typical menu programmer see section 2.2.1 of this document. The functions in the index are sorted by function name which appears in the second column of the index. To use the index:

1. Find the desired function name after the colon.
2. The lisp file in which it is contained is found to the left,
   the arguments in its definition to the right.

The files are presently stored in the directory share/lsp, with the exception of Full-Path, which is loaded from the file emerald.lsp in the system's home directory.

File	Function (arguments)
icons.lsp:	AbacusClear (x y &optional scale)
icons.lsp:	AbacusIcon1 (x y &optional scale)
icons.lsp:	AbacusIcon2 (x y &optional scale)
icons.lsp:	AbacusIcon3 (x y &optional scale)
icons.lsp:	AbacusIcon4 (x y &optional scale)
icons.lsp:	AbacusIcon5 (x y &optional scale)
icons.lsp:	AqClear (x y &optional scale)
icons.lsp:	AqIcon1 (x y &optional scale)
icons.lsp:	AqIcon2 (x y &optional scale)
icons.lsp:	AqIcon3 (x y &optional scale)
icons.lsp:	AqIcon4 (x y &optional scale)
icons.lsp:	AqIcon5 (x y &optional scale)
beep.lsp:	Beep ()
screen.lsp:	BlankDisplay (&optional color-index)
callunix.lsp:	CallUnix (command)
stdchoice.lsp:	ChoiceText (y text &optional font)
stdchoice.lsp:	ChoiceTextSize (text &optional font)
screen.lsp:	ClearScreen (&optional color-index)
icons.lsp:	ClusterClear (x y &optional scale)
icons.lsp:	ClusterIcon1 (x y &optional scale)
icons.lsp:	ClusterIcon2 (x y &optional scale)
icons.lsp:	ClusterIcon3 (x y &optional scale)
icons.lsp:	ClusterIcon4 (x y &optional scale)
icons.lsp:	ClusterIcon5 (x y &optional scale)
stdmenu.lsp:	CommandBlockColor()
stdmenu.lsp:	CommandCopyColor()
stdmenu.lsp:	CommandText (y text &optional font (color *menuCommandBlockColor*))
stdmenu.lsp:	CommandTextSize (y text &optional font)
x2lisp.lsp:	DisplayHeight ()
x2lisp.lsp:	DisplayWidth ()
drawpoly.lsp:	DrawCircle (clr-index x y radius &optional th)

drawpoly.lsp:	DrawCurPoly (colornum pointlist &optional thickness)
drawpoly.lsp:	DrawLine (c x1 y1 x2 y2 &optional th)
drawpoly.lsp:	DrawPoly (colornum pointlist &optional thickness)
drawpoly.lsp:	DrawPolyNoUpdate (colornum pointlist &optional thickness)
drawpoly.lsp:	DrawRect (c x1 y1 x2 y2 &optional th)
screen.lsp:	EchoBackground ()
screen.lsp:	EchoFont ()
screen.lsp:	EchoFontHeight()
screen.lsp:	EchoForeground ()
stdmenu.lsp:	EchoGeneralBlockTop()
limits.lsp:	EchoLimits ()
origin.lsp:	EchoOrigin ()
icons.lsp:	EmeraldClear (x y &optional scale)
icons.lsp:	EmeraldIcon1 (x y &optional scale)
icons.lsp:	EmeraldIcon2 (x y &optional scale)
icons.lsp:	EmeraldIcon3 (x y &optional scale)
icons.lsp:	EmeraldIcon4 (x y &optional scale)
icons.lsp:	EmeraldIcon5 (x y &optional scale)
enterExh.lsp:	EnterExhibit ()
center.lsp:	ErrorClear (text &optional font)
center.lsp:	ErrorOut (text &optional font fgnd bgnd)
enterExh.lsp:	ExitExhibit ()
fileout.lsp:	FileOut (x y filename &optional fontname foreground background)
fileout.lsp:	FileSize (filename &optional fontname)
fillpoly.lsp:	FillCircle (clr-index x y radius)
fillpoly.lsp:	FillCurPoly (colornum pointlist)
fillpoly.lsp:	FillPoly (colornum pointlist)
fillpoly.lsp:	FillRect (c x1 y1 x2 y2)
exhibit.lsp:	Full-path (x)
stdmenu.lsp:	FullRect (c x1 y1 x2 y2)
stdmenu.lsp:	GeneralBlock (height &optional ytop)
stdmenu.lsp:	GeneralBlockColor()
stdmenu.lsp:	GeneralBlockLeft()
stdmenu.lsp:	GeneralBlockTop()
stdmenu.lsp:	GeneralBlockSize (height &optional ytop)
stdmenu.lsp:	GeneralCopyColor()
stdmenu.lsp:	GeneralText (text &optional y MargList font fgnd bgnd)
stdmenu.lsp:	GeneralTextSize (text &optional y MargList font fgnd bgnd)
screen.lsp:	GetColor (string)
help.lsp:	HelpScreen (pname filename &optional font title)
menu.lsp:	IconHelp ()
menu.lsp:	IconMainmenu ()
menu.lsp:	IconNext ()
menu.lsp:	IconPrevious ()
menu.lsp:	IconQuit ()
menu.lsp:	IconSubmenu ()
menu.lsp:	IconText (x y t1 &optional t2 t3)
icons.lsp:	InduceClear (x y &optional scale)
icons.lsp:	InduceIcon1 (x y &optional scale)
icons.lsp:	InduceIcon2 (x y &optional scale)
icons.lsp:	InduceIcon3 (x y &optional scale)
icons.lsp:	InduceIcon4 (x y &optional scale)
icons.lsp:	InduceIcon5 (x y &optional scale)

menu.lsp:	Menu (dlist clist olist pname &optional t1 t2)
menu.lsp:	MenuDraw (dlist clist olist pname &optional t1 t2)
menu.lsp:	MenuDrawPlain (dlist clist olist)
menu.lsp:	MenuIconDraw(pname)
menu.lsp:	MenuIconOut (function x y ret)
menu.lsp:	MenuMouseSelect (plist)
menu.lsp:	MenuPlain (dlist clist olist)
menu.lsp:	MenuProgramTitle (pname)
menu.lsp:	MenuStringOut (str x y &optional ret)
menu.lsp:	MenuTitle (t1 t2)
mouse.lsp:	MouseClickLoc ()
center.lsp:	NiceCTextClear (y text &OPTIONAL font)
center.lsp:	NiceCTextOut (y text &OPTIONAL font fgnd bgnd)
menu.lsp:	NiceStringOut (x y str &optional font fgnd bgnd)
BBB_opendisplay.lsp:	OpenDisplay ()
optionbox.lsp:	OptionBox (x y txt &optional fg bg)
talk.lsp:	Punctuated? (string)
limits.lsp:	ResetLimits ()
stdmenu.lsp:	RuleBlockColor()
stdmenu.lsp:	RuleCopyColor()
stdmenu.lsp:	RuleText (y in-text &optional font)
stdmenu.lsp:	RuleTextSize (y text &optional font)
talk.lsp:	Say (string voice device)
stdmenu.lsp:	SelectBlockColor()
stdmenu.lsp:	SelectCopyColor()
stdmenu.lsp:	SelectSquare (x y text)
screen.lsp:	SetBackground (color-index)
screen.lsp:	SetFont (fontname)
screen.lsp:	SetForeground (color-index)
origin.lsp:	SetOrigin (x y)
icons.lsp:	SparcClear (x y &optional scale)
icons.lsp:	SparcIcon1 (x y &optional scale)
icons.lsp:	SparcIcon2 (x y &optional scale)
icons.lsp:	SparcIcon3 (x y &optional scale)
icons.lsp:	SparcIcon4 (x y &optional scale)
icons.lsp:	SparcIcon5 (x y &optional scale)
stdmenu.lsp:	SpecialBlockColor()
stdmenu.lsp:	SpecialCopyColor()
stdmenu.lsp:	SpecialText (x y text &optional MargList font fgnd bgnd)
stdmenu.lsp:	SpecialTextSize (x y text &optional MargList font fgnd bgnd)
stringout.lsp:	StringOut (x y string &optional fontname foreground background)
center.lsp:	StringOutC (y str &OPTIONAL font fgnd bgnd)
stringout.lsp:	StringSize (string &optional fontname)
talk.lsp:	Talk (string &optional package device voice)
center.lsp:	TitleClear (text &optional font)
center.lsp:	TitleOut (text &optional font fgnd bgnd)
stringout.lsp:	String-length (string)
limits.lsp:	UpdateLimits (x y)
waittimeout.lsp:	WaitTimeOut (seconds)
help.lsp:	WaitUser (&optional plist)
BBB_opendisplay.lsp:	YN-Question (quest)

3. AQ ROBOT

The code for the AQ portion of the exhibit essentially consists of a LISP front end to the Pascal implementation of the AQ15 algorithm. This portion of the programmer's guide consists of two main sections. The first section describes the LISP code (which includes some C code for optimized graphics). The second section briefly describes the AQ15 Pascal code. This documentation assumes that the reader has run the AQ portion of the exhibit and has a working knowledge of the languages involved. All files mentioned will appear in the aq subdirectory of the exhibit's home directory, unless explicitly stated otherwise.

3.1. Front End Code

A file of code for the front end will be one of three types: LISP code, both high level routines and interfaces with the AQ15 learning engine; C code, or text files (which, although not strictly considered code, are important elements of the front end). Each type of code is explained below in a separate subsection.

3.1.1. High-Level LISP Functions

First and least noteworthy is the file aq.lsp. It comes first in running the exhibit because it loads all the necessary lisp files for running AQ. It is least important, because it contains only a few lines of code, which are interpreted only at the time EMERALD is loaded, and are not usually encountered again. It is occasionally useful when the system is running, and all (or most) of AQ needs to be reloaded. The really important LISP code consists of the high-level functions and the AQ Interface routines.

In keeping with the convention for all "robots" in the exhibit, the top-level function for AQ is **aq-main** (found in aqmain.lsp). Aq-main is responsible for soliciting responses from the main menu. The responses are processed by the **Response1User** function which calls the appropriate function for each item on the main menu. These four functions comprise the second level of code. That is, aside from aq-main and Response1User, no other function in the AQ code is called without calling one of the four "option'" functions determined by a selection from the main AQ menu. Table 3.1 shows the four main menu options along with their corresponding high-level functions.

Each of these high-level functions uses an associated response processing function much like aq-main uses Response1User. Each response processing function consists of a loop which performs two responsibilities at each iteration. The first responsibility is the handling of the response, which usually involves incrementing or decrementing the current page (screen) number for the option. The second responsibility is the invocation of a page vector function, usually called without an argument, which checks the current page (screen) number for the option, and calls the function which draws the appropriate screen. Table 3.2 shows the correspondence between the high-level functions and their response processor and page vector counterparts. (Note there is no associated vector with the HowAqWork function because the option has only one screen.)

At first this architecture may seem somewhat confusing. Indeed, the requirement that one function must handle all appropriate responses for all screens within a high-level option causes a fair amount of convoluted code. Nevertheless, if responses are more or less standard within a given option (and they generally are), the convention does provide for relatively straightforward addition of screens within an option. A new high-level option can also be added without much trouble simply by copying and modifying a similar "option function"/responder/pager triumvirate.

OPTION	FUNCTION
AQ challenges you with a simple problem	**SeeExamples**
Challenge AQ with a simple problem	**TeachMeConcept**
Challenge AQ with a more complex problem	**ChallengeMe**
Find out how AQ works	**HowAqWork**

Table 3.1. Important High-Level AQ Functions

Another advantage of the architecture is the existence of one concise main function for each screen.

These functions are easily identified, since they consist of a base name which is similar (if not identical) to the page vector function, followed by the number of the screen in the screen sequence for the option. For example, the third screen for the TeachMeConcept option is handled by the function **TDPage3**. Program modification and debugging seldom requires tracing functions more than two deep from the screen function. The only exceptions occur when the AQ15 algorithm is invoked. These interface functions are more complex and are dealt with in a separate section below.

High-Level Function Location. The functions described in Table AQ.2 can be found among three files: aqmain.lsp, which contains both the tutorial (the **SeeExamples** functions) and the explanation screen (the **HowAqWork** functions), aqteach.lsp, which contains the simple challenge (the **TeachMeConcept** functions), and aqchal.lsp which contains the complex challenge (the **ChallengeMe** functions). These files should all be compiled before loading. Compilation can be done in UNIX using an option to the ``lisp" command, or in LISP using the **compile-file** function with the name of the file to be compiled as the only argument, in string form. The compiled files have the same name as the LISP files, but have a "fas" extension instead of ".lsp".

OPTION FUNCTION	RESPONSE PROCESSOR	PAGING
SeeExamples	**Response2User**	**DisplayPages**
TeachMeConcept	**Response3User**	**TDPage**
ChallengeMe	**Response4User**	**EDPage**
HowAqWork	**Response5User**	——

Table 3.2. Associated High-Level Functions.

3.1.2 AQ Interface Routines

These functions construct an input file for the AQ15 program, invoke AQ15, read the output file and translate the results into English text. The top-level function is called **discoverrules**, which takes as its only argument a list of groups of robots. This function calls a long series of functions which perform the steps outlined below. The call to the compiled Pascal implementation (discussed separately later) is achieved by the LISP **call-out** function. This function is called with the UNIX system function "system" as an argument. This simply executes a UNIX command line, which in this case is the command line for invoking AQ15. (This command line string is the second argument to call-out.) AQ15 takes one argument, the path to the aq directory.

AQ15 requires several files to operate properly. It expects to find properly formatted input in the file **robots.inp**. It leaves its output in a file called **robots.out**. It also maintains a history of previous rules in **rulehist**, and uses this information in avoiding redundancy when finding alternative rules. Similarly the LISP routines in the interface require the files **head** and **examples** to build the input file for AQ15.

A special case worth noting is when a characterization for only one group is generated. This is done without invoking AQ15, using essentially the **refunion** operation with a simple trimming procedure. This operation, along with the translation procedure, involves a long sequence of short, clearly understandable functions, which can easily be followed in the code, and therefore, will not be discussed further in this documentation.

All LISP functions noted above can be found in the file aqrule.lsp. This file should also be compiled, hence an aqrule.fas file should also exist.

3.1.3. C Code

One small file of C code is used to efficiently draw the robots for AQ. This file is called robot-icon.c. It consists of several functions for drawing heads, mouths, balloons, etc. The interface file for these routines is written in LISP and is contained in aqrobot.lsp (compiled version in aqrobot.fas). The top-level function is called robots, which supplies all the proper parameters to a C function (of the same name) given a robot as an argument. The other arguments are a base coordinate and a flag for the background color.

The C file robot-icon.c must be compiled with the -c option to the C compiler, which creates the file robot-icon.o.

3.1.4. Text Files

Although technically not code, text files represent a large percentage of the portions of EMERALD most likely to be changed. To change most of the text on any screen, one can usually escape the exhibit, get to UNIX command level either by Ctrl-Z or opening a new window, edit the appropriate file, and restart EMERALD without reloading or compiling any files whatsoever, (although a repositioning of a new size of textblock is often unavoidable).

Text files have names of the form text<n><i> where n is an integer and i is an optional additional index as in "text41" and "text8b". To change the text on a given screen, look at the handler routine for that screen to find the name of the text file(s) it uses. Then simply edit the file(s), and restart (it is not necessary to reload) the exhibit.

Help files are simply text files that are only read by the exhibit function *helpscreen*, which was described in section 2.3.1.1. AQ has five help files: **help.text** and **help.text1** through **help.text4**.

3.2. AQ15

Currently only the compiled version of AQ15, located in (Full-Path "aq/aq15"), can be found under the EMERALD directories. The Pascal source was determined not to be likely to require modification, and was therefore not included. Details on this code may be found in the references on AQ15.

4. INDUCE ROBOT

4.1. Introduction to INDUCE

INDUCE is a system capable of learning concepts about objects or groups of objects by example. Unlike many other learning-by-example systems, INDUCE is capable of finding relationships between the parts that make up a single example. For example, if INDUCE is trying to learn concepts relating to trains, as it is in the EMERALD system, it can also utilize knowledge about individual cars and even the relationships between different cars.

4.2. A Guide to INDUCE on EMERALD

4.2.1. Basic Structure

All of the INDUCE code, with the exception of the functions and routines shared by all sections of the EMERALD system, are found in the induce subdirectory, or subdirectories thereof. Unless otherwise noted, all files referred to will be found in the induce subdirectory.

Loading INDUCE is a trivial matter. Provided that the user is in the LISP environment and has already loaded the SHARE functions, the only necessary step is to enter:

(load (full-path "induce/induce"))

The main controller for INDUCE is found in the file **indmenu.lsp**. This file executes a large loop, displaying the proper screen on each iteration. The screens are referred to in the form "m#", where m stands for "menu", and # is the number of the screen. Due to the design changes during development of the code, not all of the numbers are contiguous, and in some cases, related screens are distinguished by trailing letters, eg, m2 and m2a. Indeed, different versions of EMERALD may employ different screens and different screen numbers, but the system was developed in such a way that isomorphic screens ought to have identical numbers.

In the current version, the following numbering convention holds:

m1 is the main menu; here the user selects which portion of the INDUCE exhibit to view.

m1a through m5c offer a tutorial on the way INDUCE solves problems, and show examples of the types of problems INDUCE can solve. At certain key stages of the tutorial, the user is asked to come up with solutions to some of these problems. **m1a** is a submenu allowing the user to go directly to m2, m4 or m5. **m2** and **m2a** display the learning of an arch as an incremental process, and **m3** shows it as a non-incremental process. **m3a** summarizes the problem of arch (concept) learning, and touches upon the difference between incremental and non-incremental learning. **m4** gives an example of a multiple group problem in the domain of geometric figures, and the user is asked to find a complete and consistent description of one of the groups. The **m5** series introduces the user to the domain that will be used for the challenge portion of the INDUCE exhibit -- that of distinguishing between groups of trains.

m19 encompasses the challenge portion of the INDUCE demonstration. Unlike the rest of INDUCE, most of this code is not in the file indmenu.lsp, but rather in the subdirectory inmenu. A subdirectory of this, *lsp*, holds the source code, all of which is cat-ed into one large file, **inmenu.lsp**, in the directory above. Three main submodules make up the inmenu file. The main one is known as **trainmenu**, and it serves to handle all the intricacies of the user interface during the challenge phase, plus perform the calls to the INDUCE program itself. The other two submodules, **drawcars** and **iofuns**, assist in the display of trains on the screen.

m27 displays a screen which briefly explains how INDUCE works, and lists several actual applications. **conc** shows a concluding screen in order to ensure that the user hasn't missed any major facts about INDUCE.

4.2.2. INDUCE Algorithm Code

This section describes the code that implements the INDUCE algorithm. This code is written entirely in LISP. The author of the original version of this code is unknown, although Brad Whitehall was responsible for a large amount of the modification to the original code as part of a class project for Professor Robert E. Stepp, Department of Electrical and Computer Engineering, University of Illinois. Theoretical descriptions of the INDUCE algorithm can be found in "A Theory and Methodology of Inductive Learning" by R.S. Michalski in Machine Learning, (Tioga 1983) and various University of Illinois Department of Computer Science Reports by Professors Michalski and/or Stepp. As is often the case, these theoretical descriptions are of limited practical

value as far as understanding the code is concerned. Nevertheless a vague understanding of how INDUCE works is helpful as a starting point for comprehending the 3300+ lines of LISP that perform the algorithm. A working knowledge of LISP is essential.

The code will be described in terms of its various modules (files). The role of the module in the algorithm will be discussed along with the important functions of the module. Any modifications made especially for EMERALD will be noted. All files below should be compiled (have ".fas" counterparts) unless stated otherwise.

defs2.lsp

A wise man once said that the best place to start is at the beginning. Obviously he was not a software engineer, who would know that the best place to start is with the data structures. The file "defs2.lsp" provides a well-commented description of each of the major data structures used by INDUCE, as well as the definitions for many associated functions (print functions, data conversion functions, etc.) A brief look at this file will make starting from the beginning significantly easier.

induce12.lsp

This file is where the algorithm actually begins. It contains the top-level function **INDUCE-1**. Perhaps more importantly, it contains the declarations for the various parameters which can produce dramatic changes in the performance and output of INDUCE. The parameter of greatest value to the INDUCE programmer is undoubtedly the *INDUCE-TRACE* parameter, which allows the programmer to see what INDUCE is doing. This parameter can be set in LISP using SETQ or SETF when the exhibit is stopped. When the exhibited is restarted (using the (X) function), the trace will automatically be in effect. (The programmer must be in the INDUCE package or specify INDUCE::*INDUCE-TRACE* when setting this and any other parameter interactively.) Increasing values for the trace parameter will increase the depth of the trace. The programmer can determine the precise nature of the trace by searching for statements containing the trace variable (while within an editor, for example), and is certainly free to add new trace statements when such is deemed appropriate.

The other parameters in this module determine the number of complexes generated and trimmed at key stages in the algorithm. The programmer is warned that the default values reflect hours of experimentation and that changing the values without saving the old values will most likely result

in inferior performance. Not that these parameters cannot be adjusted more optimally, but minor changes have most likely been tried with less than desirable results. Also worth noting is that tolerances for the lexicographical evaluation functions (LEF's) are not always as they appear, and peculiarities in the evaluation functions themselves should be thoroughly investigated before changing a tolerance that seems not to make sense.

Now that the programmer has been presented with the pitfalls of parameters, a brief description of the high-level INDUCE functions is in order. INDUCE-1 (the high-level function) invokes two major operations. The first is the invocation of **BASIC-INDUCE**, the INDUCE covering algorithm. Receiving several complexes (conjunctions) back from BASIC-INDUCE, INDUCE-1 calls **CALL-AQ** (the version of AQ modified for the INDUCE system) to generalize the references (values) of the selectors (conjuncts) in the complex. INDUCE-1 performs these operations iteratively until all the positive events are covered.

INDUCE-1 was modified for the EMERALD system to construct alternative rules. An extra parameter (old-seed-hypos) was added to INDUCE-1 which consists of a list of complexes used in previous rules. At each iteration, INDUCE-1 removes any complex from its list of candidate complexes which also appears on the list of previous complexes. This assures that no previous rule will ever be duplicated.

induce22.lsp

This module contains most of the LEF functions and all functions necessary for calling the aq portion of the algorithm. The aq functions convert the induce graph-structured representation to one suitable for input to aq, and vice versa for aq's output.

aq.lsp

This module contains the code which implements a simplified aq algorithm. Declarations for parameters to aq appear at the end of this file.

graph-match2.lsp

The routines found in this file perform the covering operations for the INDUCE algorithm.

build-graph2.lsp

This module builds graph structures for input events. The main function in this module is **BUILD-EVENT**, which internalizes data from input specifications. This function is only called from the input file "alltrains.lsp", which contains the definitions for all the trains used by INDUCE.

init2.lsp

This file loads all the INDUCE algorithm code into LISP. It serves no other purpose.

vltoeng.lsp

This file contains the functions which convert the complex(es) returned by INDUCE into naturally-sounding English. Most of the work is done by the function **subst-var**, which searches for various patterns in the input text, and replaces certain strings with others. The resulting sentences are passed back to trainmenu in the form of a list of strings, which are then written on the screen, and piped through the DecTalk unit.

Other Files

Several other files can be found in the INDUCE directory. These files are enhancements to the INDUCE algorithm which were not implemented in the EMERALD display primarily due to time and performance restraints. "increment2.lsp" allows INDUCE to do incremental learning. "bk2.lsp" allows background knowledge to be used by the system. "meta.lsp" handles meta-selectors such as FOR-ALL, MOST, LEAST, etc. These files have been maintained should such enhancements be deemed worthwhile at a later date.

5. CLUSTER ROBOT

5.1. Overall Description

The CLUSTER demonstration has three main screens: CLUSTER Challenges User, User Challenges CLUSTER, and How CLUSTER Works. These screens, the auxiliary screens they call, and the main routine that controls them all, form the core of CLUSTER. This chapter is divided into four sections corresponding to the main control routine and the three main screens that it calls.

5.2. The Cluster-Main Routine

The routine **Cluster-Main** performs most of the work involved in moving from screen to screen based upon the user's mouse input. This routine is implemented as a large case statement within a do loop. Each screen that the user is able to visit is represented as one of the cases within the case statement. Each case has a name, which is an atom, such as "screen1"; this label identifies the screen which is controlled here. The body of each of the cases follows the template:

```
(screenX
   (setf response (screenX))
   (cond ((string-equal response "QUIT") (next-page 'exit))
         ((string-equal response "PREVIOUS") (backup-page))
         ((string-equal response "HELP") (next-page 'helpX))
         ((string-equal response "NEXT") (next-page 'screenY))
         ...
         ))
```

The function ScreenX displays the text and graphics for screenX, along with the standard menu options and any non-standard ones needed for this screen. It then waits for the user to select an icon, and returns the token associated with that icon. The cond statement should have one clause for each token that can be returned by screenX. Each ``cond" clause specifies where control should pass next. Control starts at **Screen1**, the main menu screen for the CLUSTER demonstration.

The macros **next-page** and **backup-page** are used within the cond statement of each case to specify which screen control should pass to next. The statement *(next-page 'foo)* saves the current page label on a page stack and then passes control to the case labeled "foo". The statement

(backup-page) pops the label of the previously visited page off of the page stack, and then passes control to the case with that label.

Cluster-Main will return to the top-level exhibit menu whenever control passes to the page "exit". This may occur explicitly, by a call to next-page, or implicitly, by popping the last page label (which is set to "exit") off of the page stack. The return value of Cluster-Main will be the token associated with the last icon picked by the user. This token should always be one of the standard options.

The first screen given control by Cluster-Main is **Screen1**. This screen serves as the main menu to the CLUSTER demonstration, and allows the user to proceed directly to one of several sections.

5.3. CLUSTER Challenges the User

The purpose of this section of CLUSTER is to demonstrate the basic ideas behind conceptual clustering. The user is presented with a set of geometric figures, and asked to think about how they might be separated into groups. The user is then shown a set of several possible groupings, and asked to choose which one seems to be the best. A message is then displayed stating whether or not CLUSTER would choose that grouping. If a grouping that CLUSTER would not create was selected, the user is then given the option of choosing again.

The function **Screen3** handles the first screen shown in this section. This screen displays a set of geometric shapes, and asks the user to think about what groups this set could be divided into. The user is told to select NEXT SCREEN to continue. The actual graphics for this screen are produced in the routine **Screen3-Graphics**. This routine currently uses absolute screen coordinates, but in the future it might be useful to have the figures defined in terms of coordinates relative to the upper-left corner of the graphic area. In that way, the graphic area could be easily moved around to accomodate changes in the screen.

The next screen seen by the user is produced by the routine **Screen4**. Here the user is shown three possible groupings for the figures shown in Screen3, and asked which grouping looks the best. The actual drawing of the figures is done by the routines **Ex-Cl1-Icon**, **Ex-Cl2-Icon**, and **Ex-Cl3-Icon**. These routines draw both the entire area of each grouping and the grouping's accompanying text into an icon, so that the user may select anywhere within the region to choose that grouping. Since these routines are designed to create icons, the graphics are defined

relative to the upper-left corner of the region.

5.4. User Challenges CLUSTER

When the user decides to challenge CLUSTER, function **Screen7** is invoked. Screen7 will manage the user's selection of trains, and initiate the calling of CLUSTER if that option is selected. Function **Build-Train-Clustering** coordinates the data entered by the user, and stores it in a form understandable by CLUSTER's inference engine.

If the user decides to view CLUSTER's groupings and a legal group of trains (ie, at least two) has been entered, the function **Cluster-On-Few-Attr**, which resides in file call-cluster.lsp, will be invoked. This function manages the actual CLUSTER interface, and pumps the returned groupings through function **Display-Cluster-Results**. Of particular importance is the call to **Cluster2**, the main function for CLUSTER's inference process. Note that Cluster2 and its subfunctions are located in the cluster2 subdirectory of the main CLUSTER directory. See the referenced work by Stepp and Michalski for details on the clustering algorithm.

Other files contain useful support functions. The file **draw.lsp** contains the necessary functions to draw the trains on the screen. Instructions for the display of the discovered groupings may be found in **ruleout.lsp**. And **vltoeng.lsp** handles the problem of converting the returned groupings into descriptions much closer to natural-sounding English.

5.5. How CLUSTER Works

This portion of the CLUSTER demonstration is analogous to the other sections in which the workings of a system are described. It shows a single screen display briefly describing the inference algorithms used by CLUSTER, and some of the practical applications that have been found for it.

5.6. System Variables Used by CLUSTER

attribute-conversions

This list specifies the attribute classes, and the class-instance value pairs for each class, for all the attributes of the train set.

curr-attrs

Saves the list of attributes we are currently trying to cluster on so that they can be used by another routine to sort out he result returned by CLUSTER.

number-of-thinking-menus

The number of different "wait a minute while I think" messages to be cycled through.

next-thinking-menu

The index of the next "thinking" message.

thinking-talks

The text of the "thinking" messages in a form that can be used by the "talk" routine.

thinking-text-files

The names of the filenames that correspond to each "thinking" message. They are expected to be found in ".../cluster/menu_txt".

TrainIconBackground

Default background color for the train icon. Currently white

TrainIconForeground

Default foreground (outline) color for the train icon. Currently black.

TrainIconLen

Length of the train icon in pixels. Currently set to 510.

TrainIconWidth

Width of the train icon in pixels. Currently set to 50.

5.7. Functions and Macros Used by CLUSTER

In file **call-cluster.lsp**:

Build-Examples (choices all-examples)

This routine constructs and returns a list of those examples extracted from all-examples, which is a list of all the pertinent data about the examples in the set that has been presented to the user. The parameter choices is a list of offsets into all-examples specifying the examples selected by the user. The examples are returned in the same order as the offsets given in the choices parameter.

Build-Item-Num-Classes (choices cluster-result)

Builds a list grouping the results returned by CLUSTER with the index numbers of the choices that belong in each group and the text description of the attribute values that were used to produce the group.

Build-New-Ex (attrs item-ex)

Item-ex is a list of examples chosen by the user from the current class.

Attrs is a list of offsets into each example specifying the attributes whose values we wish to be considered by CLUSTER at this time.

This routine extracts the relevant data for each attribute from each example, and constructs a list of these values. A list of such lists for each example is returned.

Build-New-Vars (attrs variables)

The variables parameter is a list of all the attributes that examples of the current class may have, along with the values that these attributes may have.

The attrs parameter is a list of offsets into variables specifying the attributes that we wish to be considered by CLUSTER at this time.

This routine extracts the relevant attributes from variables and returns them as a list.

These attributes are returned in the same order as the offsets in attrs.

Cluster-On-Few-Attr (variables choices item-data icon-dim icon-func short-attrs)

Extracts the data for the examples chosen in choices from the list item-data, and from that extracts the attributes specified in the elements of the short-attrs list (one element per call to this routine). Then calls CLUSTER proper to try to determine a clustering for the examples based on these attributes. The list, variables, describes what each attribute is, and what

values it may have.

Display-Cluster-Results (choices item-data icon-dim icon-func cluster-result)

Calls Build-Item-Num-Classes to bundle the user's examples into the groups created by CLUSTER, and then passes this information to Display-Results which handles the actual screen layout. If CLUSTER could find no grouping for these examples, Display-Sorry is called to inform the user of this fact.

Display-Results (item-data icon-dim icon-func item-classes)

Draws the actual screen displaying which examples CLUSTER has placed in which group, and what those groups are. *Item-data* contains all the necessary information for all he items in the current class. *Icon-dim* is a list containing the length and width (in pixels) of the icons for this class. *Icon-func* is the name of the function that is used to display icons of the items in this class. It takes the arguments:
(left-most-x-coord top-most-y-coord data-about-item).
Item-classes is the list returned by **Build-Item-Num-Classes**. This list is composed of pairs which represent each class that has been built. The first element of each pair is a list of the offsets into Item-data of each item in that class, and the second element is a string that describes why these items are grouped together.

Display-Sorry ()

Displays a message stating that CLUSTER can find no further groupings of the examples it was given.

Display-Thinking (file-name)

This routine displays a "thinking" message on the screen while the clustering program tries to cluster trains. The routine draws a yellow rectangle in the center of the screen, and displays the text from the given file in it. The parameter file-name is one of "think[0-4]", which are found in the directory (FullPath "cluster/menu_txt").

Fold (list1 list2)

Given two lists of equal length, this function forms a list of pairs of one element from each list. For example: (a b c) and (x y z) would be folded together to produce ((a x) (b y) (c z)).

In file **cluster-main.lsp**:

Backup-Page ()

This macro pops the top of the previous page stack and places that value into the control variable *page*, so that the last page visited will be the one to which control is transferred.

Cluster-Main ()

This is the main control loop for the cluster demo package. Control starts at Screen1, which prints the main menu for the demonstration and accepts user input from the mouse. Control then passes to that section of code that controls the
screen the user chooses to view.

The routine is effectively a large case statement within a do loop. Each screen that the user can visit is represented as one of the cases within the case"statement. Each case has a name, identified by an atom, such as Screen1. This identifies which
screen is being controlled. The body of each of the cases follows the pattern:

```
(ScreenX
   (setf response (ScreenX))
   (cond ((string-equal response "QUIT") (next-page 'exit))
         ((string-equal response "PREVIOUS") (backup-page))
         ((string-equal response "HELP") (next-page 'helpX))
         ((string-equal response "NEXT") (next-page 'screenY))
         ...
         ) )
```

The function ScreenX displays the text and graphics for screenX, waits for the user to select an icon, and returns the token associated with that icon. The cond statement should have one clause for each token that can be returned by ScreenX (this includes the standard options that are always available). Each cond clause specifies where control should pass next.

Get-Menu-Choice (plist)

Allows the user to select one of the icons defined by *plist*. This routine executes a loop which repeatedly reads mouse input. If the user selects one of the icons in plist, the token associated with that icon will be returned. If the user selects a point that is not within any icon, that selection is ignored, and a mouse click is awaited again.

Next-Page (where)

This macro pushes the current page identifier onto the previous page stack, and sets the variable page to the value of where, which is the next page control will pass to.

In file **draw.lsp**:

CarLen (car_type len)

This routine returns the length, in pixels, of a train car of type car_type. The parameter len specifies whether the length of a "long" or "short" car of that type is desired..

DrawCar (x y car_type len wheel_color load_shape load_color)

Draws a train car with it's lower-left corner at absolute position (x y). The *car-type* parameter specifies which type of car to draw: ellipse, box, flat, jagged-top, slanted, closed-box, or coal. *Len* specifies whether a short or long car is to be drawn. *Wheel-color* is either black or white. *Load-shape* is one of square, rectangle, triangle, circle, or ellipse. *Load-color* is one of red, blue, green, or brown.

This routine returns the car's x-value and length.

DrawEngine (x y wheel_color color)

Draws an engine with it's lower-left corner at absolute position (x y). The engine's color is determined by the color parameter, which may be one of red blue, green, or brown. The wheels may be colored either black or white. This routine returns the x-value of the train's position, plus the length of the engine.

Draw-Train (x y train-data)

Draws the train whose specification is in train-data, with its lower-left corner at the point (x y). The parts of the train are drawn by **DrawEngine** and **DrawCar**. The return value of this outine is the x coordinate of the right end of the train.

Empty-Train-Icon (x y &optional fgnd bgnd)

Draws a square the same size as a train icon. The optional parameters bgnd and fgnd can be used to set the color of the background and foreground (outline) of the icon.

EngineLen ()

This routine returns the length of a train engine drawing in pixels.

Lookup (attribute value-name)

Given parameters *attribute*, which is a class name, and value-name, which is an atom specifying an instance of that class, looks up the numeric or string value that this class-instance has.

Train-Icon (x y train-data &optional fgnd bgnd)

Defines a train-icon (train surrounded by pick box) whose upper-left corner is at screen position (x y). The optional parameters *bgnd* and *fgnd* can be used to set the color of the background and foreground (outline) of the icon. The return value of this routine is the x coordinate of the right end of the train.

Train-Len (train-data)

Computes how long the train specified in train-data would be.

In file **interfaces.lsp**:

Remove-Same-States (clusters domain)

Controls the removal of desired selectors from the clusters passed to the function.

Find-Domains-To-Delete (cluster)

Called by Remove-Same-States, this function picks out the elements to be deleted, and returns them in the form of an array of Ts and NILs.

Strip-Lists (cluster del-array)

Performs the actual deletion in accordance with the above functions. Del-array is the array returned by Find-Domains-To-Delete.

In file **ruleout.lsp**: (Copied from Induce)

RuleOut (y in-text &optional font)

Prints out the discovered clustering.

FullRect (c x1 y1 x2 y2)

Generates a solid rectangle of color c, one unit larger in each direction than the input coordinates. By this method, text output given the same x and y parameters will lie completely inside the rectangle.

Inicefileout (x y textfile &optional fontname background foreground)

Displays the contents of *textfile* beginning at (x,y) inside a rectangular box. Colors and fonts are the default unless specified.

In file **text-list.lsp**:

Left-Justify (text-list len)

Reads the text list and returns it in left-justified form, such that no line (element of the output list) exceeds *len* characters. The function will only break a line at a space.

NOTE: This constraint will necessarily be violated if a "word" (string between two spaces) exceeds len characters.

Compress-Text-List (text-list)

Compresses a list of strings into a single long one.

Break-String (string len)

Cuts a string into two, such that the cut is made at a space, and the first portion is as long as possible without exceeding *len* characters. If the first word exceeds len characters, the cut will be made immediately after it.

Break-Word-From-String (string)

Returns a list whose first element is the first word of the input string, and whose second element is the rest of the string.

Join-String (string1 string2)

Creates the sequence string1, string2 in the form of a single string, with the components being separated by at least one space.

String-Length (string)

Returns the length of the input string.

Kill-Spaces (string)

Returns the input string minus any leading or trailing spaces.

In file **vl1toeng.lsp**:

Init-Train-English ()

Reads in the text array of templates for possible clustering outputs. Each template will be accessible by an index number known to the VL1 rule reader.

Cluster-VL1-Eng (items)

Controls the creation of naturally-sounding English based on the VL1 complexes read in.

VL1-String-2 (selector adomain)

Converts a selector (in bit string form) into a VL1 descriptor, based on the association domain parameter.

List-To-String (valuelist)

Creates a string from the elements of valuelist. In the output string, elements of the input list will be separated by commas.

In file **allscreens.lsp**:

HiliteOption (y text)

Draws the option box for text with a brown background instead of a yellow one, to show that it has been selected.

MakeOption (y text)

Outputs material contained in text in an option-box with its upper-left corner at (150, y). The box's length is always 225 pixels, and its width is the width of one line of text, as returned by ChoiceTextSize.

The return value is a list of (length width) for the box.

Pseudo-MakeOption (y text)

Draws only the outline of the option box for text. This allows the pick-list to be reconstructed without disturbing the graphics on the screen.

Screen1 ()

The main screen of the Cluster exhibit. Displays the main menu of user options, reads the user's selection, and returns the icon token selected.

Screen3 ()

"CLUSTER CHALLENGES YOU".

Displays a set of simple geometric shapes, and asks the user to think up some possible groupings for them. Returns the user's selection of one of the standard menu options.

Screen3-Graphics ()

This routine does the actual graphics for Screen3.

Screen4 ()

Displays several possible groupings of the figures from Screen3, and asks the user which one is best. The token for the icon selected by the user is returned. This may be one of the standard options, or the token for one of the three figure groupings.

Screen4-1 ()

If the user picks the first figure clustering from Screen4, control passes to this function. A message is displayed and spoken, saying that this is not the best grouping. The user is then invited to select again.

The return value is the token of the icon selected by the user. Once again this may indicate a grouping or be a standard option.

Screen4-2 ()

The same as Screen4-1, except that the second clustering option has been chosen.

Screen4-3 ()

The third clustering is the "best" one. If the user picks it, control comes here to acknowledge that, and allows the user to continue.

Screen7 ()

"YOU CHALLENGE CLUSTER".

This screen allows the user to select a set of example trains from a given set and then challenge CLUSTER to find groupings for them. The properties of the example trains are defined in **data1.lsp**.

This routine draws a screen with two areas. The top area contains the trains that the user can choose from, while the bottom area is where the user places chosen trains. The routine loops, reading the user's mouse input. If the user selects a train,

that train becomes the "current selection". If the user selects an empty area, the "current selection" (if any) is moved there. If the user picks a menu option, the loop ends, and that option is processed.

When this routine returns, it's value will be one of the standard menu option tokens.

Build-Train-Clustering (choices)

Given the user's chosen trains in choices, builds the call to Cluster-On-Few-Attr, which does the real work. This routine exists separately from Cluster-On-Few-Attr primarily for

the purpose of generalization. The clustering of a different set of examples could be accomplished by a similar routine which would then call Cluster-On-Few-Attr. This routine sets up a list of which attributes of the given trains
CLUSTER should look at each time it is called. This is done, instead of allowing CLUSTER to determine which attributes are most important, to increase processing speed. All the data about the trains are stored in the global variable, ***train-ex***. All the data about the different train attributes are stored in ***train-variables***. This routine returns one of the standard menu option tokens.

In-Data-Area (pos)

Returns TRUE if the y-coordinate in pos represents a location in the data area (top) of the screen.

In-Work-Area (pos)

Returns TRUE if the y-coordinate in pos represents a location in the working area (bottom) of the screen.

Is-Empty-Loc (choice)

Returns TRUE if the token in choice represents an empty icon location. These locations are numbered 100 to "m", but are always greater than 100.

Is-Train (choice)

Returns TRUE if the token in choice represents a train. Trains are numbered from 0 to "n", but are always less than 100.

No-Clustering ()

Tells the user, by text and voice, that at least two examples must be picked in order for CLUSTER to group them.

Screen10 ()

Displays a screen that gives a brief explanation of how CLUSTER creates groups.

In file **help7-x**.lsp:

Help7-1 ()

Prints a message explaining the options available after a clustering has been displayed.

General help for the standard menu options is also given.

Help7-2 ()

Prints a message that gives instructions on how to select trains for the example set (to be clustered). General help for the standard menu options is also given.

6. SPARC ROBOT

6.1. Introduction to SPARC

6.1.1. The Purpose of SPARC/Ex

SPARC/Ex (Sequential Pattern Recognition / Exhibit) has been written specifically to be part of the EMERALD system. While the program was being designed, it was kept in mindthat the users would possess a wide range of technical knowledge. SPARC/Ex has been designed with the aim that anyone from a high school student to a professor in Artificial Intelligence would be able to gain something out of a short interaction with it.

6.1.2. The SPARC/G Methodology

In this section, we will briefly describe the SPARC/G (SPARC/General) methodology. The SPARC/G methodology [Chen, Ko & Michalski 86] enables one to inductively describe and predict discrete processes. SPARC/G also refers to the program which generates rules that predict sequences. SPARC/Ex makes use of the SPARC/G rule generation program when required to predict possible continuations for sequences.

There are three description models used in SPARC/G: the lookback decomposition (decomp) model, the periodic model, and the disjunctive normal form (DNF) model. Given a sequence of objects (events), SPARC/G will generate rules based on user-definable parameters; each rule will follow one of the description models listed above.

6.1.2.1. The Lookback Decomposition (Decomp) Model

This model governs the characteristics of the next object of a sequence based on a previous objects. For example:

((1 (color (1) = red) -> (color (0) = black))
(2 (color (1) = black) -> (color (0) = red)))

is a complete decomp rule which states that if the last object was red, then the next object should be black, and if the last object was black, then the next object should be red. (Note that the "->" is not required in the interpretation and is omitted in the syntax of the rule in SPARC/Ex. Also note the

use of subscripts in the terminology; in general, characteristic(0) will refer to that characteristic of the next object in the sequence, characteristic(1) to the previous object, characteristic(2) to the one before that, and so forth.)

A complete decomp rule is therefore a list of if-then rules, the first part of each consisting of one selector (eg, (color(1) = red)) and the remainder containing a complex of one or more selectors which will specify the characteristics of the next object of the sequence, should the sequence satisfy the antecedent.

6.1.2.2. The Periodic Model

The periodic rule model allows SPARC to describe sequences as repeating subsequences. For example:

((1 (color (0) = red) (parity (0) = odd))
(2 (color (0) = black) (parity (0) = even)))

is a periodic rule with two phases. The rule describes a sequence which is made up of a repetition of a red object with odd parity and a black object with even parity.

6.1.2.3. The Disjunctive Normal Form (DNF) Model

The DNF model uses the Aq algorithm to generate a disjunction of complexes which will describe the sequence and he objects within. So long as the characteristics of each object in the sequence fall into one of the disjuncts of a DNF rule, the rule is a candidate rule for describing the sequence. For example:

(((color (0) = red) (parity (0) = odd)) OR
((color (0) = black) (parity (0) = even)))

is a DNF rule which states that an object in the sequence is either red and has odd parity or black and has even parity..

6.1.3. The Domains

Simple domains have to be chosen for SPARC/Ex as we would neither want to overwhelm the average user with too complicated a domain, nor overwhelm SPARC with the combinatorial

explosions inherent in sequence prediction. Two domains were finally chosen for the initial version of SPARC, geometric figures and playing cards. A third module, involving passages through mined sea channels, was later added.

Geometric figures were chosen for a SPARC module because they were relatively simple; each shape has only two attributes - shape and color. This domain was used in the user's introduction to SPARC due to the ease in which a simple sequence of figures could be used to introduce the user to the basics of SPARC operation. Because a smooth transition to a more difficult domain was desired, an option for the user to be challenged by more complex geometric sequences was also incorporated into SPARC/Ex. These sequences are generated by random selecting a rule template from a collection of common rule patterns. Both Decomp and Periodic rules may be generated; DNF rules were omitted after experimentation determined that their nature too often ran contrary to the patterns perceived by humans.

Playing cards were chosen as SPARC/Ex domain because we wanted to show SPARC/Ex playing the Eleusis card game [Abbott], which tests players' abilities to predict continuations for a given sequence. This domain is implemented in both the "Sparc challenges user" and the "user challenges SPARC" modules. As with the geometric figures, SPARC will challenge the user by generating a sequence based on a rule selected randomly from a rule base. As with the figures, it will then generate three incorrect continuation choices, and one correct one, and then ask the user to pick the most likely next card from the four choices. For these problems, the Decomp and Periodic rules are used, and the important characteristics in the cards are their color and parity. When the user challenges SPARC, these restrictions are lifted; the suits of the cards and whether or not they have prime rank or pictures may be significant, and DNF rules may be generated to describe the user's sequence. The user is able to challenge SPARC by composing cards and placing them on a layout; the user can then ask the system to find rules that could describe the sequence and to generate the "best" next card in the sequence.

Mined channels were chosen as a domain for Sparc/Ex as an example of a "real-world" application. It assumes that a ship is attempting to pass through a channel which has been mined by the enemy. The enemy's beacons contain the key to a secret code which hides the identity of the safe path. Based on known portions of the path, sequence prediction techniques can be used to determine the rest of the safe path. This domain, like the playing cards, allows the user to challenge or be challenged. To challenge the user, SPARC will offer up to three problems given in order of increasing difficulty. As with the figures and cards, these problems will demonstrate Decomp and Periodic rules. Unlike those other domains, the problems for this domain remain

constant, rather than being randomly selected. When challenging SPARC, the user may adjust the beacons as desired. This domain is more complex than the other two in that the beacons have three independent characteristics (shape, color, and left shore/right shore), rather than two. (Note how all characteristics of a playing card in SPARC's model may be determined by its suit and rank.)

6.1.4. The Modules of SPARC/Ex

SPARC/Ex was written with three major modules: the Tutorial Module (TM), the Quiz-User Module (QUM), and the Interactive Prediction Manager (IPM). These modules and their sub-modules, are written in Common Lisp and make use of graphic and voice interfaces which were designed specifically for EMERALD.

The TM provides the user with interactive sessions in which the user learns SPARC/Ex's capabilities. This module includes both the introductory geometric figure problem and a detailed description of SPARC's learning modules and its applications.

The QUM interactively tests the user with problems similar to the ones SPARC/Ex can solve. It is hoped that this module will provide the user with a better understanding of the difficulty of the problems SPARC/Ex can solve.

The IPM allows the user to specify a sequence, and SPARC/Ex is then asked to find rules that govern the sequence. This module is where the inductive capabilities of SPARC/Ex are exhibited.

6.2. The Top-Level Module

The main entry point to SPARC subsystem is the function **sparc-main**. It calls **sparc-top**, which displays the main menu and asks the user to select one from the following options:

SPARC challenges you: Simple sequence --> TM
SPARC challenges you: More complex sequence --> QUM
SPARC challenges you: Card game (advanced) --> QUM
SPARC challenges you: Mined Channel (advanced) --> QUM
You challenge SPARC: Card game (advanced) --> IPM
Find out how SPARC works --> TM
Then it calls the appropriate function.

6.3. The Tutorial Module

The tutorial module is broken down into two sub-modules, the Introduction sub-module and the Tutorial sub-module.

6.3.1. The Introduction Sub-Module

The introduction involves presenting a simple sequence of geometric objects to the user and giving up to two chances to pick the object that continues the sequence. The user is given the choices of returning to main menu or going to more advanced examples after the initial example.

The introduction begins with the function **intro-ex**. If the correct object is chosen, **intro-ex-yes** is called; if not, intro-ex-no is called. **Intro-ex-no** allows the user to make another choice. If the correct choice is made, intro-ex-yes is then called, but otherwise **intro-ex-nono** is called. Intro-ex-yes and intro-ex-nono are the exit points for this example.

6.3.2. The Technical Tutorial Sub-Module

The technical tutorial illustrates the rule models used by SPARC and their complexities. It consists of main function sparc-tutor and sub-functions page1, page2, page3, page4 and page5. These functions display subsequent screens with different tutorial information.

6.4. The Quiz-User Module (QUM)

Quiz-user operates in three domains: geometric shapes, mined channels and playing cards. Quiz-user is entered by calling the function **quiz-user** and specifying a symbol: either 'geo, 'boats or 'cards. Quiz-user contains a number of variables that specify parameters for the user quiz such as the number of guesses and the length of the initial sequence.

Due to the fact that the geometric figure and playing card domain functions were developed together at the beginning of the development of the EMERALD system and the mined channels were added later, the format of the first two are identical, and they share a large amount of

common code. Meanwhile, the mined channel routines behave differently, and are even stored in a separate file, (Full-Path "sparc/boats.lsp"). For these reasons, this domain will be discussed separately from the other two.

6.4.1. Figures and Cards

One of the three rule models is selected to be the model used. Under the present implementation, it will not be the DNF model, for reasons discussed above. A different function handles the random selection of the actual rule for each of the rule models; these functions are called **choose-periodic-rule, choose-dnf-rule** and **choose-lookback-rule**.

One of three functions is then called to handle the presentation of the problem to the user, again depending on the rule model. The three functions are called **periodic-quiz, dnf-quiz**, and **lookback-quiz**, and all three perform similar tasks and have the same parameters.

These functions first generate an initial sequence of specified length. The screen is then set up for the quiz, and the initial sequence is displayed. Four objects (or elements) are generated as user choices, one of which is a correct continuation of the sequence. A loop allows the user to choose one of the four choices.

Whether or not the element is a correct continuation of the sequence, the element is added to the sequence. If the correct element was chosen, the number of correct answers is incremented, and the chosen object takes its position and extends the known sequence. If an incorrect element is chosen, the program displays the object selected in an area indicating that it was an incorrect choice at the time, and adds the correct object (from the given choices) to the sequence and displays it.

Each of the three quiz functions is different because of the variety in the rule models. For example, **lookback-quiz** stores the last object added to the sequence in a variable called **last-card** to aid in the generation of the next object.

6.4.2. Mined Channels

The Mined channel problems are regulated by the function **Challenge-You**. This function displays the introductory screen to the problems in this domain, and if the user elects to continue,

control is passed to **CU2**. This function presents the first of the three mined channel problems. Similarly, **CU3** and **CU4** control the second and third problem screens respectively.

CU2, CU3 and CU4 use a number of common functions to set up their screens. **Canal-Background** paints the portion of the screen on which the straits are displayed a marine blue. The "map" is then drawn by three functions: **Upper-Shoreline**, **Lower-Shoreline** and **Rocks**. **Draw-Beacons** draws a set of beacons on the screen, by making repeated calls to **Draw-Beacon**. This function is passed a position, a shape and a color, and performs the appropriate drawing. **Draw-Path** fills in the path the ship has been known to travel. **Draw-Boat** is passed a position and a color, and draws the ship in the appropriate location. **Draw-Arrows** is also passed a position and a color; this shows the possible paths for the ship to continue from that position, of which the user will presumably select one. Finally, **Explosion** shows a burning and sinking ship, in case the user selected a mined path.

Note that of the above functions, all but Draw-Arrows and Explosion are also used by the You Challenge SPARC portion of the mined channel domain. Two primitives are used by the above functions: **Boat-Icon** and **Arrow-Icon**. These contain the details on drawing each of those objects.

6.5. The Interactive Prediction Module (IPM)

The Interactive Prediction Module (IPM) allows the user to compose a sequence of cards or beacons, and ask SPARC/Ex to find a rule and generate a possible continuation of that sequence. It is run by calling the Lisp function **Test-SPARC**. The IPM uses SPARC/G for the inductive generation of rules, and was designed to be as general as possible (with respect to the use of different domains). However, it is necessary to design some of its routines to be domain-specific. The graphics interface, for example, has to display cards or beacons, and is domain-specific, as are the routines that generate possible next cards or ship paths.

6.5.1. Playing Cards

6.5.1.1. The Interface

There are two screens in the card challenge. The first is the tutorial screen, which introduces the

user to the problem. The second, the Layout screen, is where most of the interaction between the user and SPARC/Ex occurs. In the Layout screen, the user can compose cards and create a sequence, ask SPARC/Ex to generate rules and next cards, and view the rules and next cards generated.

6.5.1.2. Find Rules & Guess Next Card

The title of this section is also the name of the option the user chooses in the Layout screen to ask SPARC/Ex to find a rule to fit the user's sequence and, using that rule, generate a next card for the sequence.

When the user chooses the option, SPARC/Ex does the following:

(1) Instantiate the rules in a rule base (the canned rules) against the sequence and generate a list of rules which can describe the layout.

(2) If no rules were found in step 1, SPARC/Ex calls SPARC/G to inductively generate a rule.

(3) If there are still no rules from step 1 and 2 (which is very unlikely), SPARC/Ex will tell the user that it could not find the rule.

(4) If rules were found, a heuristic weighing function is used to determine the best rule and this rule is used to generate the next card and both the rule and the next card are presented to the user.

6.5.1.3. Canned rules

Because it is difficult to adjust SPARC/G's parameters in such a way that most rules a user might think of can be found inductively, it was suggested that we created a rule base (a list of plausible rules; thus the term "canned rules"), which we did. Thus, for every sequence given to SPARC/Ex

by the user, SPARC/Ex first checks its rule base to see if any of the canned rules matches the given sequence. If one or more are found, SPARC/Ex will not call SPARC/G to inductively generate rules for the sequence (unless all the canned rules have been presented to the user, but the user requests more alternative rules).

6.5.1.4. Interface to Pascal

The present SPARC/G is a Pascal program which requires a "cfile" for input, and outputs its inductively generated rules in "ofile". The cfile is a file which contains a description of the domain, parameters for SPARC/G to do induction (advice parameters, to use SPARC terminology), and the description of the sequence from which rules are to be induced. The **PascalInterface** function in the IPM handles these interactions with SPARC/G (i.e. given the sequence, it generates the cfile, calls SPARC/G, and loads the ofile).

6.5.1.5. Next Card Generation

Because there are only fifty-two cards possible, the next card generation routine, **MakeOneGuess**, simply enumerates each of the fifty two cards, and accepts the first which will satisfy the rule as a continuation of the sequence. If the cardinality of the domain were higher (say 200 or more), a different method of next object generation, using set intersections, might have to be used. The enumeration begins with the Aces, then the Twos, and so on, in the order Clubs, Diamonds, Hearts and Spades.

6.5.1.6. Alternative Rules

There is a list of correctly instantiated rules for the sequence being displayed, ***GoodRules***. Each time the user asks for an alternative rule, a rule is taken out of *GoodRules*, and presented to the user. If *GoodRules* is empty and the rules were previously canned rules, SPARC/Ex will call PascalInterface to generate more rules. However, if the previous rules were generated by SPARC/G (via PascalInterface), SPARC/Ex will tell the user that it has no more alternative rules.

6.5.2. Mined Channels

In many respects, this domain behaves similarly to that of the playing cards. The differences are

listed below:

As with the QUM, the mined channel IPM code is stored separately in the file boats.lsp (or fas, in the case of the compiled version.) It's main controlling function is called **You-Challenge**. It uses the same screen-drawing sub-functions as the mined channel QUM, plus a few other functions allowing for its needs. **Change-Beacon** allows the user to change the color and/or shape of a selected beacon, in order to redefine the problem to the user's specifications. This function uses the subfunctions **Get-Beacon, Beacon-Shapes, Beacon-Colors, Beacon-Color** and **Reset-Beacon**.

Otherwise, behavior will be similar to that of the playing cards domain. There will be a file of "canned rules", an interface to the Pascal SPARC code using a cfile and an ofile, and a demonstration of the discovered rule by SPARC's attempting to move the ship through the channel.

6.5.3. Segmentation

The ability to instantiate rules which involves segments (see [Michalski, Ko & Chen, 1986] for an explanation of segments) was added to SPARC/Ex in its third version. Initially, it was felt that this ability was not crucial. As we began to experiment with different layouts, we realized that it is quite important to have segmented rules in some cases. (Segmented rules allow us to describe sequences such as a sequence of one black card followed by two red cards followed by three black cards, and so on.)

However, because of time limitations and also because we thought that the periodic model was the only model which could benefit from being able to handle segmented rules, we did not extend the TryCannedRules function to handle DNF and decomp segmented rules. This just implies that we cannot have canned DNF or decomp rules that require segmentation. However, the MakeOneGuess routine was developed to handle these rules if they should be generated by SPARC/G.

6.6. Decompositional Rule Translator

This program translates decompositional rules discovered by SPARC into English sentences of

reasonable quality.

6.6.1. Using the Translator

This portion of the program is kept in the file (Full-Path "sparc/peng/fas/98_dnf-decomp.lsp"). To translate a periodic rule, the function **decomp-eng** is called in the following way:

(decomp-eng <rule>)

where <rule> is a defstruct that contains a decompositional rule. The rule body of <rule> is a list of complexes, where each complex describes one if-then rule. Each complex is a list of selectors, where each selector is of the form:

(<attribute-name> (<index>) <relation> <values>)

For example, the following selector:

(color (0) = red)

indicates that the color of the "current card" (index is (0)) is red. And the following selector:

(color (0 1) = true)

indicates that the difference of the color between the current card and previous card is true.

The output of the translator is a list of English sentences (strings) which can be either printed to the screen, sent to the speech synthesizer, or both.

6.6.2. Inside the Translator

The translator separates the selector on the left hand side of the rule from the complex on the right hand side for each complex. Each selector is sent either to **selector-eng** when the selector is of type "(color (0) = red)" (an absolute descriptor) and to **dselector-eng** when the selector is of type "(color (0 1) = true)" (a difference descriptor). This procedure parses the selector into attribute-name, index, relation and values, and translates them individually. For attribute name, this procedure simply uses its symbol-string. Only the "Face" attribute is handled specially in the playing cards domain. **Decomp-eng** then concatenates each string returned by the selector translator as appropriate. The procedure may be used in different domains with little or no change to selector-eng and dselector-eng.

6.7. Disjunctive Normal Form Translator

This portion of the program translates DNF rules discovered by SPARC into English sentences of reasonable quality.

6.7.1. Using the Translator

This routine is kept in the file (Full-Path "sparc/peng/fas/98_dnf-decomp.lsp"). In order to translate a DNF rule, call the function dnf-eng is called in the following way:

(dnf-eng <rule>)

where <rule> is a defstruct that contains a disjunctive normal form rule.

6.7.2. Inside the Translator

The rule body consists of a list of complexes where each complex is disjunctively related. Dnf-eng concatenates each string returned from the selector translator using "and", and concatenates each complex passed through the translator with "or".

6.8. Periodic Rule Translator

The purpose of this portion of the program is to translate periodic rules discovered by SPARC into English sentences of reasonable quality.

6.8.1. Using the Translator

The periodic rule translator is kept in the file (Full-Path "sparc/peng/fas/99_periodic.lsp"). In order too translate a periodic rule, the function vl-english-periodic is called in the following way:

(vl-english-periodic <rule-body> <segmentation-condition>)

where <rule-body> contains a list of complexes that describe a periodic rule, with each complex describing a single phase of the discovered rule. The parameter <segmentation-condition> describes the segmentation condition that was used to discover the rule. A <segmentation-condition> is either NIL or a simple selector. The selector indicates how the layout was segmented

in the first place. For example, the segmentation rule:

(color (0 1) = 0)

indicates the periodic rule was discovered by grouping (segmenting) the layout into cards of same color.

In order to better explain this, some examples are shown here:

```
(vl-english-periodic '(((length (0) = 1)) ((length (0) = 2))) '(color (0 1) = 0))
(vl-english-periodic '(((length (0) = 2) (length (0 1) = 0))) '(color (0 1) = 0))
(vl-english-periodic '(((length (0 1) = -1))) '(color (0 1) = 0))
(vl-english-periodic '(((length (0 1) = 1))) '(color (0 1) = 0))
(vl-english-periodic '(((color (0) = red)) ((face (0) = faced))) nil)
(vl-english-periodic '(((prime (0) = prime-y)) ((rank (0) = jack))) nil)
(vl-english-periodic '(((length (0) <= 2) (dlength (0 1) = -1 between 1))) '(rank (0 1) = 1))
```

The output of the translator is a list of English sentences (strings) which can be printed to the screen and/or sent to the speech synthesizer.

6.8.2. Inside the Translator

The translator simply maps the symbols in a rule-body into their corresponding predefined English translations, and fits them into a template to convert it to standard English. Strings corresponding to an attribute name or relation are defined as a property of the symbol, such as:

```
(setf (get 'prime 'print-string) "rank"),
(setf (get '<> 'print-string) "is not")
```

while the corresponding translations of other symbols are defined as the "value" of the symbol:

```
(defconstant prime-y "a prime number")
```

thus it is possible to translate a selector of the form:

(prime (0) <> prime-y)

into "rank is not a prime number", instead of an awkward expression such as "prime is not true".

In order to adapt this translator to other domains, one must only define a new set of symbols with values or properties corresponding to the attributes present.

6.9. Conclusion

Future work on SPARC/Ex (and also SPARC/G) should include the completion and testing of the routines which handle segmentation.

Rule generation for sequences within segments should also be considered. For example, this sequence cannot be represented within the present SPARC/Ex:

> A repeating sequence of two red cards of even parity,
> followed by two black cards of odd parity.

However, generation of rules of this level of specificity requires an exponential increase of the search space. This might be considered too high a price to pay.

So far, the feedback we have obtained from users of SPARC/Ex has been quite positive. The third version of SPARC/Ex has been completed, and the program is now in the beta test stage. We feel that though there is room for improvement, SPARC/Ex performs up to our expectations. Ultimately, however, public reaction will be the best judge of the success of SPARC/Ex.

7. ABACUS ROBOT

7.1. OVERALL DESCRIPTION

ABACUS has four main screens: two are tutorials designed to familiarize the user with the purpose of ABACUS and the other two are "challenge" screens where either the user or ABACUS is challenged to shoot a ball into a box using a predefined "environment". The first tutorial is a simple example demonstrating the discovery of Ohm's Law given a set of events describing a circuit. The second tutorial is a more complex example using Stoke's Law of Falling Bodies which shows how ABACUS can find multiple equations and preconditions if necessary. The third main screen consists of a cannon pointing towards a wall and a box behind the cannon. The user is asked to select one of the angle ranges shown such that if the cannon were aimed in that range and fired, the ball would bounce off the wall and land in the box. The last screen is identical to the third one except that the user is allowed to change the environment (described later) and challenge ABACUS to shoot the ball into the box using the new environment.

This chapter is divided into four main sections corresponding to the main screens: Ohm's Law, Stoke's Law, ABACUS Challenges User, and User Challenges ABACUS.

7.2. OHM'S LAW

This screen is defined in the function **Screen-2**. After ABACUS draws a table of events, the user is asked to choose an equation that fits the events in the table. Screen-2 gives the user two chances to get the correct equation. After each wrong guess, an explanation of why the guess was incorrect is given. The main objective of this screen is to show how ABACUS can find equations from a set of examples by challenging the user to do the same task.

After this interaction, a brief description of what ABACUS does with the events follows. At this point, if the user chooses the NEXT SCREEN option, a description of ABACUS' general abilities is shown. This is the only entry point to this screen and future work should change this to allow the user to "backtrack" to this screen using the PREVIOUS SCREEN option.

The main function of Screen-2 is drawing the screen. It first defines some strings and mouse locations for the choice boxes and then draws the screen using the functions from the SHARE package. It then loops a maximum of two times and prints the appropriate messages. The

explanation of ABACUS's functions is located in the file **screen-23.txt.**

7.3. STOKE'S LAW

This screen is defined in the function **Screen-6**. It demonstrates how ABACUS can divide recorded events into subsets and find an equation for each subset. In addition, it shows how the AQ algorithm is used to define preconditions for each equation. Screen-6 draws the screen and prints out the text files **screen-61.txt, screen-62.txt,** and **screen-63.txt.** It then calls the function **Balls,** which simulates 3 balls falling down three containers of different media (glycerol, caster oil, and a vacuum) at different rates to demonstrate Stoke's Law. The optional parameters to the function Balls are used to position the balls and cylinders. The coordinates given determine the upper-left corner of the left-most cylinder. The balls are drawn in the cylinders and the simulation is accomplished by repeatedly erasing them and then redrawing them at a new, slightly lower location. The leftmost ball (the ball in the vacuum) is moved down every cycle, the middle ball (the ball falling through glycerol) is moved every other cycle, and the rightmost one (the ball in castor oil) is moved every third cycle, thereby simulating how each media affects the motions. After the simulation, Screen-6 waits for the user to choose an option and then control is transferred appropriately.

7.4. ABACUS CHALLENGES THE USER

This portion of the demonstration is much more complex than the preceding parts. Here, the user is presented with a simple problem of the type ABACUS can solve, and is challenged to do the work that ABACUS would do. The screen consists of a cannon at a fixed distance from a wall and a target box placed in a "random" distance behind the cannon. The user is challenged to choose one of the angle regions shown which will allow the cannon to shoot the ball off the wall and into the box. The user is challenged twice - once for a steel wall and a high explosive power and once for a wooden wall and low explosive power. The wall type and explosive power are part of the "environment" which is formally described in the next section. The possible ball paths for each angle region have been computed in advance and saved in global variables to speed up the simulation. The functions which were used to calculate these paths will be described in the next section.

Screen-7 is the main function for this screen. It calls several subroutines to draw the screens and

the most important of those will be described briefly here. **DrawDiagram** draws the diagram area (for both this screen and the other "challenge" screen), which consists of a light colored area, a wall, a floor, a cannon, and optionally, a target box (the next section describes the situation where the box is not displayed). Several global variables are used in conjunction with the various diagram functions:

DIAGRAMXO -- x-coordinate of the left edge of the diagram
DIAGRAMYO -- y-coordinate of the top edge of the diagram
DIAGRAMWIDTH -- width of the diagram in pixels
DIAGRAMHEIGHT -- height of the diagram in pixels
BALL-RADIUS -- radius of the cannonball in pixels
WALL-RIGHT -- position of the right edge of the wall in pixels
FLOOR-THICKNESS -- thickness of the floor in pixels
CANNON-TIP -- (x y) position of the center of the tip of the cannon in pixels.

Draw-Wall draws the wall in the appropriate color (based on the wall type), and draws the floor. Functions **Draw-Cannon**, **Draw-Box**, **Draw-Ball**, and **Draw-Ball-Path** draw the cannon, box, ball, and ball path respectively. These functions use a variable called ENV which is a structure defining the current environment (described below). **Draw-Diagram-With-Choices** is called to draw the choice areas. These are used to define the angle regions which are used in the interactive session with the user. Each region consists of a triangular-shaped area located between the cannon and wall. This function calls **Draw-Choice-Areas** to draw the choice areas. Other functions used by Draw-Diagram-With-Choices include **Draw-Triangle** and **Draw-Chosen-Area**. Draw-Chosen-Area draws the area chosen by the user. Draw-Triangle draws a triangle whose area lies between the angles ANGLE1 and ANGLE2, where the horizontal is defined as zero degrees. The triangle starts X-DIST pixels to the right of the origin (usually defined as the right edge of the wall) and ends at x=0.

The flow of control begins when Screen-7 calls **Choose-And-Fire** to draw the screen (by calling Draw-Diagram-With-Choices) and to read the user's choice of angle regions. This function then calls **Fire-Cannon** (assuming that the user chose an angle region), which simulates the cannon firing the ball by drawing the ball path (defined in the global variables) for the selected angle region. The desired path is copied since Draw-Ball-Path destructively modifies the path it is given. After the simulation, **Get-Land-Pos** is called to return the location (in pixels) where the ball landed. This location is used to determine whether or not the ball landed in the box. If not, **Ball-Missed** is called to print out appropriate messages and to allow the user to try again. The user is allowed to choose as often as desired, until the correct region is chosen. When the correct region is chosen, **Local-Change-Env** is called to change the wall type and explosive power (to a

predetermined value) and repeat the simulation. After this second time in the loop, the user is shown the equations which describe the flight of the ball using the corresponding environments.

7.5. USER CHALLENGES ABACUS

This portion of the demonstration allows the user to challenge ABACUS by setting certain parameters of the environment and challenging ABACUS to find an equation that it can use to shoot the ball into the target box. Since different environments have different effects on the flight of the ball, ABACUS must generate a new equation for each new environment. **Screen-8** is the main control function for this portion of ABACUS. It starts by setting up the variables *DIAGRAMXO*, *DIAGRAMYO*, *DIAGRAMWIDTH*, AND *DIAGRAMHEIGHT* for the draw-diagram routines, then calling several routines to change the environment and perform the simulations to collect data for ABACUS. After this, ABACUS is called with this data to derive an equation which it then uses to determine the appropriate angle at which to shoot the ball. Before describing these functions, the environment structure should be discussed.

The environment is defined formally as a structure (DEFSTRUCT) which describes various values related to the "environment" in the two challenge screens. It is used in the drawing routines and in the equations used for the "real-world" simulation. The following is a list of the environment variables and their meaning:

WALL-MEDIUM - medium of the wall
VO1 - initial velocity. Translated from the EXPLOSIVE-POWER.
ANGLE1 - initial angle of the cannon.
BOX-LOC - location of the target box.
LAND - location where the ball landed.
CANNON-WALL - distance from cannon to wall.
VO2 - new velocity after the ball hits the wall.
VX1, VY1 - x and y components of the initial velocity.
VX2, VY2 - x and y components of VO2.
ANGLE2 - new angle after ball hits the wall.
TWALL - time for ball to hit the wall from the cannon.
DATA - holds the current events generated by shooting the ball.
This is used by the ABACUS learning system.
DATASET - holds the input for ABACUS.

The dimensions of the variables are all in terms of meters and seconds.

After drawing the screen, Screen-8 calls **Setup-Env** to record the user's changes to the current environment. Setup-Env calls **ChangeEnv**, which uses **Draw-Env-Options** to draw the options for the user. If the response is a standard menu option, SETUP-ENV exits with this choice as the returned value. Other options at this point are moving the target box, changing the wall material or changing the initial velocity. ChangeEnv handles all of these options. Once Draw-Env-Options finishes drawing the screen as chosen, the user input is read and **Update-Box** and **Update-Env** are called to perform error checking and update the box location and the environment.

Draw-Env-Options draws a screen which has one rectangle area for the box location and two areas for the wall material and the initial velocity. The optional parameter SHOW-BOX? is used to indicate whether the routine should draw the rectangle choice area for the box location. It returns two lists; the first one is a list of information on each icon that makes up the table entries and the second is the "pick-list" that was generated when all icons were displayed. Each list consists of the icon token, the name of the function to draw the icon, the (X Y) position of the icon (upper-left corner), and the name of the access function that is used to change the corresponding slot in the Env structure.

As described earlier, Update-Env is called to update the environment with the new values. It is passed the new value (**Choice**), the environment structure (**Env**), and the icon data (**Icon-Data**). This last parameter has the access function needed to reset the value in Env.

When the user has set up the environment, **Challenge-Abacus** is called. This routine redraws the diagram, forces a garbage collection to minimize delays during the simulation, and then simulates the firing of the cannon ball using different preset cannon angles. This will allow for the collection of data pertaining to the flight characteristics of the ball in the current environment. The equations that are used for this simulation are used as follows:

The equations described in this paragraph act as the physical laws of nature to simulate where the ball would go if it was in the "real-world". They are not known by the learning portion of ABACUS. These equations are given the current environment and the time (relative to when the ball was shot), and return the corresponding (X,Y) pair. The coordinates are calculated using the standard equations from physics, with a few exceptions. The flight of the ball is divided into two intervals - from the cannon to the wall ([T0..Twall]) and from the wall to where the ball lands ([Twall..Tground]). The functions corresponding to the first interval include **X1**, **Y1**, and **TWall**. X1 calculates the X position, Y1 calculates the Y position, and TWall calculates the time

for the ball to hit the wall when shot from the cannon. **X2** and **Y2** respectively calculate the same values as X1 and Y1 except for the fact that they operate on the second interval. **VO2** finds the new velocity after the ball hits the wall and **Angle2** returns the new angle. The coordinates are eventually converted to pixels by the **Meters-Pixels** function.

As mentioned in the above paragraph, the equations do not correspond exactly to real-world physics. The reason is the complexity involved when trying to simulate the different effects of varying wall types. The purpose of including adjustable wall types is to show that the less elastic the wall is, the more energy it absorbs, and thus, the shorter the distance the ball will travel. To account for this, coefficient constants were defined which effectively determine that a steel wall absorbs less energy than a brick wall, which in turn absorbs less energy than a wooden wall, and so forth. These constants define the percentage of energy which is not absorbed after the ball collides with the wall. This is not an exact simulation of what would actually occur, but it suffices for the purposes of the demonstration.

After Challenge-Abacus performs the simulations to collect data for ABACUS, it calls ABACUS via several interface functions to generate the equation. **Find-Eqn** takes the data generated from the simulation (which is stored in the Env-Data slot of the environment structure), and calls ABACUS with this data via the function **Run-Abacus**. The results are stored in **Env-Dataset**. Once ABACUS returns the discovered equation, **Retrieve-Angle** is called to get the angle for the cannon. It does this by retrieving the equation (**Retrieve-Formula**), and then passing this equation and the environment to **Calculate-Angle**. Calculate-Angle is an ad-hoc function which simply initializes the angle to 20.0 degrees (the minimum value) and then increments it until the resulting value from the equation with that angle value is approximately equal to the constant generated by ABACUS. This approach is necessary since a full-sized problem solver would be required otherwise due to the fact that no assumptions were made regarding the type of equations generated.

With this new angle value, Challenge-Abacus calls **New-Box-Pos** in order to make sure that the box location chosen by the user is within the area that may be reached by the cannon. For certain environment values, the angle required to hit the box may be below the minimum or above the maximum. If this is the case, then New-Box-Pos asks the user to choose a new location within a rectangular box that it draws. The dimensions of this box are a parameter whose x-constraints represent the minimum and maximum distances the cannon can fire in the current environment.

The next step is the simulation of the new cannon angle, showing that the equation generated by

ABACUS is accurate enough to hit the box. This simulation is done twice. At this point, the user can either challenge ABACUS again, see the equation generated, or choose one of the standard menu options. If the user wishes to see the equation, **Show-New-Eqn** is called to draw the new screen using function **Draw-Eqn-Screen**, and then calls **Show-Eqn** to retrieve the constant associated with the equation. This portion of the presentation is a bit artificial, since these functions already know the general form of the equation, implying that the only thing that changes from one equation to the other is the value of a constant.

7.6. List of ABACUS Functions

The following is a list of the functions used by the graphics routines in ABACUS, and a brief description of the function each performs:

Abacus-Main ()
Main routine for "Abacus". Calls Screen-1 to display the main menu and get a response from the user. It then dispatches control to the proper screen routine, based on that response.

Ball-Hit (hit-count env StdPlist SelectAreaY0 SelectAreaHeight)
Tells the user that the ball landed in the target-box. If this is the first environment, change the environment with 'local-change-env', and allow the user to continue. Otherwise, display the equations that abacus would have derived for the two demonstrations that the user has experienced. This routine returns the user's next menu choice.

Ball-Missed (StdPlist file X0 Y0 Width Height)
Tells the user that the ball missed the target-box. Icons are then displayed allowing the user to try again, or continue on to Screen-8. The user's menu choice is returned.

Balls ([(x0 0) (y0 0)])
This routine simulates the falling balls in different media for the demonstration of Stoke's law. The optional parameters are used to position the ball and cylinder arrangements for the demonstration. The coordinates given determine the upper-left corner of the leftmost cylinder.
The balls are drawn in these cylinders, and the dropping is accomplished by erasing them, and redrawing them at a new location. Each ball is re-drawn before the next one is erased each cycle.
The leftmost ball ("vacuum") is moved down every cycle, the middle ball ("glycerol") is moved every second cycle, and the rightmost ball ("castor oil") is moved every third cycle to show the effects of the different media densities.

BoxWord (x0 y0 text [bgnd[fgnd[outline[x-span[y-span]]]]])
Outputs 'text' to the screen, centered in a box whose upper-left corner is at position (x0, y0). The box defaults to a size of 130 pixels long, and tall enough for a single line of text. The box is yellow with black text and an outline of coral. Any of these
defaults may be changed by supplying one of the optional arguments to the function. These optional arguments are background-color, foreground-color, outline-color, x-span and y-span.

BoxWordSize (text [x-span])
Returns the size of a word-box containing 'text' in the form (x y). The x-span and y-span of the box may be specified, as per BoxWord.

Build-Path (file-name)
Appends 'file-name' to the partial path defined in the global variable *PATH* to give an absolute path to the file. Thus (Build-Path "foo") is equivalent to (Full-Path "abacus/foo")

Challenge-Abacus (env StdPlist X0 Y0 Width Height)
When the user has configured a satisfactory environment, control is passed to this routine.

This routine re-draws the diagram so that only actual pieces of the diagram are displayed, forces a garbage-collection to ensure smooth "animation", and then simulates the firing of the cannon at several pre-set angles. The results of these simulations are recorded and passed to the abacus program which derives a rule that will compute the landing location of the cannonball given the angle of the cannon for this environment. Once this law is known, it is used to determine what angle at which to aim the cannon at in order to fire the cannonball into the target box. In the event that the target box is outside of the possible range of the cannon, the user is asked to choose a new location for the box in the reachable area. Changing any other parameter in the environment would invalidate the rule found by the Abacus program, and necessitate a re-running of the simulation.
Once the rule is found and demonstrated, the user is presented with the options of seeing the equation derived by the Abacus program, or continuing to challenge Abacus.
The routine returns the menu-token for the option selected by the user. The user may select standard menu options at any time input is requested.

ChangeEnv (env StdPlist Y0)

This routine controls the changing of the environment parameters by the user. It calls Draw-Env-Options to output the rectangle for possible target box placement and the tables of possible wall media and cannon explosive power. User input is then read. If the user selected a point that is not within an icon, Update-Box is called to see if a valid new target box location has been selected. If an icon token was returned, it is checked against the data for the environmental icons and Update-Env is called if a match is found. In all other cases, the user picked a standard menu option, or "Challenge Abacus", so the icon token will be passed back to Setup-Env.

Choose-And-Fire (StdPlist env angle-list path-list SelectAreaX0 SelectAreaY0 SelectAreaWidth SelectAreaHeight intro-file intro-msg?)

This routine causes the diagram-area to be drawn, and then reads the user's response. If the response is a standard menu option, the routine returns with this choice as it's value.
If the user selected one of the icons specifying an angle region for the cannon, Fire-Cannon is called to fire the cannon, and the routine returns HIT or MISSED to indicate whether the ball landed in the box or not.

ClearMidScreen (&optional color-index)

Clears the area in the center of the screen (i.e. the region from below the title line to above the standard options). The standard menu background rectangle is redrawn on a background that is black by default, but which can be reset by calling the routine with a color index .

ClearSelectArea (X0 Y0 Width Height)

Clears the area between the bottom of the diagram area and the standard options. This area is defined to have its upper-left hand corner at (X0 Y0) and extend*Width* pixels in the x-direction and *Height* pixels in the y-direction.

ConsEnd (list element)

Adds an element to the end of a list, and returns the new list. This function is non-destructive.

Distance (pt1 pt2)

Computes the distance between the points pt1 and pt2, which are both (x,y) pairs.

Draw-Ball (pos)

Draws the ball at the given position pos, which is given in terms of pixels.

Draw-Ball-Path (env [ball-path [leave-ball]])

Displays the path of the ball fired from the cannon. This path may be defined in one of two ways: If the optional parameter *ball-path* is given, it specifies the path of the ball as pairs of (x y) coordinates; otherwise the ball's path is derived from the data in env by Get-Xy-Pairs. The optional parameter *leave-ball?* may be used to keep the ball from being erased after it is drawn, thus displaying the entire path.

Draw-Box (env [box-color])

Draws the target box at the location specified in env (in terms of meters). An optional parameter can be used to override the default box color.

Draw-Cannon (env [barrel-color [wheel-color]])

Given the cannon to wall distance (given in meters) and cannon angle from env, draws the cannon at the correct position in the diagram. Optional parameters can be used to choose the color of the cannon and the wheel. The global variable *cannon-tip* is set for routines which need to know the current cannon tip location in pixels.

Draw-Choice-Areas (env angle-list)

Controls the drawing of the triangular angle regions that the user may choose from. These regions start at the tip of the cannon and end at the wall. Each region is identified by a number placed within the wall directly to the left of the region. The function Draw-Triangle is used to actually draw the regions.

Draw-Chosen-Area (env angle-list choice)

Calls Draw-Triangle to draw just the angle region chosen by the user.

DrawDiagram (env [no-box])

Draws the diagram area for screens 7 and 8, consisting of a light-colored area, a wall, a floor, a cannon, and optionally a target box. Several global variables are used in conjunction with the various diagram functions. They are:

DiagramX0	-- x-coord. of left edge of diagram
DiagramY0	-- y-coord. of top edge of diagram
DiagramWidth	-- width of diagram in pixels
DiagramHeight	-- height of diagram in pixels
ball-radius	-- radius of cannon ball in pixels
wall-right	-- position of the right edge of the wall in pixels
floor-thickness	-- thickness of the floor in pixels
cannon-tip	-- (x y) position of the center of the tip of the cannon in pixels. This is set by Draw-Cannon.

Draw-Diagram-Area (env)

Draws the light colored rectangle of the area, with upper left corner at (*DiagramX0*, *DiagramY0*). A dark border is also drawn around the diagram area.

Draw-Diagram-With-Choices (env angle-list file Y0 intro-msg?)

Draws the diagram-area with the four triangular angle areas that the cannon may be aimed into. If this is the first time this part of the demonstration has been entered (since the invocation of Screen-7), a title and some labels such as "target box" are added to the diagram. A message explaining what to do is then printed, and icons for each of the four region choices are created. The return value of this routine is a pick list of these four icons.

Draw-Env-Options (env Y0 [show-box?])

This routine draws tables of possible values of the wall medium and cannon explosive power, and highlights the current values from the env parameter. If show-box? is non-nil, a rectangular area where the target box may be placed
is also displayed. Draw-Env-Options returns a two element list. The first element is a list of information on each icon that makes up the table entries. Each icon information list consists of the icon token, the name of the function to draw the icon, the (x y) position of the icon (upper-left corner), and the name of the access function that is used to change the corresponding slot in the env structure. The second element of the return value is the pick-list that was generated when all these icons were displayed.

Draw-Eqn-Screen (eqn &optional is-file?)
(Defined in screen-7, but not used there)
This routine will build a page of information explaining the equation which was derived by Abacus. The equation can be delivered to this routine as a string in the eqn parameter (in which case, the parameter 'is-file?' should be absent or nil), or
as a one line file whose name is passed as eqn (with is-file? non-nil).

Draw-Triangle (angle1 angle2 x-dist [marker])
Draws a triangle whose area lies between the angles angle1 and angle2 (where the horizontal is zero degrees). The triangle starts x-dist pixels to the right of the origin (usually defined as the right edge of the wall) and ends at x = 0. An optional parameter may be used to define a single character string marker for this triangle which will be placed a small distance (16 pixels) to the left of the left end of the triangle. This distance places the marker centered within the wall, if the wall is there.

Draw-Wall (env)
Draws the wall in the appropriate color for the current wall-medium as defined in environment. This routine also draws the floor in the same color as the diagram-area border.

Erase-Ball (pos)
Erases the ball at the given position *pos* (which is in pixels) by re-drawing it in the background color.

Erase-Cannon (env)
Redraws the cannon at the current location in the background color so that the cannon is effectively erased.

Fix-Path (env path)
Alters the ball's path so that it does not start before the tip of the cannon, and that it does not continue through the target box.

Fire-Cannon (env choice angle-list path-list X0 Y0 Width Height)
Redraws the diagram-area, with only the selected angle region displayed, then fires the cannon by drawing the ball-path for the selected angle region. The desired path is copied with Copy-Tree because Draw-Ball-Path destructively modifies the ball-path it is given, and altering it would change the results the next time this constant path was used.

FullRect (color x1 y1 x2 y2)
Fills a rectangle that is one pixel larger than the area given with the color given.

GeneralBlockRight ()
Returns the size of a general block minus the x-position of the left edge of the block.

Get-Box-Width ()

Returns the half-width of the target box in pixels. (The box extends box-width units on either side of the origin.) This routine was written so that only one constant needed to be changed to change the box width for the entire ABACUS demonstration. The box-width constant is given in meters.

Get-Box-Locs (path-list)

Calculates where the ball will come to rest (in pixels) for each path in path-list and builds a list of all these landing points as possible target box locations.

Get-Land-Loc (path)

Given a ball-path, determines the (x,y) coordinates (in meters) where the ball will come to rest.

Get-Land-Pos (path)

Given a ball-path, determines the (x,y) coordinates (in pixels) where the ball will come to rest. The function Get-Land-Loc is used to get the coordinates in meters.

Get-Menu-Choice (plist &aux choice)

Reads the user's mouse selection, and disregards any selections that do not reference icons (i.e. any selection that returns a screen position rather than an icon reference).

Hit-Box? (env &optional land-loc)

If the cannon ball has hit the target box in the current environment (or the landing position land-pos would do so), this routine returns the atom HIT, otherwise it returns the atom MISSED.

In-area (x y x0 y0 x1 y1)

Determines if the point (x, y) is within the rectangle defined by the lower-left corner (x0, y0) and the upper-right corner (x1, y1).

Local-Change-Env (env StdPlist SelectAreaY0 SelectAreaHeight)

After the user successfully shoots the cannon ball into the box, this routine is called to change some of the parameters of the environment in order to give the user another problem to attempt. The routine Draw-Env-Options is called to display tables of the current environment, and then Update-Env is called to change the env structure, as well as the displayed tables.

A new icon is generated that asks if the user wants to try this environment. The user's response is then read, and returned by this routine.

Meters-Pixels (pos)

Converts pos from meters to pixels. Both pos and the return value are (x,y) pairs.

New-Box-Pos (env area-dim StdPlist file X0 Y0 Width Height)

If the box location chosen by the user is not within the area that can be reached by the cannon with the current environment, this routine is called to allow the user to choose a new box location. The parameter *area-dim* contains a list of least-x and greatest-x locations that the cannon can reach. The routine displays a box around this area and asks the user to select a location within it. The box is then redrawn at the new location. The box can be moved as many times as the user desires. Once the box is moved into the area reachable by the cannon, a menu icon is displayed that gives the user the option of continuing with the demonstration using the current target box location. Any selection of the standard menu

options causes this routine to return that menu token to Challenge-Abacus.

Past-Tip (pos)
Computes whether a given (x y) position (in pixels) is past the tip of the cannon.

Pixels-Meters (pos)
Converts pos from meters to pixels. Both pos and the return value are (x,y) pairs.

Restore-Ball (env)
Restore-Challenge (env title SelectAreaY0 SelectAreaHeight)
Restore-Hit (env title SelectAreaY0 SelectAreaHeight)
Restore-Missed (env title SelectAreaY0 SelectAreaHeight)
Restore-Set-Up (env title SelectAreaY0 SelectAreaHeight)
Restore-See-Eqn (env title SelectAreaY0 SelectAreaHeight)
Restore-Std-Options (title)
Restore-Try-Again (env title SelectAreaY0 SelectAreaHeight)
The Restore-* functions are used to return the screen to the state it was in before the user selected HELP to the best degree possible. If the pages of a screen are changed, these routines must be updated to reflect this change so that the before-HELP display matches the after-HELP display.

Rotate-Pt (pt sine cosine)
Rotates an (x,y) point through an angle around the origin, where sine and cosine are the sine and cosine of the desired angle of rotation.

Say-Hit-Or-Missed (env)
Tells the user (by voice) whether the ball hit the target box or missed it.

Screen-1 ()
Displays the main abacus menu, gets a choice from the user, and returns that choice.

Screen-2 ()
Demonstrates to the user the use of a simplified example of abacus to discover Ohm's law.

Screen-4 ()
Displays a page briefly describing Abacus's abilities. Currently only reachable from the "Ohm's law" example (Screen-2).

Screen-6 ()
Demonstrates an experiment that could be used by ABACUS to derive Stoke's law for falling bodies. This demonstration shows that ABACUS may find several rules relating to a single experiment.

Screen-7 ()
This screen presents the user with a simple problem of the type abacus could be used to solve, and challenges the user to do the work that abacus would do. The user is presented with a cannon placed a fixed distance from a wall and a target box placed a "random" distance past the cannon. The user is challenged to pick the proper angle for the cannon to be aimed at, so that the ball will bounce off the wall and land in the target box. The possible ball paths for each angle region have been computed in advance and saved in global variables in order to enable this screen to respond as quickly as possible. This screen sets up the global variables *DiagramX0*, *DiagramY0*, *DiagramWidth*, and *DiagramHeight* for the diagram drawing routines in order to place the diagram area

where desired.
There are several actual "pages" to this screen, including a HELP page. When the user chooses help, the routine displays the help message that is most appropriate for where the user is in the demonstration. When the user exits help, the routine attempts to restore the display to a state that resembles the pre-help display as closely as possible. This is accomplished by calling the various Restore-* routines after the HelpScreen function returns.

Screen-8 ()

This screen presents essentially the same environment as screen-7 (cannon, wall, target box, etc.), except that this time the user may manipulate certain elements of the environment, and then challenge ABACUS to shoot the cannonball into the target box.
Once the environment has been set up, the actual ABACUS program is called with the results of several test shots of the cannon.
The Abacus program computes an equation for where the ball will land given the current environment, as a function of the cannon angle. This equation is then used to shoot the cannonball directly into the target box. The user may repeat this challenge as many times as is desired.
This screen sets up the global variables *DiagramX0*, *DiagramY0*, *DiagramWidth*, and *DiagramHeight* for the diagram drawing routines in order to place the diagram area where desired.
There are several actual "pages" to this screen, including a HELP page. When the user chooses help, the routine displays the help message that is most appropriate for where the user is in the demonstration. When the user exits help, the routine attempts to restore the display to a state that resembles the pre-help display as closely as possible. This is accomplished by calling the various Restore-* routines after the HelpScreen function returns.

Setup-Env (env StdPlist SelectAreaY0 SelectAreaHeight intro-msg?)

This routine causes the diagram-area to be drawn, and then calls ChangeEnv to read the user's mouse selection. If the response is a standard menu option, the routine returns with this choice as its value. Other options include moving the target box or changing the wall medium and explosive power of the cannon. ChangeEnv handles all these options. The first time this routine is called (per invocation of Screen-8) an introductory message explaining what is happening is displayed.

Show-New-Eqn (env StdPlist X0 Y0 Width Height)

This routine builds the equation that Abacus found for shooting the cannonball into the target box in the current environment. The user is then given the option of challenging Abacus with another environment, or selecting one of the standard menu options. The function Draw-Eqn-Screen is called to create most of this page.

Trans-Pixels (pos)

Translates an (x,y) location in pixels to be offset that distance from (*DiagramX0*,*DiagramY0*) instead of from (0,0).

Update-Box (pos env)

The screen-location in pos is checked to see if it falls within the target box location area. If it does, it defines the new center of the target box. The old box is erased, the location of the box is changed in env, and the new box is drawn at the new location.

Update-Env (choice env icon-data)

Given the choice parameter (an icon token returned when the user selected an icon), this

function changes the value of the env-slot chosen to be the value desired. The value of choice is the new value the env-slot is to have. The data associated with the icon which would return choice is looked up in icon-data. This gives the access function needed to reset the value in env. If the current value of the env-slot is different from choice, it is de-highlighted. The icon associated with choice is then highlighted, and the slot in env changed to reflect the user's selection.

8. AUTOEVALUATOR

8.1. Introduction

The autoevaluator is a mechanism for providing a feedback for the developers from the users about the EMERALD system. The autoevaluator consists of two component systems, the **passive** system and the **active** system, as well as a set of functions which run the autoevaluator.

Passive variables, or counters, simply log when certain portions of the program are reached. With these, we hope to get a count not only of how many people are using EMERALD, but also which modules they have a tendency to view. Currently, six passive variables are in use, with the descriptive names: EXHIBITSEEN, AQSEEN, INDUCESEEN, CLUSTERSEEN, SPARCSEEN, and ABACUSSEEN.

Active, or input variables, are multi-valued, and require input (or the absence of it) by the user in order to register. Each is being implemented as a property list with three properties: YES, NO and ABSTAIN. YES indicates how many times the users displayed a liking for that which was asked about; NO counts the dislikings, and ABSTAIN counts the number of times the users did not answer the question, either by letting the exhibit time out, or by choosing one of the standard menu selections instead.

Only two active variables are currently implemented, EXHIBITLIKED and INDUCELIKED. When a user selects QUIT THE EXHIBIT at some point, if EXHIBITLIKED is activated, the system will ask whether the user enjoyed the exhibit. When INDUCE finds and displays a rule, if INDUCELIKED is activated, the system will be asked whether the user liked the rule. Future development will allow for the implementation of AQLIKED, CLUSTERLIKED, SPARCLIKED, and ABACUSLIKED. These, like INDUCELIKED, will test user reactions to the rules and descriptions generated during the challenge stages. The reason only one of the rule liking variables has been implemented so far is that it is a non-trivial programming problem to incorporate it into the existing code, so that the EMERALD administrator may turn the inquiry mechanism on or off at will. Ultimately, all will be installed.

8.2. Files and Functions Used by the Autoevaluator

The autoevaluator system uses two files to store its data, evaldata.lsp and evaldata.init. The

former holds the current values of all counters, as well as keeping track of which subset of the counters is currently activated (see below). The function **WriteEvalData** takes the updated values for these out of memory, and rewrites this file in LISP-readable form. In turn, function **LoadEvalData** initiates the loading of this file's contents into memory. Currently, WriteEvalData operates each time an EMERALD run is exited or the closing screen is shown. There is no way to account completely for an abnormal EMERALD exit, but it is hoped that this frequency of update minimizes the potential amount of lost data, without subjecting the user to too many delays. LoadEvalData only needs to be run at the start of an EMERALD session.

The file evaldata.init is similar to evaldata.lsp, but all counters are set to zero, and the default activated set is indicated. Copying this file into evaldata.lsp therefore reinitializes the autoevaluator.

WriteEvalData, LoadEvalData, and all other autoevaluator functions are stored in the file (Full-Path "share/autoeval/autoeval.lsp"). The other main autoevaluator functions are as follows:

AskUser is the function which handles the i/o for the active variables. **ViewEvalData** produces a hidden screen which displays all values of all autoevaluator variables which are currently activated. This function is invoked by going to the EMERALD screen, moving the X to Emerald's mouth, and hitting the SELECT button. The screen then displayed also shows the 10 digits, the sequential selection of which allows the user to enter a five-digit code number. If incorrectly entered, control goes back to the EMERALD screen, while if correctly entered, function **SpecialPage** is called, which generates yet another hidden screen. This one allows the user to toggle both active and passive variables on or off, thereby altering the set of activated variables. If a variable is not activated, its counters will not operate until it is reactivated.

All source and data files are found in the directory, (Full-Path "share/autoeval").

REFERENCES

General:

Dietterich, T. G. and Michalski, R. S., "A Comparative Review of Selected Methods for Learning from Examples," Chapter 3 in Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, Tioga Publishing Company, Palo Alto, CA, 1983, pp. 41-82.

Kodratoff, Y. and Michalski, R. S. (Eds.), *Machine Learning: An Artificial Intelligence Approach, Vol III*, Morgan Kaufmann Publishers, San Mateo, CA, 1990.

Lenat, D., "AM An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search," Computer Science Department, Rept. STAN-CS-76-750, Stanford University, Stanford, CA, 1976.

Michalski, R. S., "Pattern Recognition as Rule-Guided Inductive Inference," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Vol. PAMI-2, No. 4, pp. 349-361, July 1980.

Michalski, R. S. (Ed.), Proceedings of the International Machine Learning Workshop, University of Illinois Allerton House, Urbana, IL, June 22-24, 1983.

Michalski, R. S., "Concept Learning," *Encyclopedia of Artificial Intelligence*, S. Shapiro ed. John Wiley and Sons Publishers, New York, NY 1987, pp. 185-194.

Michalski, R. S. and Chilausky, R. L. , "Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *International Journal of Policy Analysis and Information Systems*, Vol. 4, No. 2, pp. 125-161, 1980.

Michalski, R. S., Carbonell, J. and Mitchell, T. (Eds.), *Machine Learning: An Artificial Intelligence Approach*, TIOGA Publishing Company, Palo Alto, CA, 1983.

Michalski, R. S., Carbonell, J. and Mitchell, T. (Eds.), *Machine Learning: An Artificial Intelligence Approach, Vol II*, Morgan Kaufmann Publishers, Los Altos, CA, 1986.

Mitchell, T., Carbonell, J. and Michalski, R. S. (Eds.), *Machine Learning: Guide to Current Research,* Kluwer Publishing Co., 1986.

Mitchell, T. M., Keller, R. M. and Kedar-Cabelli, S. T., "Explanation-based Generalization: A Unifying View," *Machine Learning*, pp. 47- 80, Vol. 1, No. 1, 1986.

Rose, D. and Langley, P., "STAHLp: Belief Revision in Scientific Discovery," *AAAI-86, Fifth National Conference on Artificial Intelligence*, Vol. I, pp. 528-532, Philadelphia, PA, August 1986.

Winston, P. H., "Learning Structural Descriptions From Examples," Tech. Report AI TR-213, MIT, AI Lab, Cambridge, MA, 1977.

Winston, P. H., "Learning by Augmenting Rules and Accumulating Sensors," Chapter in *Machine Learning: An Artificial Intelligence Approach, Vol. II*, Morgan Kaufmann Publishers, Inc. 1986.

For AQ:

Michalski, R. S. and Larson, J. B., "INCREMENTAL GENERATION OF VL1 HYPOTHESES: The Underlying Methodology and the Description of Program AQ11," ISG 83-5, UIUCDCS-F-83-905, Department of Computer Science, University of Illinois, Urbana, IL, 1983.

Michalski, R. S., Mozetic, I., Hong, J. and Lavrac, N., "The AQ15 Inductive Learning System: An Overview and Experiments," Report No. UIUCDCS-R-86-1260, Department of Computer Science, University of Illinois, Urbana IL, July 1986.

Exemplary applications:

Michalski, R. S., Mozetic, I., Hong, J. and Lavrac, N.,"The Multi-Purpose Incremental Learning System AQ15 and Its Testing Application to Three Medical Domains," *Proceedings of AAAI-86, Fifth National Conference on Artificial Intelligence*, Vol. 2, pp. 1041-1045, Philadelphia, PA, August 1986.

Mozetic, I., "Compression of the ECG Knowledge-base Using the AQ Induction Learning Algorithm", ISG 85-13, UIUCDCS-F-85-943, Department of Computer Science, University of

Illinois, Urbana, IL, March 1985.

For INDUCE:

Bentrup, J., Mehler, G. and Riedesel, J., "INDUCE.4: A Program for Incrementally Learning Structural Descriptions from Examples", Technical Report UIUCDCS-F-87-958, Department of Computer Science, University of Illinois, Urbana, IL, 1987.

Hoff, W. A., Michalski, R. S. and Stepp, R. E., "INDUCE 2: A Program For Learning Structural Descriptions From Examples," Technical Report UIUCDCS-F-83-904, Department of Computer Science, University of Illinois, Urbana, IL, September, 1983.

Exemplary applications:

Lewis, C. M., "Identification of Rule-based Models," Report No. 86-5, Center for Man-Machine Systems Research, Georgia Institute of Technology, May 1986.

For CLUSTER:

Michalski, R. S. and Stepp, R. E., "Automated Construction of Classifications: Conceptual Clustering Versus Numerical Taxonomy," IEEE Transactions on Pattern Analysis and Machine Intelligence, 1983.

Michalski, R. S., and Stepp, R. E., "Clustering," *Encyclopedia of Artificial Intelligence*, S. Shapiro ed. John Wiley and Sons Publishers, New York, NY 1987, pp. 103-111.

Michalski, R. S., Stepp, R. E. and Diday, E., "A Recent Advance in Data Analysis: Clustering Objects into Classes Characterized by Conjunctive Concepts," in *Progress in Pattern Recognition*, Vol. 1, L. N. Kanall and A. Rosenfeld (Eds.), New York: North-Holland, pp. 33-56, 1981.

Stepp, R. E., "Conjunctive Conceptual Clustering: A Methodology and Experimentation," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana IL, June 1984.

For SPARC:

Abbott, R., "The new Eleusis", available from Abbott at Box 1175, General Post Office, New York, NY 10001 ($1.00).

Dietterich, T. and Michalski, R. S., "Learning to Predict Sequences," Chapter in *Machine Learning: An Artificial Intelligence Approach Vol. II*, R. S. Michalski, J. Carbonell and T. Mitchell (Eds.), Morgan Kaufmann Publishers, Los Altos, CA, pp. 63-106, 1986.

Michalski, R. S., Ko, H. and Chen, K., "SPARC/E(V.2), An Eleusis Rule Generator and Game Player," ISG 85-11, UIUCDCS-F-85-941, Department of Computer Science, University of Illinois, Urbana, IL, February 1985.

Michalski, R. S., Ko, H. and Chen, K., "Qualitative Process Prediction: A Method and Program SPARC/G," *Expert Systems*, C. Guetler, (Ed.), Academic Press Inc., London, 1986.

For ABACUS:

Falkenhainer, B., "ABACUS: Adding Domain Constraints to Quantitative Scientific Discovery," ISG 84-7, UIUCDCS-F-84-927, Department of Computer Science, University of Illinois, Urbana, November 1984.

Falkenhainer, B. and Michalski, R.S., "Integrating Quantitative and Qualitative Discovery: The ABACUS System," in *Machine Learning: An Artificial Intelligence Approach, Volume III*, Kodratoff and Michalski, Eds., Morgan Kaufmann Publishers, San Mateo CA, 1990.

For EMERALD:

Kaufman, K., Michalski, R. S. and Schultz, A., "EMERALD 1: An Integrated System of Machine Learning and Discovery Programs for Education and Research: User's Guide", *Reports of the Machine Learning and Inference Laboratory*, MLI 90-7, Center for Artificial Intelligence, George Mason University, Fairfax VA, 1990.

Michalski, R. S., "Machines That Learn and Discover", Artificial Intelligence Center, George

Mason University, Fairfax VA, January 1988.