Some Tools for Visualizing Icon Programs

Gregg M. Townsend

Department of Computer Science, The University of Arizona

Introduction

This document describes a set of tools for visualizing Icon programs. Each reads an *event stream* [1] produced during the execution of an Icon program. Most tools produce color displays under the X Window System using an extended version of Icon [2]. Some annotated examples appear in the Appendix.

Common Characteristics and Options

Some tools use program line numbers recorded in the event stream. Meaningful results are unlikely for subject programs built from multiple source files.

Some tools use ‘‘clock ticks’’ recorded in the event stream. These clock ticks are based on the Unix system clock, which typically ticks every 4 to 20 milliseconds; on a Sun 4 it ticks every 10 milliseconds.

All display programs accept the following options in addition to those appearing in their individual descriptions:

–B *color*	background color
–F *color*	foreground color
–L *label*	window label
–X *xpos*	window x position
–Y *ypos*	window y position
–W *width*	window width
–H *height*	window height
–M *margin*	margin within window
–G [*w*x*h*][+*x*+*y*]	window geometry in X form

After a display program completes, its window can be deleted by typing ^C, ^D, DEL, Q, or q in the window. The display programs were developed and tested on color screens; usability on monochrome screens is unknown.

anim: An Animated Program Listing

anim displays a miniature program listing, highlighting each line of code as it is executed.

Two accompanying barcharts display execution profiles. The one on the extreme left shows the number of clock ticks attributable to each source line. The second chart shows the number of times each line was executed.

A chart to the right of the listing shows a time-based history similar to that of the **roll** program.

Synopsis: anim [*options*] *sourcefile*.icn [*eventstream*]

–d *n*	decay interval
–b *n*	barchart length (0 to disable)
–z *n*	history length (0 to disable)
–t *n*	ticks per history pixel
–s *n*	vertical line spacing, in pixels
–w *n*	width of one character, in pixels
–h *n*	height of one character, in pixels
–p *n*	pointsize for text display
–S *n*	spacing between display sections
–P *color*	program text color
–C *color*	comment color
–A *color*	active text color
–O *color*	old (past) text color
–F *color*	barchart and history color
–R *color*	barchart and history background color

–p works only under OpenWindows and overrides –w and –h. Setting either –p or –s establishes good defaults for the other size parameters.

melody: Play the Tune of an Icon Program

melody translates each line-number event into a musical note in a three-octave range, playing the results through the speaker on a Sparcstation. **melody** does not actually drive the audio port directly; it pipes directives into a separate **piano** program, which is not described here.

Synopsis: melody [*options*] [*eventstream*]

–t	do not pipe output through the **piano** program
–w	use whole-tone instead of chromatic scale
–i	invert the direction of the scale
–p *n*	pitch of the highest note, in Hz
–m *n*	meter, in beats per minute
–l *n*	highest line number expected

roll: A Piano-Roll Display of Flow Control

roll displays a chart recording a time-history of program execution by line number. Unless –o is given, each pixel column marks the lines executed during one clock tick. Gaps indicate garbage collections.

Synopsis: roll [*options*] [*eventstream*]

–l *n*	highest line number expected
–o *n*	overlap *n* line-number events per column

vitals: Display Vital Signs

vitals acts as a multi-pen chart recorder displaying program behavior over time.

Up to five narrow, color-coded traces at the top show conversions to integer, real, numeric, string, and cset respectively. The first pixel of each trace marks the first conversion for that clock tick; each additional pixel indicates a doubling of the number of conversions.

Two more traces show allocation within the string and block regions (in that order). The gray background indicates the available memory; the black area indicates the current allocation.

Synopsis: vitals [*options*] [*eventstream*]

–R *color*	background color for allocation displays
–S *n*	spacing between sections
–s *n*	allow growth factor of *n* in string region
–b *n*	allow growth factor of *n* in block region

drive: A Front-End Driver for Parallel Visualization

drive runs one or more programs simultaneously from a single event stream. No explicit synchronization is provided.

Synopsis: drive [*options*] <*eventstream* ′*command*′ ...

The commands specified are interpreted by **sh** and must be in the current search path. There are no options specific to **drive**.

vis: Run Visualization Programs with Defaults

vis is a script that supplies parameters and runs multiple visualization programs in concert. It uses built-in defaults to find the source and event stream files corresponding to the specified Icon program.

Synopsis: vis [*options*] *program*

–a	run **anim**
–m	run **melody**
–r	run **roll**
–v	run **vitals**

References

1. R. E. Griswold, *Event Monitoring in Icon*, The Univ. of Arizona Icon Project Document IPD152, 1990.

2. C. L. Jeffery, *An Experimental Windows Facility for Icon*, The Univ. of Arizona Icon Project Document IPD150, 1990.

Appendix

The figures that follow show displays produced by the **roll**, **vitals**, and **anim** programs. All displays were derived from the same program run.

Figure 1, from the **roll** program, shows how the focus of program control moves around within the program. The vertical axis represents source code location, with each pixel row corresponding to one source line. The horizontal axis represents time, with each pixel column covering one clock tick. The gaps are caused by garbage collections, where multiple clock ticks occur without the execution of any user code. The light vertical line on the right marks program termination.

Several distinct phases of execution are clearly visible. A brief initialization phase is followed by three processing phases and then a short termination phase. The relative disorder of the central phase is caused by calls to several procedures at scattered source locations. The first processing phase reflects activity at one location in the main procedure plus one called procedure. The single wide bar in the last main phase reflects a multi-line loop with no calls to other procedures.

Figure 2, from the **vitals** program, uses the same horizontal time scale; note how the garbage collections line up with the other display. The ragged bands show the number of conversions to integer (gold), numeric (black), and string (red) occurring during each clock tick. Conversions to real and cset would also be shown.

The black areas within the gray bands show allocation in the string and block regions. The block region fills up three times, necessitating garbage collection. After a garbage collection there is again room for growth; were there not, the region would have been expanded and the gray band would have widened proportionally.

Figure 3 is a snapshot of the **anim** program taken early during the Icon program's central processing phase. The source program is not really legible but it is easily correlated with a separate listing.

anim highlights the last ten lines executed in red. Since only four lines are highlighted, some looping is in progress. Previously executed lines are black; unreached lines are medium gray; and comments are light gray.

The far left bar chart shows a profile based on clock ticks. One particular line near the center was credited with the vast majority of the ticks and presumably accounts for most of the execution time so far. The other bar chart, based on line counts, shows that the expensive line is one of four that were executed about the same number of times.

The chart along the right is a distorted version of the **roll** display.

Figure 4 shows the same **anim** display at the conclusion of execution. Notice how both bar charts were rescaled. The line that consumed most of the early execution time has been overshadowed by a new champion, and significant time was also consumed in a group of six lines towards the bottom.

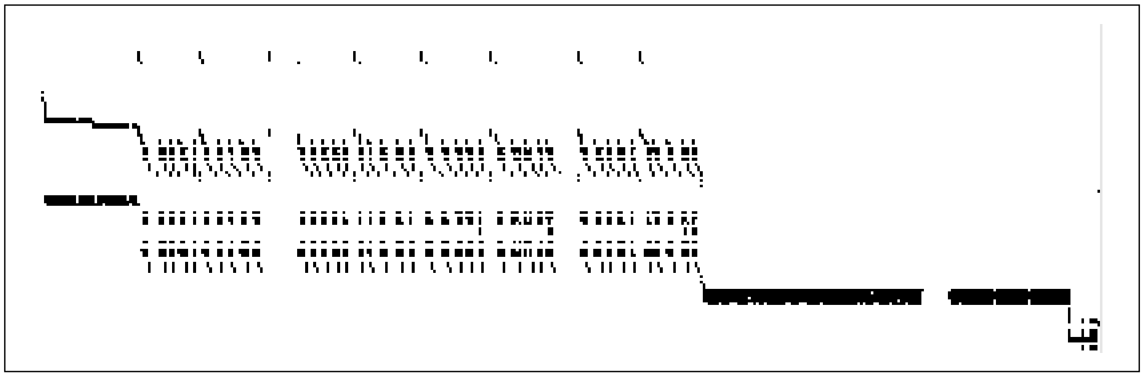


Figure 1: A Display Produced by the **roll** Program

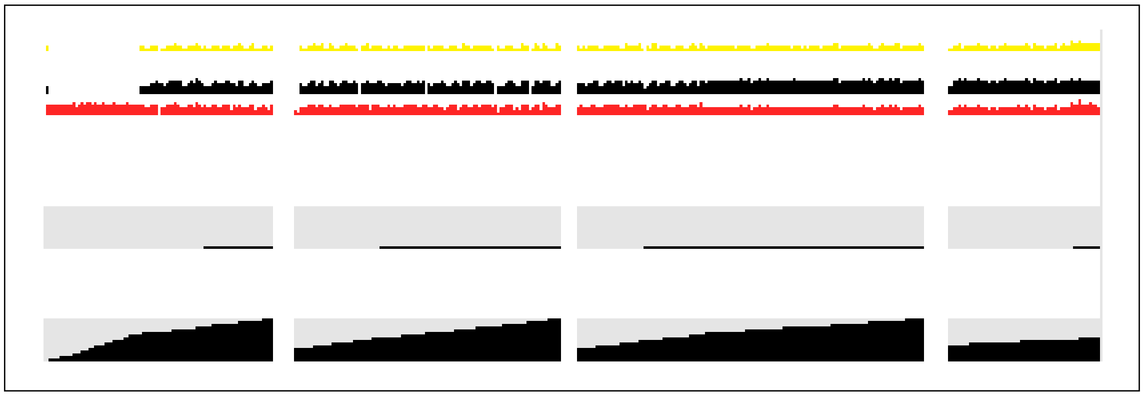
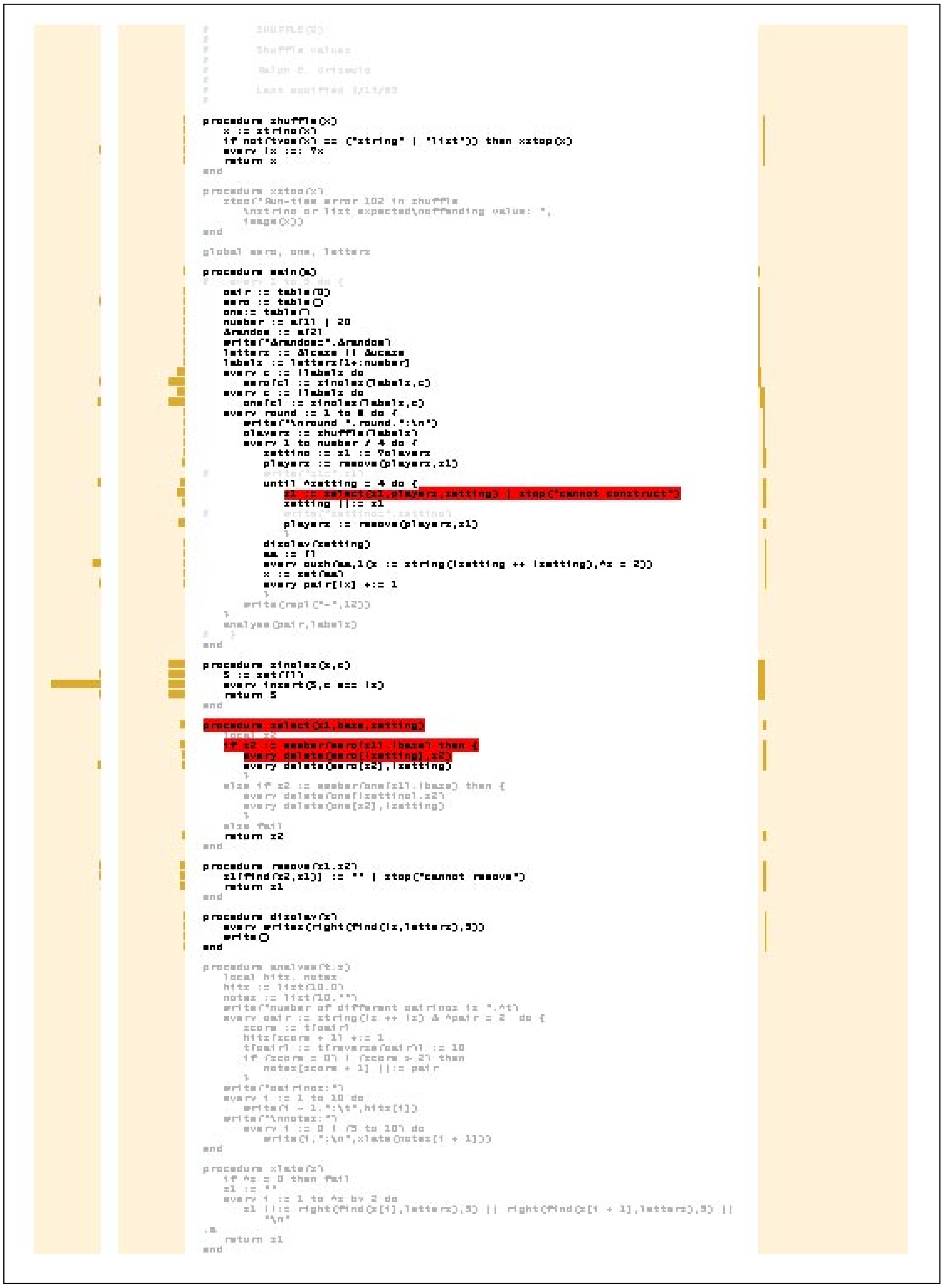
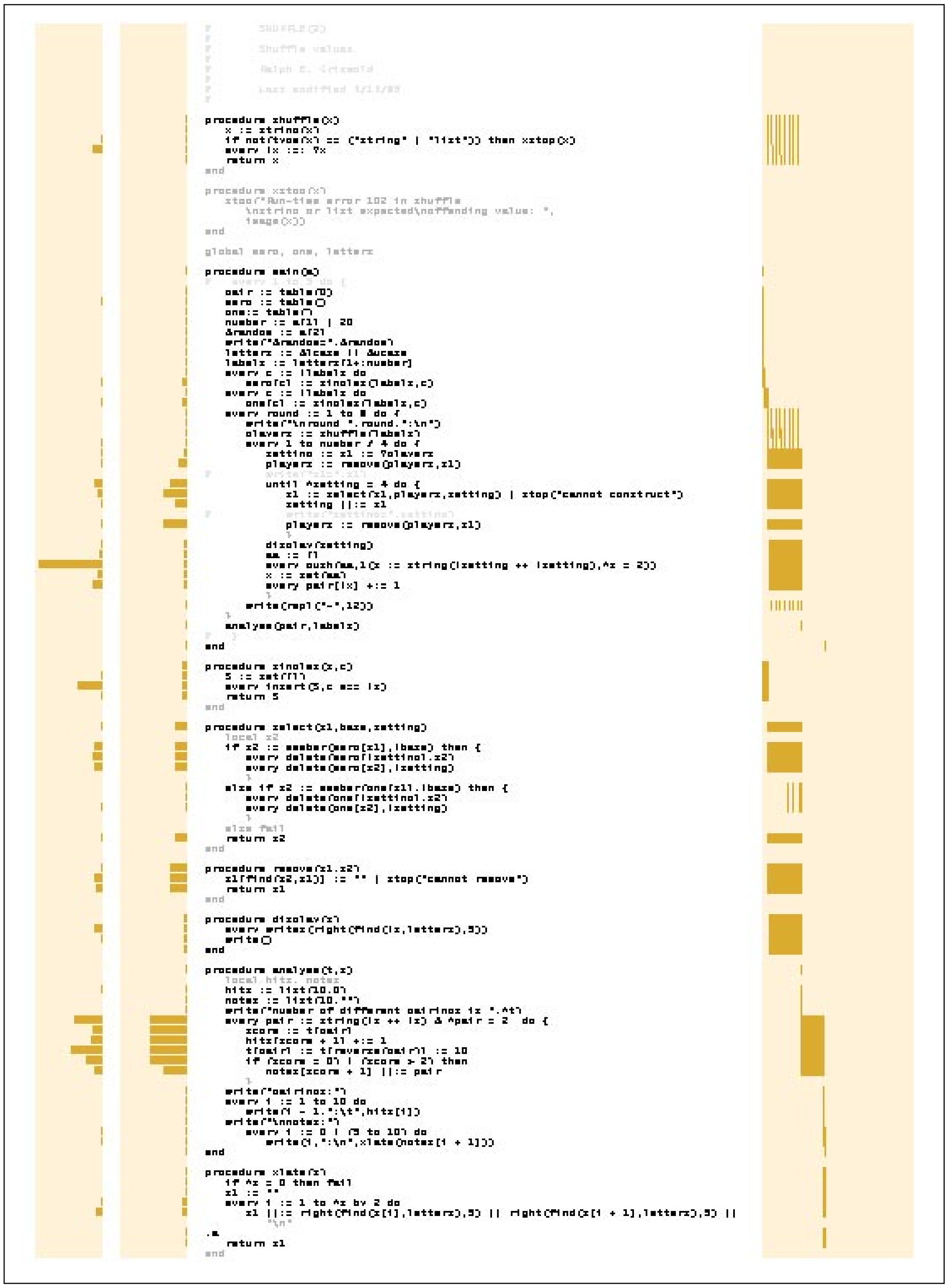


Figure 2: A Display Produced by the **vitals** Program



```
#       SHUFFLE(X)
#
#       Shuffle values
#
#       Ralph E. Griswold
#
#       Last modified 1/13/83
#

procedure shuffle(x)
   x := string(x)
   if not(type(x) == ("string" | "list")) then sstop(x)
   every !x :=: ?x
   return x
end

procedure sstop(x)
   stop("Run-time error 102 in shuffle
      \nstring or list expected\noffending value: ",
      image(x))
end

global zero, one, letters

procedure main(a)
#  every 1 to 3 do {
   pair := table(0)
   zero := table()
   one:= table()
   number := a[1] | 20
   &random := a[2]
   write("&random=",&random)
   letters := &lcase || &ucase
   labels := letters[1+:number]
   every c := !labels do
      zero[c] := singles(labels,c)
   every c := !labels do
      one[c] := singles(labels,c)
   every round := 1 to 8 do {
      write("\nround ",round,":\n")
      players := shuffle(labels)
      every 1 to number / 4 do {
         setting := s1 := ?players
         players := remove(players,s1)
#        write("s1=",s1)
         until *setting = 4 do {
            s1 := select(s1,players,setting) | stop("cannot construct")
            setting ||:= s1
#           write("setting=",setting)
            players := remove(players,s1)
            }
         display(setting)
         ss := []
         every push(ss,1(s := string(!setting ++ !setting),*s = 2))
         x := set(ss)
         every pair[!x] +:= 1
         }
      write(repl("-",12))
   }
   analyze(pair,labels)
#  }
end

procedure singles(s,c)
   S := set([])
   every insert(S,c ~== !s)
   return S
end

procedure select(s1,base,setting)
   local s2
   if s2 := member(zero[s1],!base) then {
      every delete(zero[!setting],s2)
      every delete(zero[s2],!setting)
      }
   else if s2 := member(one[s1],!base) then {
      every delete(one[!setting],s2)
      every delete(one[s2],!setting)
      }
   else fail
   return s2
end

procedure remove(s1,s2)
   s1[find(s2,s1)] := "" | stop("cannot remove")
   return s1
end

procedure display(s)
   every writes(right(find(!s,letters),5))
   write()
end

procedure analyze(t,s)
   local hits, notes
   hits := list(10,0)
   notes := list(10,"")
   write("number of different pairings is ",*t)
   every pair := string(!s ++ !s) & *pair = 2  do {
      score := t[pair]
      hits[score + 1] +:= 1
      t[pair] := t[reverse(pair)] := 10
      if (score = 0) | (score > 2) then
         notes[score + 1] ||:= pair
      }
   write("pairings:")
   every i := 1 to 10 do
      write(i - 1,":\t",hits[i])
   write("\nnotes:")
      every i := 0 | (3 to 10) do
         write(i,":\n",xlate(notes[i + 1]))
end

procedure xlate(s)
   if *s = 0 then fail
   s1 := ""
   every i := 1 to *s by 2 do
      s1 ||:= right(find(s[i],letters),5) || right(find(s[i + 1],letters),5) ||
         "\n"
.m
   return s1
end
```

Figure 3:   An Intermediate Display Produced by the **anim** Program



```
#       SHUFFLE(X)
#
#       Shuffle values
#
#       Ralph E. Griswold
#
#       Last modified 1/11/85
#

procedure shuffle(x)
   x := string(x)
   if not(type(x) == ("string" | "list")) then xstop(x)
   every !x :=: ?x
   return x
end

procedure xstop(x)
   stop("Run-time error 102 in shuffle
      \nstring or list expected\noffending value: ",
      image(x))
end

global zero, one, letters

procedure main(a)
#   every 1 to 5 do {
   pair := table(0)
   zero := table()
   one:= table()
   number := a[1] | 20
   &random := a[2]
   write("&random=",&random)
   letters := &lcase || &ucase
   labels := letters[1+:number]
   every c := !labels do
      zero[c] := singles(labels,c)
   every c := !labels do
      one[c] := singles(labels,c)
   every round := 1 to 8 do {
      write("\nround ",round,":\n")
      players := shuffle(labels)
      every 1 to number / 4 do {
         setting := s1 := ?players
         players := remove(players,s1)
#        write("s1=",s1)
         until *setting = 4 do {
            s1 := select(s1,players,setting) | stop("cannot construct")
            setting ||:= s1
#           write("setting=",setting)
            players := remove(players,s1)
            }
         display(setting)
         ss := []
         every push(ss,1(s := string(!setting ++ !setting),*s = 2))
         x := set(ss)
         every pair[!x] +:= 1
         }
      write(repl("-",12))
      }
   analyze(pair,labels)
#   }
end

procedure singles(s,c)
   S := set([])
   every insert(S,c ~== !s)
   return S
end

procedure select(s1,base,setting)
   local s2
   if s2 := member(zero[s1],!base) then {
      every delete(zero[!setting],s2)
      every delete(zero[s2],!setting)
      }
   else if s2 := member(one[s1],!base) then {
      every delete(one[!setting],s2)
      every delete(one[s2],!setting)
      }
   else fail
   return s2
end

procedure remove(s1,s2)
   s1[find(s2,s1)] := "" | stop("cannot remove")
   return s1
end

procedure display(s)
   every writes(right(find(!s,letters),5))
   write()
end

procedure analyze(t,s)
   local hits, notes
   hits := list(10,0)
   notes := list(10,"")
   write("number of different pairings is ",*t)
   every pair := string(!s ++ !s) & *pair = 2  do {
      score := t[pair]
      hits[score + 1] +:= 1
      t[reverse(pair)] := 10
      if (score = 0) | (score > 2) then
         notes[score + 1] ||:= pair
      }
   write("pairings:")
   every i := 1 to 10 do
      write(i - 1,":\t",hits[i])
   write("\nnotes:")
      every i := 0 | (3 to 10) do
         write(i,":\n",xlate(notes[i + 1]))
end

procedure xlate(s)
   if *s = 0 then fail
   s1 := ""
   every i := 1 to *s by 2 do
      s1 ||:= right(find(s[i],letters),5) || right(find(s[i + 1],letters),5) ||
         "\n"
.e
   return s1
end
```

Figure 4: A Final Display Produced by the **anim** Program