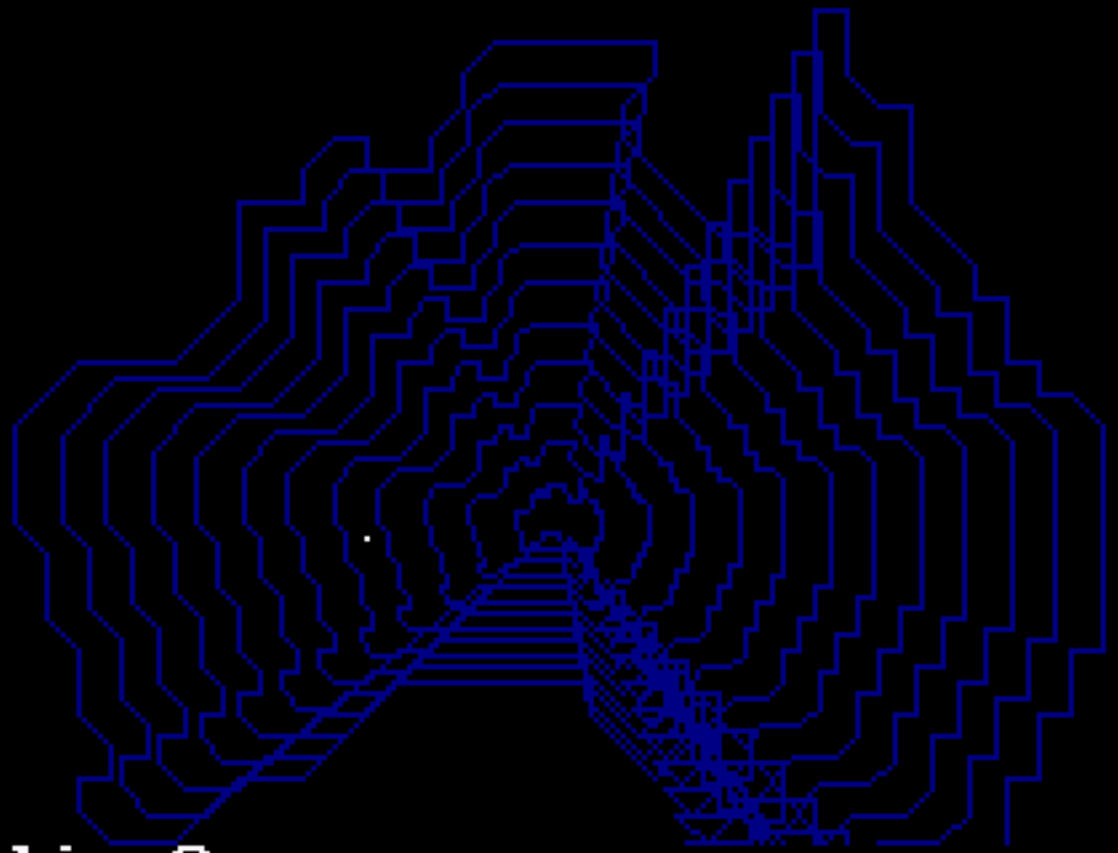
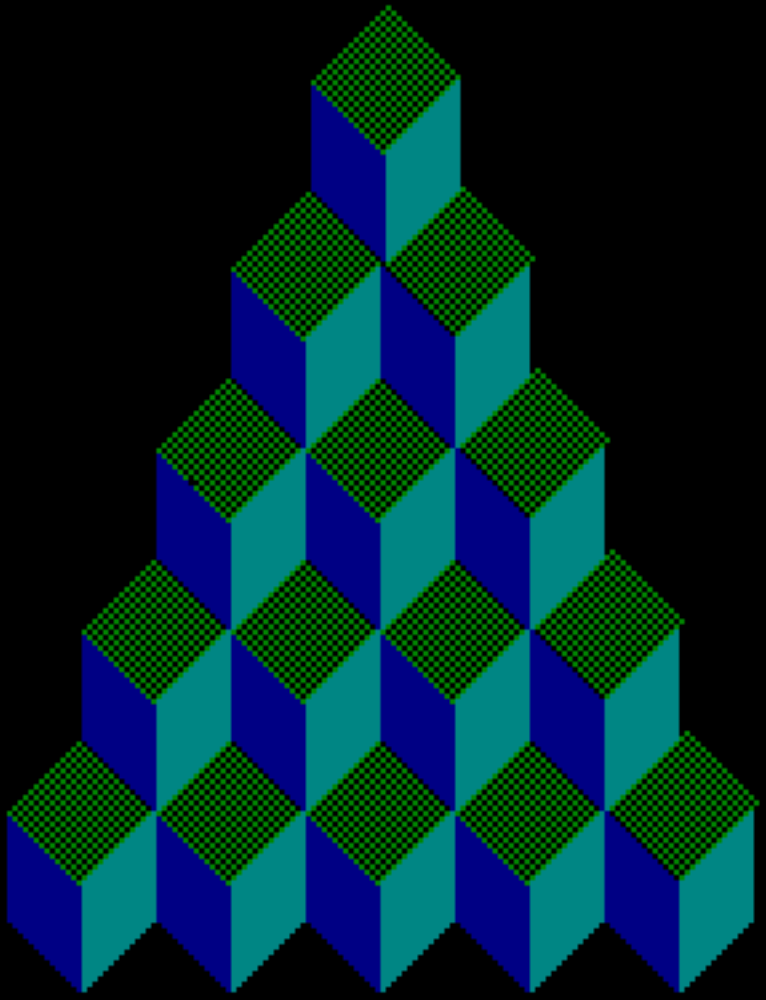
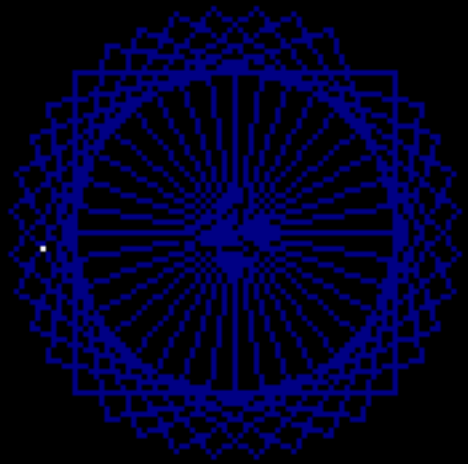


```
?australia 9
?australia 10
?australia 11
?australia 12
?
```





```
?cs
?home
?pd
?design
?
```

DRLNL1.WS4 (= "Dr. Logo NewsLetter #1")

- "Dr. Logo Newsletter"
*** First Edition: April 1984 ***
6000-1031-001

(Retyped by Emmanuel ROCHE.)

Table of Contents

Editor's introduction

Welcome to the first ever Dr. Logo Newsletter! We have worked long and hard to bring you information I hope will be both useful and enjoyable to you. There is quite a bit of material here, so don't be surprised it it doesn't all make sense at the first reading (it didn't all make sense at the first writing). We made the newsletter the same format as all the other Dr. Logo documentation, to help you keep all of your material together.

As you read through the articles, you will probably notice a slightly more sophisticated orientation that is usual in Logo materials. This is a reflection of the survey responses and my personal tastes, both in Logo and in writing. If you find you need help penetrating my purple prose, I will answer any letters accompanied by a SASE (Self-Addressed Stamped Envelope). My address is:

Joseph R. Power
Digital Research
160 Central Ave.
Pacific Grove, CA 93950

Response card results

At the end of December, 1983, we had received back 470 response cards for the free Dr. Logo Newsletter. This reflects a much higher response rate than for ANY other Digital Research product (including CP/M). There were some surprises in the numbers we recorded -- most of you are adults, or adult & child combination, and Dr. Logo is being used almost exclusively in the home. The numbers are listed below, with percentages that were derived by dividing the totals by 470. The numbers don't always add up to 470 (100%) because many cards had multiple items checked off. This is who you are:

I use Dr. Logo at

Home: 401 (85.3%) School: 65 (13.8%) Business: 106 (22.6%)

Dr. Logo's primary user's age is

5-8 : 137 (29.1%)
9-12: 166 (35.3%)
13-18: 55 (11.7%)
Adult: 312 (66.4%)

By state or country

AK 2 .42 %
AZ 6 1.3 %
CA 133 24.0 %
CN 1 .21 %

```
CO 10   2.1  %
CT 13   2.76 %
DC 3     .638%
FL 5    1.06 %
GA 4     .85 %
HI 22   4.68 %
IA 2     .42 %
IL 16   3.4  %
IN 5    1.06 %
KS 6    1.27 %
KY 3     .638%
LA 3     .638%
MA 29   6.17 %
MD 12   2.55 %
ME 2     .42 %
MI 8    1.7  %
MN 3     .638%
MO 7    1.49 %
MT 1     .21 %
NC 6    1.3  %
NE 2     .42 %
NH 3     .638%
NJ 11   2.34 %
NM 9    1.91 %
NV 1     .21 %
NY 24   5.1  %
OH 6    1.3  %
OK 7    1.49 %
OR 8    1.7  %
PA 13   2.76 %
RI 1     .21 %
SC 2     .42 %
SD 1     .21 %
TN 3     .638%
TX 31   6.59 %
UT 2     .42 %
VA 14   2.98 %
VT 2     .42 %
WA 11   2.34 %
WI 8    1.7  %
WV 1     .21 %
WY 1     .21 %
CANADA   21 4.47%
COLUMBIA  1  .21%
JAPAN     1  .21%
KOREA     1  .21%
SWEDEN    1  .21%
```

Since this initial survey, the numbers have been changing somewhat, with the kids (especially the 9-12 age range) starting to overtake the adults. Come on, adults! let's get in there, and rally back to the lead. They might be able to demolish us at blinkey-death video games, but in Logo we all start as equals.

Logy, morph, and Ma Bell

The Young Person's Logo Association has a special treat for all you Logophiles -- a computer bulletin board system (CBBS for short). A CBBS is a computer that people can call and talk to, using their own computer and a modem. Your computer must be able to act like a terminal (usually by running a terminal program like PC-Talk III) and your modem must be in 'originate' mode at 300 baud. Once all that is set up, call the Midnight Turtle (the name of this CBBS) after 7PM CST (central standard time) at (214)-783-7548 and, once the two computers start whistling at each other, you're in!

This bulletin board is a great place to leave messages, ask questions, and answer someone else's questions if you can. There will be useful information, friendly tips, and even Logo programs you can download. So, give it a try.

Here is a brief summary of the commands available at the CBBS 'toplevel'. Once you have chosen one of these commands, other subcommands might be needed. If you are ever unsure of what to do, simply type HELP or ?.

Command	Stands for	Effect
B	Bulletins	Read or send messages to everyone
C	Chat	Try to talk to Jim Muller
H	Helpful info	Helpful hints and command summary
I	Information	Information about the CBBS
L	List TP files	For up- and down-loading

```
   N     Normal info     More information about the CBBS
   R     Recommendation  Private message to Jim Muller
   RN    Reread News     Retypes the log-in messages
   S     Status          Information about your call
   T     Time info       Info about duration of call
   U     new User info   A very good place to start
   MR    Mail Read       Read messages sent to you
   MS    Mail Send       Send messages to other users
   OFF   get OFF CBBS    End session, and hang up.
```

Unless you are a member of YPLA (which we heartily recommend), some functions are unavailable to you. The system is VERY new, and there might be some bugs in it. If something doesn't seem to work right, leave a message for Jim Muller via the R command.

One word of caution: Ma Bell eats pennies faster than Pac-Man chomps dots, so don't stay on the system for hours at a time. Also, remember that this is a text-only system. While there might be turtle graphics procedures on the CBBS, there are no finished drawings to look at.

For more information, contact

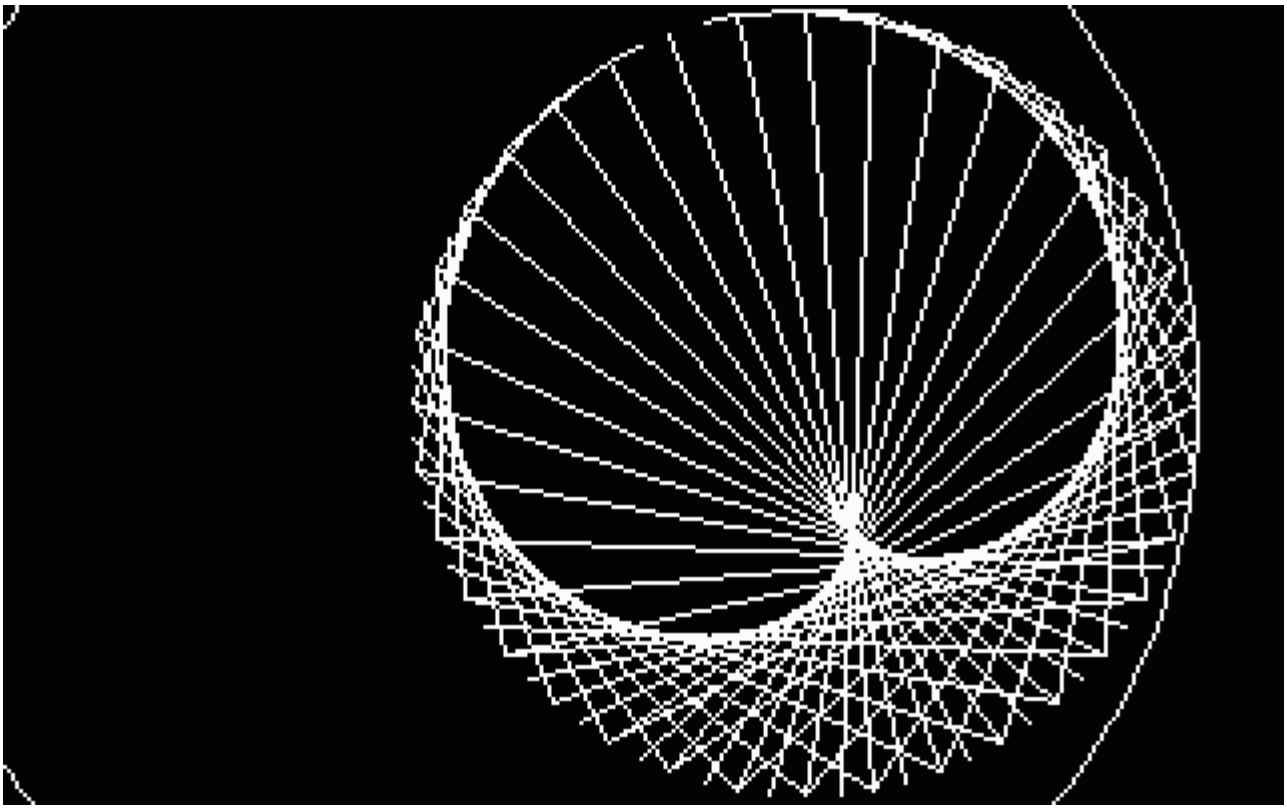
The Young Peoples' Logo Association
1208 Hillsdale Drive
Richardson, TX 75081

String art with Dr. Logo

```
to string :sz :offset :halves
(local "n "o "p "q)
make "o pi / 108
clean ht make "p pos pu setx xcor + :sz pd
repeat 360
  [fd :sz * :o lt 1]
pu setpos :p make "n 1
repeat 36 * :halves
  [pu fd :sz make "q pos
   setpos :p seth remainder (5 * :n * :offset) 360
   fd :sz pd setpos :q pu
   setpos :p seth remainder (5 * (:n - 1)) 360
   make "n :n + 1]
end
```

Example:

```
cs pu setpos [40 0]
string 98 2 2
```



Faster turtle graphics in Dr. Logo

While Dr. Logo currently has the fastest graphics of all the Logos we know about, there probably isn't anyone who wouldn't like to see even faster graphics. This article presents a number of techniques for speeding the turtle on its merry way.

We begin with a very simple, but often ovelooked one: hide the turtle! When the turtle is show, and you move or rotate it, the old turtle is erased, the action is performed, and the turtle is redrawn at the new position and heading. When the turtle is hidden, all the overhead of erasing and redrawing it is saved. A hidden turtle ALWAYS moves faster than a shown turtle when it is on the screen.

Another technique that helps is the use of SETHEADING (SETH) instead of LEFT (LT) and RIGHT (RT). Pointing the turtle to absolute headings, instead of relative ones, generally reduces the amount of math performed by the Logo interpreter. Thus, if the turtle is pointing straight up (heading = 0), use SETH 90 in preference to RT 90. If the turtle's heading is 315 (after a LT 45), use SETH 288 instead of LT 27 (288 = 315 + -27). The general formula is New_heading = Old_heading + Angle, which means that the new heading is the sum of the old heading and the angle to be turned. This angle will be a positive number if the turn is to the right, and a negative number if the turn is to the left. Numbers larger than 360 (or smaller than -360) will work properly (seth's argument is taken modulus 360).

A closely-related technique is the use of SETPOS in lieu of FORWARD (FD) and BACK (BK). Here again, absolute positioning requires less math on Logo's part than relative motion does. One way to make use of these two techniques is to code a procedure with the normal relative commands originally, until it works properly. Then, add print statements that tell what the turtle's heading and/or position are at various points in the program. Finally, using this information, many of the relative commands can be replaced by moving the turtle directly to the locations and headings observed in the print statements.

Another appropriate use of setpos and setheading is to go quickly to some fixed position. Instead of:

```
        fd 60 rt 90 fd 100 bk 100 lt 90 bk 60
use:
        make "p pos
        make "h heading
        fd 60 rt 90 fd 100
        pu setpos :p seth :h pd
```

Now, before Logo purists descend with fire in their eyes at the espousing of such rank heresies, it must be stressed that all of these techniques should be employed only when faster graphics are important. Follow the principle of 'make it work, then make it fast'.

The observant reader will have noticed the PENUP (PU) command in the second example above. It is plain to see that putting the pen up prevented drawing an unwanted diagonal line. What is not so plain to see is that the turtle also moves faster with the pen up than with it down (or erasing or reversing). Why is this?

When the turtle is commanded to move, Logo must calculate the new position, and draw a line from where the turtle is to where it will be. When the pen is up, Logo can quit right after figuring out the new position. So, whenever possible, keep that pen up.

In the discussion of SETHEADING and SETPOS, it was stated that use of those primitives cut down on the amount of math that Logo had to perform. Another trick that cuts down on the amount of math is to cut down on the precision of the numbers being used. Whenever possible, use integers as these are the easiest for Logo. If decimal numbers are needed, use the fewest digits of precision tolerable. For example, Logo says that expression SQRT 2 is 1.4142135623731 but, for most graphics applications, 1.4142 is more than adequate.

Finally, there is a great deal of overhead when Logo enters and exits a procedure, and a smaller, though still noticeable, amount when using REPEAT loops. Whenever possible, therefore, unwind loops, and expand procedures in-line. As an example of these two, try these procedures:

```
to design
repeat 36
  [square 30 rt 10]
end

to square :sz
repeat 4
  [fd :sz rt 90]
end
```

First, the loop in SQUARE is unwound:

```
to square :sz
fd :sz rt 90 fd :sz rt 90 fd :sz rt 90 fd :sz rt 90
end
```

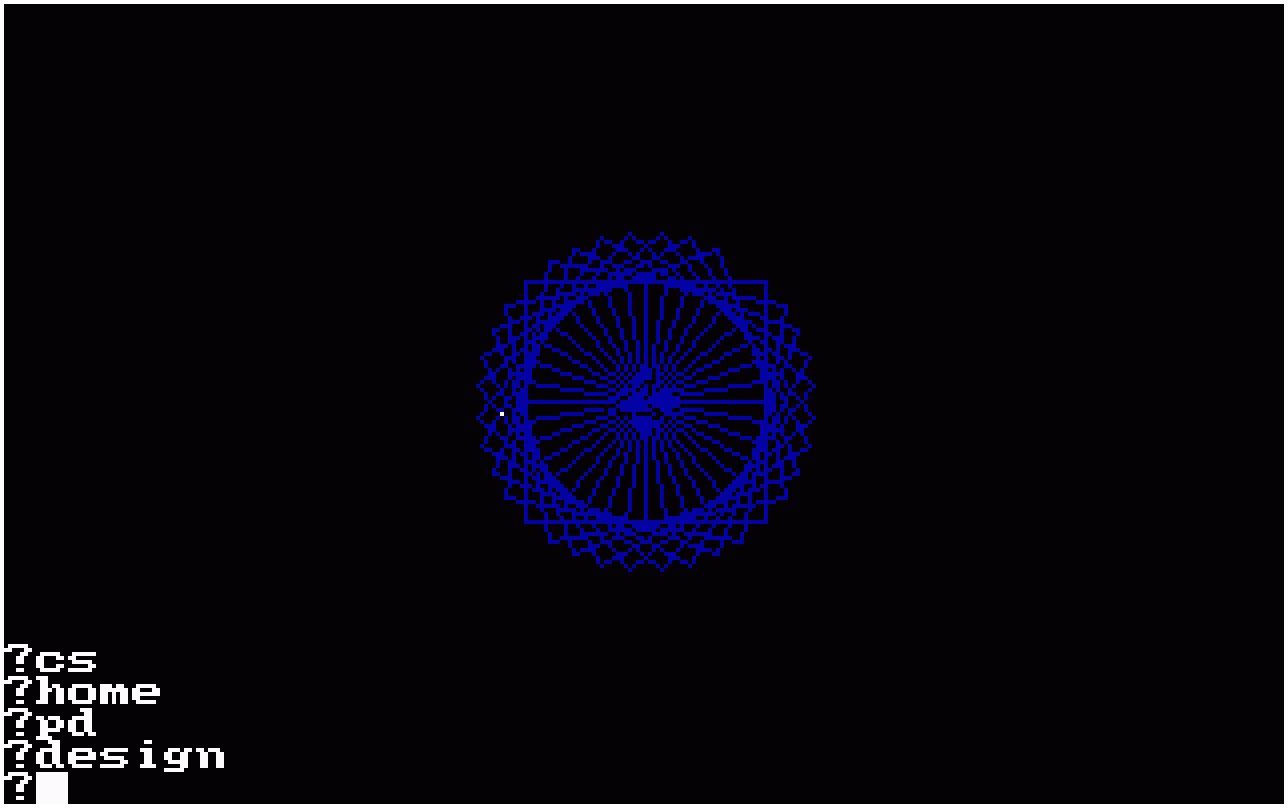
Next, expand the SQUARE procedure in-line in DESIGN:

```
to design
repeat 36
  [fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100]
end
```

Finally, unwind the loop in DESIGN:

```
to design
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
; Repeat 35 times
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
```

```
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 rt 100
fd 30 rt 90 fd 30 rt 90 fd 30 rt 90 fd 30 ; No final turn
end
```



This is a great deal less tedious if ^K and ^Y are used.

In summary, the primary techniques for making Logo graphics run faster are:

1) Hide the turtle
2) Use SETHEADING and SETPOS
3) Use only the required amount of precision
4) Keep the pen up as much as possible
5) Unwind loops and expand procedures in-line

These techniques will work with most versions of Logo, although with varying levels of speedup. Remember -- do it right, then do it fast.

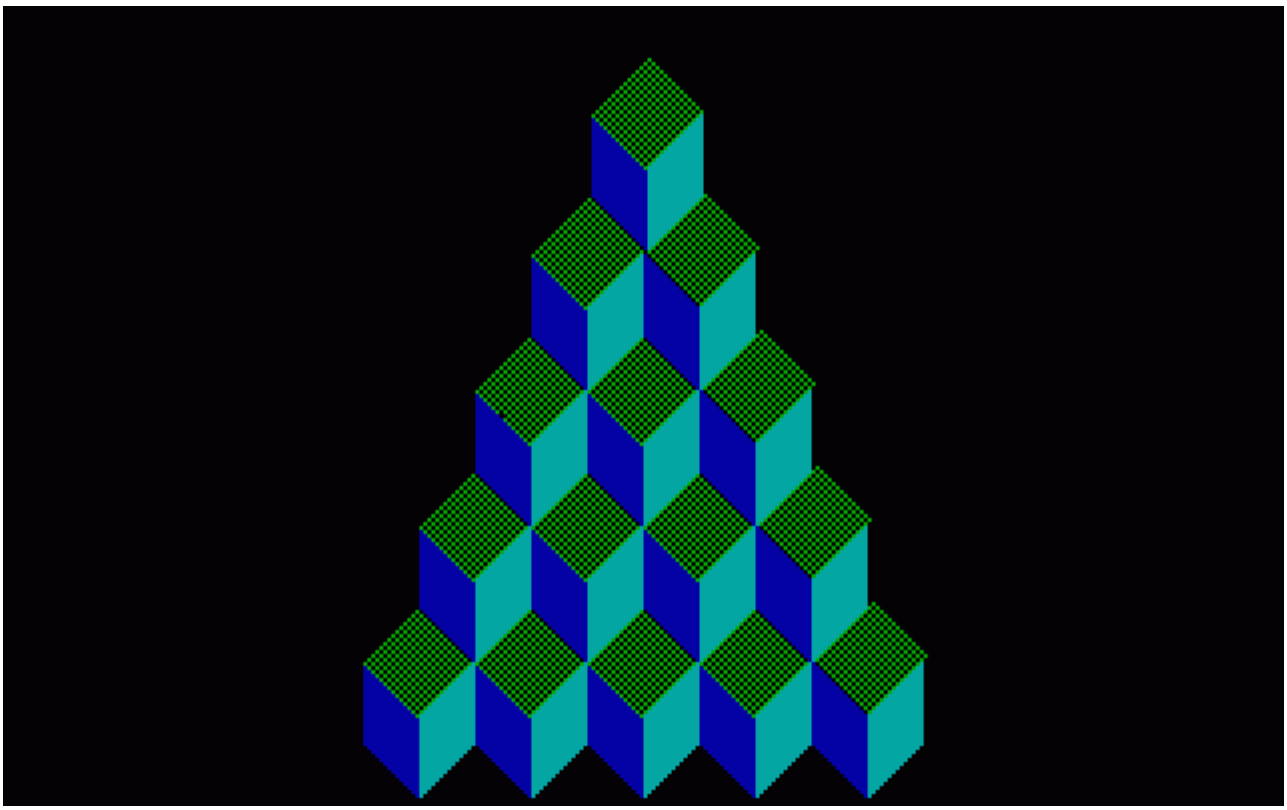
A cascade of color

These three procedures draw a spectacular pattern on the graphics screen. After you have typed them in and saved them on disk (always save your work on disk), simply type 'cascade'.

```
to cascade
make "sqr2 sqrt 2
setbg 16 fullscreen
cs ht pu setpos [-70 -85] pd
make "x xcor make "y ycor
repeat 5 [side] pu setx :x sety :y + 20 seth 45
repeat 5 [top]  pu setx :x sety :y + 20 seth 45 fd 20 seth 0
make "x xcor make "y ycor
repeat 4 [side] pu setx :x sety :y + 20 seth 45
repeat 4 [top]  pu setx :x sety :y + 20 seth 45 fd 20 seth 0
make "x xcor make "y ycor
```

```
repeat 3 [side] pu setx :x sety :y + 20 seth 45
repeat 3 [top]  pu setx :x sety :y + 20 seth 45 fd 20 seth 0
make "x xcor make "y ycor
side side       pu setx :x sety :y + 20 seth 45
top  top        pu setx :x sety :y + 20 seth 45 fd 20 seth 0
make "x xcor make "y ycor
side            pu setx :x sety :y + 20 seth 45
top
end

to top
setpc 2
repeat 7
  [pd fd 20 seth 135 fd :sqr2 seth 45
   bk 20 pu seth 135 fd :sqr2 seth 45]
fd 20
end

to side
setpc 1
repeat 6
  [pd fd 20 seth 135 fd :sqr2 seth 0
   bk 20 seth 135 pu fd :sqr2 seth 0]
pd fd 20 seth 135 fd :sqr2 seth 0 bk 20 pu
setx xcor + 1
setpc 3
repeat 6
  [pd fd 20 seth 45 fd :sqr2 seth 0
   bk 20 pu seth 45 fd :sqr2 seth 0]
pd fd 20 seth 45 fd :sqr2 seth 0 bk 20 pu
setx xcor + 1
end
```

People who frequent video arcades should recognize this pattern.



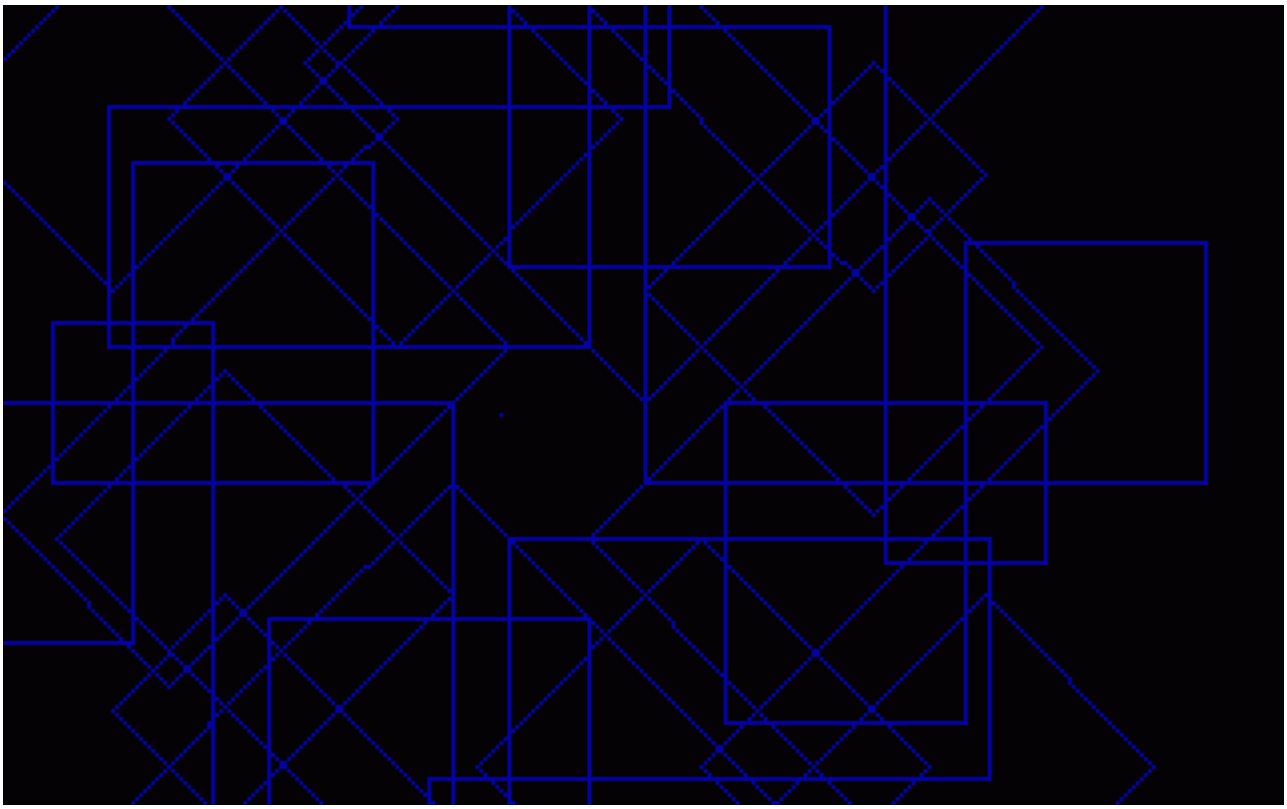
Mine

```
to mine :sz
fd 5 * :sz rt 90 fd 3 * :sz rt 90 fd 6 * :sz lt 90 fd 4 * :sz lt 90
fd 3 * :sz lt 90 fd 3 * :sz lt 90 fd 6 * :sz rt 90 fd 3 * :sz rt 90
fd 4 * :sz rt 90 fd 4 * :sz rt 90 fd 2 * :sz rt 90 fd 2 * :sz rt 90
fd 1 * :sz lt 90 fd 3 * :sz rt 90 fd 1 * :sz bk :sz
end

to mine2 :sz :num
```

```
repeat :num
  [mine :sz rt 360 / :num]
end
```

Example:

```
mine2 20 8
```



A simple 3-D graphics package

By now, all of you are familiar with the turtle graphics of Dr. Logo. Many of you are quite good at putting fantastic objects on the screen with amazing ease. So, having mastered this level of difficulty, it is time to move up to the next dimension -- the third dimension. That's right, this article is about a third dimensional graphics system written in Dr. Logo.

In searching for a system that was small and not too slow, I finally chose to abandon the usual turtle graphics for Cartesian coordinate graphics. This allowed me to use some simple matrix multiplications to rotate or alter the view of the shape.

In this package, the basic unit is the point, defined with the procedure POINT (oddly enough). Points have names and [x y z] coordinate lists telling where they are in 3-D space.

Once you have defined all the points, you can construct shapes. The SHAPE procedure takes the shape name and a list of two-element lists (ex: [[a b] [a c] [b g] [g j]]). The two-element lists represent the line segments of the shape, with each element being an endpoint. You can have as many shapes as you want, but only one at a time can be manipulated.

Once you've defined your shape, you can expand (or contract) it, rotate it, magnify (or shrink) it, shear it, or restore it to its original state.

To expand a shape, use the EXPAND procedure, and tell it which shape to expand, which axis (x, y, or z) the expansion will operate on, and how much to expand it. Amounts between 1 and 0 will contract, rather than expand, the shape. Negative amounts mirror the shape across the center point of the screen.

Rotation occurs not on an axis, but on a plane (xy, xz, or yz). Again, you specify the shape, the plane, and the amount to rotate with the ROTATE procedure.

The procedure MAGNIFY is very similar to the EXPAND procedure. You don't specify an axis, however, because the shape is expanded or decreased in all directions.

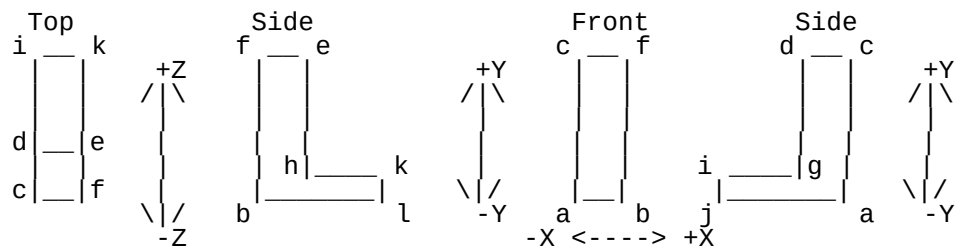
Shearing the shape involves tilting it to the left or right, or up or down, direction. Because of the way the shapes are displayed, shearing in the Z axis has no effect, and is therefore forbidden.

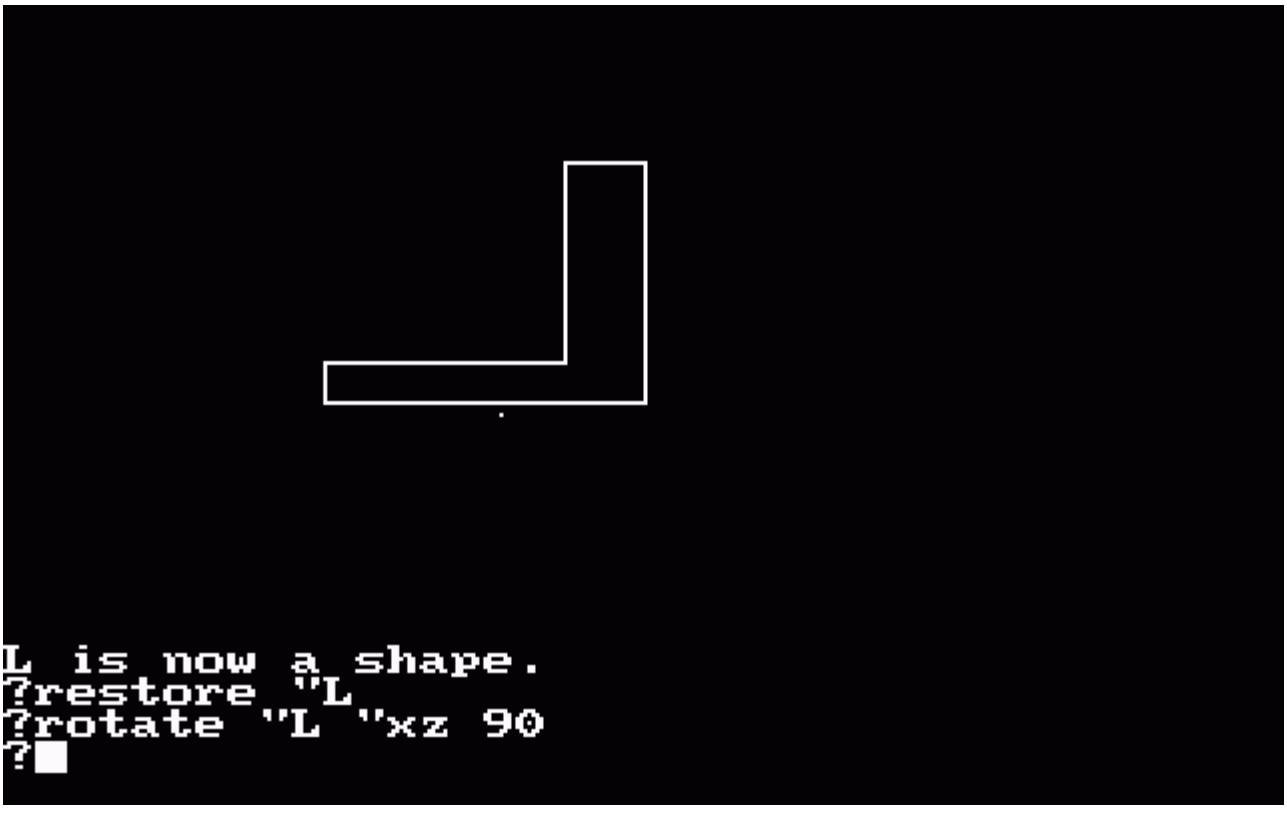
When you want to start all over with a shape (since transformations are cumulative), use the RESTORE procedure.

Believe it or not, that's all there is to it. We strongly suggest you play with these procedures to get a better feel for how they work. If you want to explore 3-D graphics further, try the September 1978 issue of BYTE magazine and the Abelson and DiSessa book, "Turtle Geometry" (listed in your Dr. Logo Bibliography).

In addition to providing the listings, we are also providing a simple shape to get you started. Just type in the following lines, and watch what happens.

```
?L
L is now a shape.
restore "L
rotate "L "xz 45
rotate "L "yz 30
etc...
```





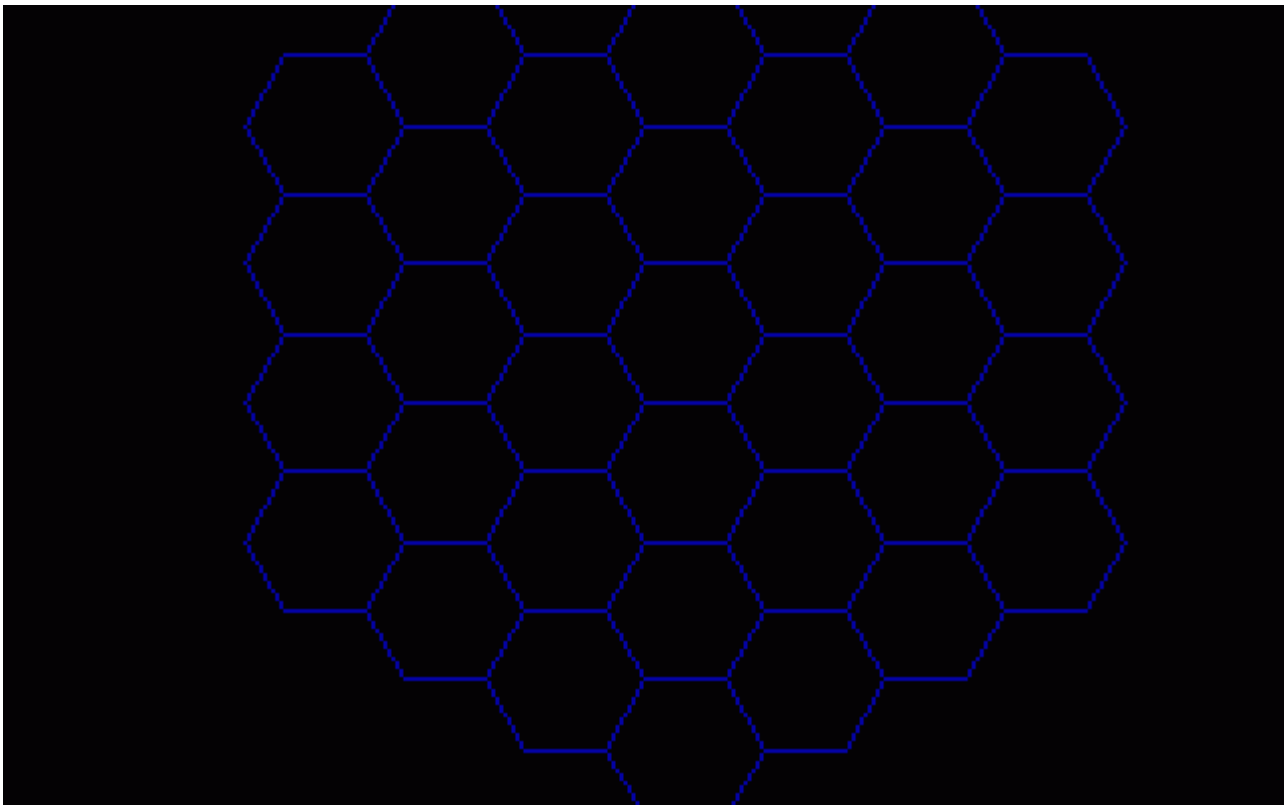

```
to L
point "a [0 0 0]
point "b [50 0 0]
point "c [0 60 0]
point "d [0 60 20]
point "e [50 60 20]
point "f [50 60 0]
```

```
point "g [0 10 20]
point "h [50 10 20]
point "i [0 10 80]
point "j [0 0 80]
point "k [50 10 80]
point "l [50 0 80]
shape "L [[a b] [a c] [a j] [b f] [b l] [c d] [c f] [d g] [d e] [e f] [e h] [g
i] [g h] [h k] [i j] [i k] [j l] [k l]]
end

to point :point_name :coords
make :point_name :coords
pprop :point_name "point "TRUE
pprop :point_name "orig :coords
end

to shape :shape_name :line_pairs
if (gprop :shape_name "point) = "TRUE
   [(pr :shape_name [is already a point name.]) stop]
make :shape_name :line_pairs
pprop :shape_name "shape "TRUE
make "shapex (word :shape_name "_pts)
make :shapex []
make "n9 1
repeat count :line_pairs
  [if not memberp first (item :n9 :line_pairs) thing :shapex
      [make :shapex fput first (item :n9 :line_pairs) thing :shapex]
   if not memberp last (item :n9 :line_pairs) thing :shapex
      [make :shapex fput last (item :n9 :line_pairs) thing :shapex]
   make "n9 :n9 + 1]
make "matrix [1 0 0 0 1 0 0 0 1]
draw :shape_name
(pr :shape_name [is now a shape.])
end

to expand :shape :axis :amt
if not memberp :axis [x y z]
   [pr [The axis must be "x, "y, or "z.] stop]
if not (gprop :shape "shape) = "TRUE
   [(pr :shape [is not a shape.]) stop]
if :axis = "x
   [make "matrix (list :amt 0 0 0 1 0 0 0 1)]
if :axis = "y
   [make "matrix (list 1 0 0 0 :amt 0 0 0 1)]
if :axis = "z
   [make "matrix (list 1 0 0 0 1 0 0 0 :amt)]
draw :shape
end

to rotate :shape :axis :amt
if not memberp :axis [xy xz yz]
   [pr [The axis must be "xy, "xz, or "yz.] stop]
if not (gprop :shape "shape) = "TRUE
   [(pr :shape [is not a shape.]) stop]
if :axis = "xy
   [make "matrix (list
   (cos :amt) 0 - (sin :amt) 0
   (sin :amt) (cos :amt) 0
   0 0 1)]
if :axis = "xz
   [make "matrix (list
   (cos :amt) 0 0 - (sin :amt)
   0 1 0
   (sin :amt) 0 (cos :amt))]
if :axis = "yz
   [make "matrix (list
   1 0 0
   0 (cos :amt) 0 - (sin :amt)
   0 (sin :amt) (cos :amt))]
draw :shape
end

to magnify :shape :amt
if not (gprop :shape "shape) = "TRUE
   [(pr :shape [is not a shape.]) stop]
make "matrix (list :amt 0 0 0 :amt 0 0 0 :amt)
draw :shape
end

to shear :shape :axis :amt
if not memberp :axis [x y]
```

```
    [pr [The shear axis must be "x or "y.] stop]
  if not (gprop :shape "shape) = "TRUE
     [(pr :shape [is not a shape.]) stop]
  if :axis = "x
     [make "matrix (list 1 :amt 0 0 1 0 0 0 1)]
     [make "matrix (list 1 0 0 :amt 1 0 0 0 1)]
  draw :shape
  end

  to restore :shape
  if not (gprop :shape "shape) = "TRUE
     [(pr :shape [is not a shape.]) stop]
  make "n9 thing (word :shape "_pts)
  repeat count :n9
    [make first :n9 gprop (first :n9) "orig
     make "n9 bf :n9]
  make "matrix [1 0 0 0 1 0 0 0 1]
  draw :shape
  end

  to draw :shape
  make "s9 thing (word :shape "_pts)
  repeat count :s9
    [make "p9 first :s9
     make :p9 (list
       (item 1 :matrix) * (item 1 thing :p9) +
       (item 2 :matrix) * (item 2 thing :p9) +
       (item 3 :matrix) * (item 3 thing :p9)
       (item 4 :matrix) * (item 1 thing :p9) +
       (item 5 :matrix) * (item 2 thing :p9) +
       (item 6 :matrix) * (item 3 thing :p9)
       (item 7 :matrix) * (item 1 thing :p9) +
       (item 8 :matrix) * (item 2 thing :p9) +
       (item 9 :matrix) * (item 3 thing :p9))
     make "s9 bf :s9]
  make "s9 thing :shape
  cs ht
  repeat count :s9
    [pu setpos bl thing (first first :s9)
     pd setpos bl thing (last first :s9)
     make "s9 bf :s9]
  end


  Presenting -- the doctor
  ------------------------

  Here is another set of procedures to astound and confound! Yes, Dr. Logo  will
  display a grand portrait of the good Doctor himself. The three braid  routines
  (BRAID, STRIP, and CORNER) have been modified for speed in this collection.

  to braid
  (local "sqr2 "hfsq2 "s2 "h2 "s2h2)
  make "sqr2 1.4 ; sqrt 2
  make "hfsq2 0.7 ; :sqr2 * 0.5
  make "s2 8.5 ; :sqr2 * 6
  make "h2 4.2 ; :hfsq2 * 6
  make "s2h2 12.7 ; :s2 + :h2
  pu fd 24 rt 45 fd 4.2 seth 0 pd
  strip 13 corner strip 21 corner
  strip 13 corner strip 21 corner
  end

  to circle2
  repeat 36
    [fd 2 rt 10]
  end

  to face
  make "x xcor make "y ycor
  repeat 2 [circle2 lt 90 fd 12 lt 90] ht lt 180 fd 35 bk 5
  lt 90 fd 4 rt 90 fd 5 rt 90 fd 20 rt 90 fd 5
  rt 90 fd 4 lt 90 fd 30 rt 180 fd 35
  lt 90 fd 6 repeat 10 [fd 3 rt 9]
  rt 90 repeat 10 [fd 3 rt 9]
  lt 90 repeat 10 [fd 3 lt 9]
  lt 90 repeat 10 [fd 3 lt 9] pu
  lt 90 fd 20 rt 90 pd bk 12 fd 65
  lt 90 fd 20 lt 90 fd 20 bk 22 lt 90 fd 80
  rt 90 fd 2 rt 90 fd 60 bk 38
  lt 90 fd 65 pu sety :y - 70 setx :x - 6
```

```
lt 135 pd fd 22 bk 22
lt 90 fd 22 pu setx :x - 27 sety :y seth 0
lt 90 pd repeat 10 [fd 4 lt 20] pu setx :x + 13 sety :y seth 0
rt 90 pd repeat 10 [fd 4 rt 20] ht
end

to strip :n
make "pc 0
repeat :n
  [lt 45 fd :h2 rt 45 fd 6 rt 45 fd :s2h2 pu
   rt 90 fd :h2 pd rt 90 fd :s2 lt 45 fd 6
   if pc < 3
      [setpc pc + 1]
      [setpc 1]
   if pc = 3
      [setpc 4]
   pu lt 45 fd :s2h2 pd lt 135
   rt 45 fd :h2 lt 45 fd 6 lt 45 fd :s2h2 pu
   lt 90 fd :h2 pd lt 90 fd :s2 rt 45 fd 6
   pu rt 135 fd :s2h2 rt 45 fd 6 pd]
end

to corner
make "pc 0
lt 45 fd :h2 rt 45 fd 6
rt 45 fd :s2 rt 45 fd 18
rt 45 fd :s2h2 pu
rt 90 fd :h2 pd rt 90 fd :s2
lt 45 fd 18 lt 90 fd 6 pu
if pc < 3
   [setpc pc + 1]
   [setpc 1]
if pc = 3
   [setpc 4]
lt 45 fd :s2 pd lt 90 fd 17 pu
rt 90 fd :h2 pd rt 90 fd 17 pu
if pc < 3
   [setpc pc + 1]
   [setpc 1]
if pc = 3
   [setpc 4]
rt 45 fd 6 rt 90 fd 12 pd
rt 45 fd :h2 rt 45 fd 6
rt 45 fd :h2 pu rt 90 fd :h2 pd
rt 45 fd 6 pu bk 15 rt 90 fd 9 rt 90 pd
if pc < 3
   [setpc pc + 1]
   [setpc 1]
if pc = 3
   [setpc 4]
end

to drlogo
local "y
fs cs
setbg 0 setpc 7 ; textbg 0
cs ht
face
pu setpos [-150 -100] pd ht seth 0
setpc 1 braid
pu setpos [-110 60]
setpc 4 pd
tt [Digital Research presents:]
pu
bk 30 setx xcor + 25 pd seth 0
setpc 1
;
; Dr. Logo in turtle graphics
;
make "y ycor fd 22 rt 90 repeat 19 [fd 2 rt 10]
pu setx xcor + 17 pd seth 0 fd 10 bk 3 rt 22 fd 3 seth 90 fd 5
pu seth 0 sety :y setx xcor + 20 pd fd 22 bk 22 rt 90 fd 15
pu fd 5 seth 0 fd 11 pd circle2 pu
repeat 9 [fd 2 rt 10] fd 30 rt 180 pd
repeat 27 [fd 2 lt 10] lt 90 fd 10 pu bk 15 rt 90 pd circle2
end

to pc
op item 5 tf
end
```

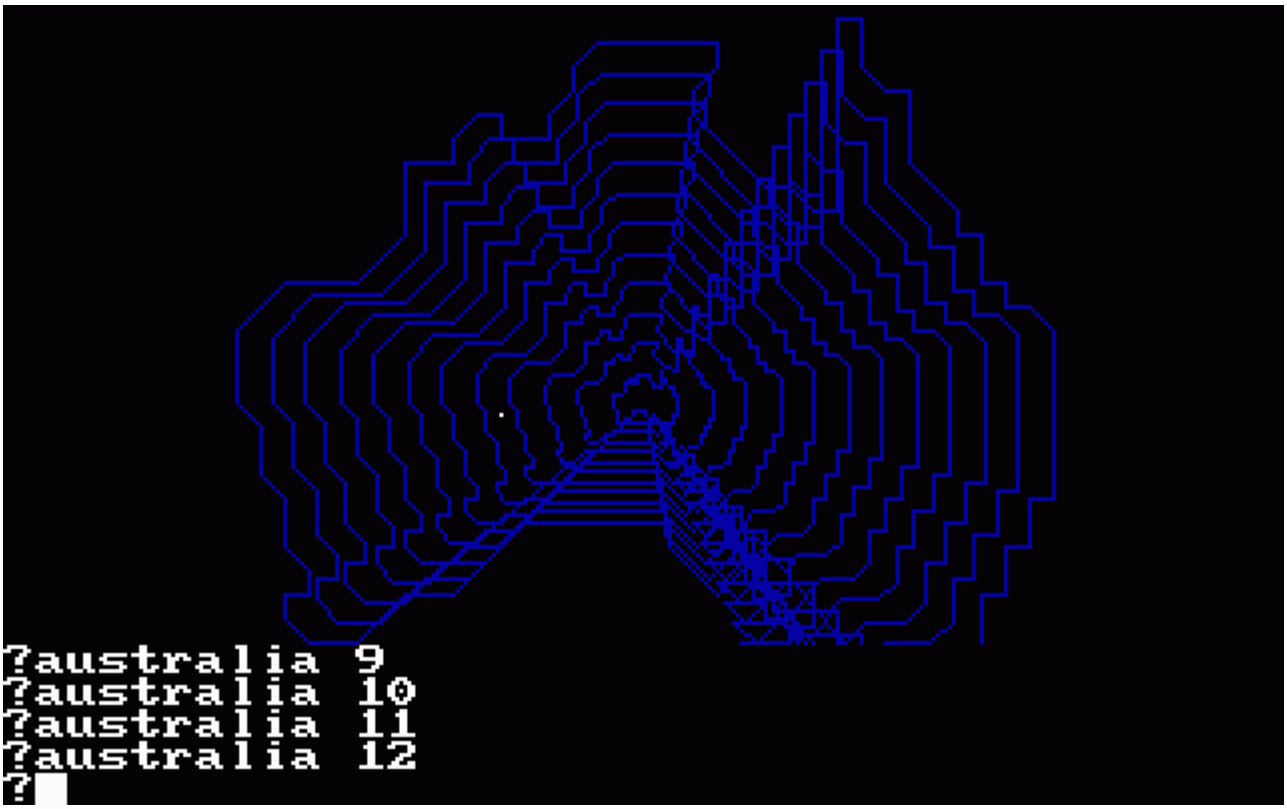

```
hex
---

to hex :sz :level
repeat 6
  [lt 30 fd :sz
   if :level > 0
      [lt 30 hex :sz :level - 1 rt 30]
   rt 90]
end


Example:

hex 20 3
```



Logo at work

The following program is a modified version of one that will be in a forthcoming Digital Research product. The picture shows what the end result will look like on your screen. A worthwhile project might be to plot some of the major cities, and show the 6 Australian states. The larger island is Tasmania, and the smaller is King island. Hats off to the winners of the 1983 America's Cup!

```
to australia :n
(local "p "h)
make "p pos make "h heading pu lt 90 fd 8.5 * :n rt 90 pd
fd 1.5 * :n rt 45 fd 1.414 * :n rt 45 fd 1.5 * :n lt 45
fd 1.414 * :n lt 45 fd 1.5 * :n rt 90 fd :n lt 90
fd 0.5 * :n rt 45 fd 0.707 * :n rt 45 fd 0.5 * :n rt 90
fd 0.5 * :n lt 90 fd :n lt 90 fd 0.5 * :n rt 45
fd 0.707 * :n lt 45 fd 0.5 * :n rt 45 fd 0.707 * :n rt 45
fd 2.5 * :n rt 90 fd 0.5 * :n rt 45 fd 0.707 * :n lt 45
fd 0.5 * :n lt 45 fd 2.121 * :n lt 45 fd 0.5 * :n rt 45
fd 0.707 * :n lt 45 fd 0.5 * :n lt 90 fd 4 * :n rt 90
fd 0.5 * :n rt 90 fd :n lt 45 fd 0.707 * :n lt 45
fd 0.5 * :n rt 90 fd 1.5 * :n lt 45 fd 1.414 * :n rt 45
fd 0.5 * :n lt 90 fd 0.5 * :n rt 90 fd :n lt 90
fd 0.5 * :n rt 90 fd 0.5 * :n lt 90 fd 0.5 * :n rt 45
fd 0.707 * :n rt 45 fd 3.5 * :n rt 90 fd 0.5 * :n lt 90
fd :n rt 90 fd 0.5 * :n lt 90 fd :n rt 90
fd 0.5 * :n lt 90 fd :n rt 45 fd 0.707 * :n rt 45
fd :n lt 45 fd 0.707 * :n rt 90 fd 0.707 * :n lt 90
fd 0.707 * :n rt 90 fd 0.707 * :n rt 45 fd 0.5 * :n lt 45
fd 0.707 * :n lt 45 fd 0.5 * :n rt 90 fd 1.5 * :n lt 135
fd 1.414 * :n rt 90 fd 1.414 * :n rt 45 fd 0.5 * :n lt 90
fd 3 * :n lt 45 fd 2.121 * :n rt 45 fd :n lt 45
fd 1.414 * :n rt 45 fd :n rt 45 fd 0.707 * :n rt 45
fd 0.5 * :n rt 90 fd 0.5 * :n lt 90 fd 0.5 * :n lt 45
fd 0.707 * :n rt 45 fd :n lt 45 fd 0.707 * :n rt 45
fd :n lt 45 fd 0.707 * :n rt 45 pu bk 5 * :n rt 90 fd 10.5 * :n
pd fd :n rt 135 fd 0.707 * :n rt 90 fd 0.707 * :n rt 45 pu
bk 1.5 * :n rt 90 fd 2 * :n pd fd 2 * :n rt 90 fd 1.5 * :n rt 90
fd :n rt 45 fd 0.707 * :n rt 45 fd 0.5 * :n lt 45
fd 0.707 * :n rt 45 pu setpos :p seth :h
end
```



Toolbox -- A collection of useful tools

Graphics tools

The procedure CIRCLE draws a circle with the given radius centered at the given point. The turtle's state and color is obeyed. When the circle is complete, the turtle is positioned at the center point with its initial heading.

```
to circle :center :radius
(local "p "amt)
make "p (list pen heading)
make "amt :radius * 1.75e-2  ; pi / 180
pu setpos :center setx xcor - :radius seth 0 setpen first :p
repeat 360
  [fd :amt rt 1]
pu setpos :center setpen first :p seth first bf :p
end
```

Example:

```
circle [10 30] 40
```

The procedure FILLED.CIR assumes the turtle's current position is the center, and draws a solid circle of the given radius. Again, the procedure obeys the current state of the turtle.

```
to filled.cir :radius
(local "x "p)
make "x pos
make "p pen
repeat 360
  [fd :radius pu setpos :x rt 1 setpen :p]
end
```

Example:

```
filled.cir 8
```

The procedures ARC.L and ARC.R are very similar to CIRCLE, but they require you to specify the number of degrees of arc to draw. The direction of the arc draw (to the left or right of the current turtle heading) is determined by

what procedure is used.

```
to arc.l :center :radius :angle
(local "p "amt)
make "p pen
make "amt :radius * 1.75e-2  ; pi / 180
pu setpos :center
fd :radius lt 90
setpen :p
repeat :angle
  [fd :amt lt 1]
end

to arc.r :center :radius :angle
(local "p "amt)
make "p pen
make "amt :radius * 1.75e-2  ; pi / 180
pu setpos :center
fd :radius rt 90
setpen :p
repeat :angle
  [fd :amt rt 1]
end
```

Examples:

```
arc.l [1 17] 13 45
arc.r [12 3] 21 7
```

The procedure PIE is identical to ARC.R, except that the endpoints of the arc are connected to the centerpoint. The procedure FILLED.PIE is to PIE what FILLED.CIR is to CIRCLE.

```
to pie :center :radius :angle
(local "p "amt)
make "p pen
make "amt :radius * 1.75e-2  ; pi / 180
pu setpos :center setpen :p
fd :radius rt 90
repeat :angle
  [fd :amt rt 1]
rt 90 fd :radius rt 180
end

to filled.pie :center :radius :angle
local "p
make "p pen
pu setpos :center setpen :p
repeat :angle
  [fd :radius pu setpos :center rt 1 setpen :p]
end
```

Examples:

```
pie [100 14] 7 32
filled.pie [0 0] 10 15
```

Workspace management tools

The procedure NERASE is, in some ways, the inverse of the primitive ERASE. Invoking this procedure will erase all procedures, except those listed. This is very useful if you have many procedures, but only want to retain a few. Note that NERASE claims to erase buried procedures, but it cannot. Neither can it erase itself!

```
to nerase :keeplist
(local "x "y "z)
make "x sort proclist
if wordp :keeplist
   [make "y (list :keeplist)]
   [make "y :keeplist]
make "y (se :y "nerase)
make "z count :y
repeat :z
  [if memberp (first :y) :x
      [if where = 1
          [make "x bf :x]
          [if where = count :x
```

```
                [make "x bl :x]
                [make "x (se
                  piece 1 (where - 1) :x
                  piece (where + 1)(count :x) :x)]]
    make "y bf :y]]
pr [These procedures will be erased:]
pr []
(pr :x)
pr []
type [Is this what you want (Y/N) ?]
if "y = lc rc
   [pr [y] erase :x]
   [pr [n]]
end
```

Example:

```
nerase [sort.procs verall]
```

The procedure SORT.PROCS simply sorts all procedures in the workspace into alphabetical order. This generally makes it easier to find procedures in the workspace.

```
to sort.procs
local "x
make "x sort proclist
if count :x = 1
   [stop]
repeat (count :x) - 1
  [(follow first :x first bf :x)
   make "x bf :x]
end
```

Example:

```
sort.procs
```

The procedure UNPKGALL takes the specified procedures out of any packages they are in. The example below, UNPKGALL PROCLIST, drops all packaging.

```
to unpkgall :set
local "x
if emptyp :set
   [stop]
make "x :set
repeat count :set
  [remprop first :x ".PAK make "x bf :x]
end
```

Example:

```
unpkgall proclist
```

The procedure VERALL inquires, for each procedure in turn, whether you want to erase that procedure. When finished, VERALL shows you all the procedures you have chosen to delete, and reconfirm your choice. This is a very popular procedure.

```
to verall
(local "x "y)
make "x sort proclist
make "y []
pr []
repeat count :x
  [type (se [Erase] uc first :x [(Y\/N) ?\ ])
   if "y = lc rc
      [pr [y] make "y lput first :x :y]
      [pr [n]]
   make "x bf :x]
pr []
pr [These procedures will be erased:]
pr []
(pr sort :y)
pr []
type [Is this what you want (Y\/N) ?\ ]
if "y = lc rc
   [pr [y] er :y]
   [pr [n]]
```

```
end

Example:

verall


The  procedure USES lists the title line of each procedure in  your  workspace
and,  indented  underneath,  any  procedures that  each  of  those  procedures
references,  including  themselves  if recursive. After  each  procedure,  the
system waits for a keypress to continue.

to uses
(local "x "y)
make "x sort proclist
repeat count :x
  [pocall first :x
   make "x bf :x
   pr []
   if not emptyp :x
      [make "y rc pr []]]
end

Example:

uses


List manipulation tools
-----------------------

The  procedures  DELETE  and REMOVE are very similar in  function.  They  both
remove  an  object  from  something.  The  former  removing  only  the   first
occurrence, the latter removing all occurrences.

to delete :object :objlist
if not memberp :object :objlist
   [op :objlist]
if emptyp :objlist
   [op []]
if :object = first :objlist
   [op bf :objlist]
op fput first :objlist delete :object bf :objlist
end

to remove :object :objlist
(local "n "m)
if not memberp :object :objlist
   [op :objlist]
make "n where
make "m count :objlist
if :n = 1
   [op remove :object bf :objlist]
if :n = :m
   [op bl :objlist]
op remove :object (se
   piece 1 (:n - 1) :objlist
   piece (:n + 1) :m :objlist
end

Examples:

delete "cat [dog cat pony]
remove "cat ["cat "dog "cat]


The  procedures EVERY, SOME, and SUBSET work by applying some test  (called  a
predicate  in  Logo) to each element of the list being tested.  The  procedure
EVERY returns "TRUE if the predicate returns "TRUE when applied to every  item
in  the list. The procedure SOME returns "TRUE if the predicate returns  "TRUE
when  applied to any element of the list. The procedure SUBSET returns a  list
of those elements which the predicate returned "TRUE when applied to.

to every :objlist# :predicate#
repeat count :objlist#
  [if not run (se :predicate# "first (list :objlist#))
      [op "FALSE]
   make "objlist# bf :objlist#]
op "TRUE
end
```

```
to some :objlist# :predicate#
repeat count :objlist#
  [if run (se :predicate# "first (list :objlist#))
      [op "TRUE]
   make "objlist# bf :objlist#]
op "FALSE
end

to subset :objlist# :predicate#
local "x##
if emptyp :objlist#
   [op []]
make "x## []
repeat count :objlist#
  [if run (se :predicate# "first (list :objlist#))
      [make "x## lput first :objlist# :x##]
   make "objlist# bf :objlist#]
op :x##
end
```

Examples:

```
every [1 2 a 3 b] "numberp
some [a [1] [2 3]] "wordp
subset [[1] 2 [3 4] 5] "listp
```

The procedures INTERSECTION and UNION compare the membership of two lists. Only those items in both lists are returned by INTERSECTION, while all the items in both lists are returned by UNION (with no duplicates in the return list). There is a very special case under which UNION returns duplicate items. This occurs when :set2 has more than 2 extra items than :set1, and there are duplicates in these extra items. Fixing UNION would be a great project in list processing.

```
to intersection :set1 :set2
if or (emptyp :set1) (emptyp :set2)
   [op []]
if memberp (first :set1) :set2
   [op (se (list first :set1) intersection (bf :set1) :set2]
op intersection (bf :set1) :set2
end

to union :set1 :set2
if emptyp :set1
   [op :set2]
if memberp (first :set1) :set2
   [op union bf :set1 :set2]
op union bf :set1 (se (list first :set1) :set2)
end
```

Examples:

```
intersection [1 2 3][2 4 6]
union [1 2 3 4][1 2 a b]
```

The procedure REPLACE changes all occurrences of the first object to the second object in the list. This is the classic search and replace function. Note that his function is level sensitive, so that no replacement occurs for items in sublists.

```
to replace :old :new :in
if emptyp :in
   [op []]
if :old = :new
   [op :in]
if wordp :in
   [if :old = :in
       [op :new]
       [op :in]]
op fput (replace :old :new (first :in))
        (replace :old :new (bf :in))
end
```

Example:

```
replace "1 "2 [1 2 3 2 1]
```

The procedure REVERSE returns a list with the elements in reverse order of the

elements in the list passed to. It does not reverse the order of elements in sublists.

```
to reverse :set
if count :set < 2
   [op :set]
op (se (list last :set) reverse bl :set)
end
```

Example:

```
reverse [eat snails everyday]
```

Mapping tools

The procedure MAPFIRST returns a list whose elements are created by applying the function to each element of the maplist in turn. Note that the function, whether a built-in primitive or a user-defined function, must return a value, or an error occurs.

```
to mapfirst :function# :maplist#
if emptyp :maplist#
   [op []]
op (se
   (run
   (list :function# "first :mapfirst#))
   (mapfirst :function# bf :maplist#))
end
```

Example:

```
mapfirst "sin [0 45 90 -90]
```

The procedures MAP and MPAC are similar to MAPFIRST but, instead of having the function operate on the individual elements of the list, they operate on the whole list and then successive BUTFIRST's of the list. The two forms are provided, to give you a chance to compare how they operate. There might be cases where one would be better to use than the other. The same restriction on the functions for MAPFIRST apply to the functions for both of these as well.

```
to map :function# :maplist#
if emptyp :maplist#
   [op []]
op (se
   (run
   (list :function# :maplist#))
   (map :function# bf :maplist#))
end

to mapc :function# :maplist#
repeat count :maplist#
  [run lput :maplist# :function#
   make "maplist# bf :maplist#]
end
```

Examples:

```
map "xyzzy [2 12 a b]
mapc "random [6 6 6]
```

The procedure APPLY is just like MAP, except the function you use does NOT need to output values. Building an APPLYFIRST procedure is left to you.

```
to apply :function# :maplist#
if emptyp :maplist#
   [stop]
run (list :function# :maplist#)
apply :function# bf :maplist#
end
```

Example:

```
apply "print [This is a triangle]
```

These procedures are very similar to the mapping functions of Lisp, and should prove very useful in constructing Artificial Intelligence programs.

Flow of control tools

The procedure COND works just like the Lisp primitive of the same name -- if the first condition evaluates to "TRUE, COND returns the value from evaluating the first body. Otherwise, it tests the second condition, and so on, until it finds a "TRUE condition or runs out of possibilities. If the latter case occurs, COND returns []. Remember that the evaluated bodies must return a value, so you might need to use the RISE function described below.

```
to cond :condlist#
; Format of :condlist# is [test1 result1 test2 result2 ...]
local "cl#
if (remainder (count :condlist#) 2) > 0
   [op [condlist unbalanced]]
make "cl# :condlist#
label "loopst
if emptyp :cl#
   [stop]
if run first :cl#
   [run first bf :cl# stop]
make "cl# bf bf :cl#
go "loopst
end
```

Example:

```
cond [[xy 1][yz 2]["TRUE 0]]
```

The LOOP procedure repeats forever the evaluation of :body#. It can be exited only by STOP, OP, THROW "TOPLEVEL, or a ^G (Control-G).

```
to loop :body#
label "loopst
run :body#
go "loopst
end
```

Example:

```
loop [op run readline]
```

The procedures UNTIL and WHILE are two useful loop constructs that allow you to express certain ideas with great clarity. The :cond# list can contain multiple statements, as long as evaluating it returns a "TRUE or "FALSE value. In WHILE, the condition is tested and, while "TRUE, the body is performed. In UNTIL, the body is performed, then the condition is checked until the condition becomes "TRUE.

```
to until :cond# :body#
label "loopst
run :body#
if not run :cond#
   [go "loopst]
end

to while :cond# :body#
label "loopst
if run :cond#
   [run :body#]
   [stop]
go "loopst
end
```

Examples:

```
until [:n < 10][make "n :n + 1]
while [not night][measure light]
```

Miscellaneous tools

The procedures ASK and ASKYN type out the question and wait for a response. The procedure ASK returns the first item of the response, while ASKYN returns "TRUE if the response was 'y' (upper- or lowercase), and "FALSE otherwise.

```
to ask :question
(type :question)
op first rl
end

to askyn :question
local "ans
(type :question)
make "ans lc rc
(pr :ans)
op :ans = "y
end
```

Examples:

```
ask [What should I know?]
askyn [Play again (Y/N) ?]
```

The procedure FORGET completely eliminates a word from the workspace, erasing any procedure definition, value, and bound properties. WARNING: This procedure can even eliminate primitive functions. Use it carefully, or you might blow the system away.

```
to forget :object
if (se :object) = [forget]
   [stop]
repeat (count plist :object) / 2
  [remprop :object first plist :object]
end
```

Example:

```
forget "xyzzy
```

The INKEY procedure was provided for people familiar with the BASIC function of the same name. If a key is pressed, it is returned; otherwise, [] is returned. Unlike READCHAR, this function does not wait for a key to be pressed.

```
to inkey
if keyp
   [op rc]
   [op []]
end
```

Example:

```
inkey
```

The MENU procedure allows you to quickly construct a simple selection menu. The format of :menulist# is [choice1 action1 choice2 action2 ...] where the choices are objects that are printed out and the actions are the list of statements to be executed if the associated choice is picked. When printing out the choices, MENU numbers each one, then asks the user to type in the number of the choice he wants to take.

```
to menu :menulist#
; Format of :menulist# is [choice action ...]
(local "l# "m# "n#)
if emptyp :menulist#
   [stop]
label "loopst
make "l# :menulist#
make "n# 0
pr []
repeat (count :menulist#) / 2
  [make "n# :n# + 1
   (pr :n# first :l#)
   make "l# bf bf :l#]
pr []
type [Enter choice:]
make "m# first rl
if not numberp :m#
   [go "loopst]
make "m# int :m#
if or (:m# < 1) (:m# > :n#)
   [go "loopst]
run (se item :m# * 2 :menulist#)
```

```
end

Example:

menu [[on][sw.on][off][sw.off]]
```

Procedures PUSH and POP implement software stacks. They allow for multiple stacks, so you must specify the stack name.

```
to push :object :stack
if emptyp (plist :stack)
   [make :stack []]
make :stack fput :object thing :stack
end

to pop :stack
local "pop##
if emptyp thing :stack
   [op []]
make "pop## first thing :stack
make :stack bf thing :stack
op :pop##
end

Examples:

push 123.3 "rpn
pop "addresses
```

The procedure SINK allows you to throw away the returned value of a procedure, using it only for the side-effects. The example below, SINK RC, simply waits for a key to be pressed.

```
to sink :object
end

Example:

sink rc
```

The procedure RISE does just the opposite. It allows you to use functions that don't return a value (in places that require values) by returning []. In many versions of Logo, procedures that return a value are called functions, while those that return no value are called commands. These mirror-image procedures allow you to interchange the usage of the two types of procedures. These two procedures allow Logo to be more Lisp-like (always a desirable goal).

```
to rise :object#
local "y##
catch "error [(make "y## run (se :object#)) op :y##]
op []
end

Example:

rise [pr "xyzzy]
```

Toolbox -- An example of tool usage

Some people find they can understand something much better if they can see an example of it in operation. In fact, given Logo's use of familiar objects to express abstract ideas, many of you using the language should learn this way. Therefore, we have included an example program using a number of these tools. The program PLAY implements the classic game REVERSE -- a simple thinking game. Examine the procedures, and you will see how using the tools makes the code easier to understand.

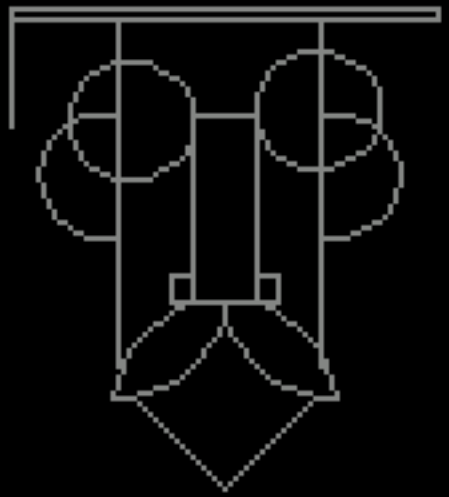
```
to ask :question
; Returns a user response to a question
(type :question)
op first rl
end

to askyn :question
; Returns TRUE if user answers question yes
local "ans
```

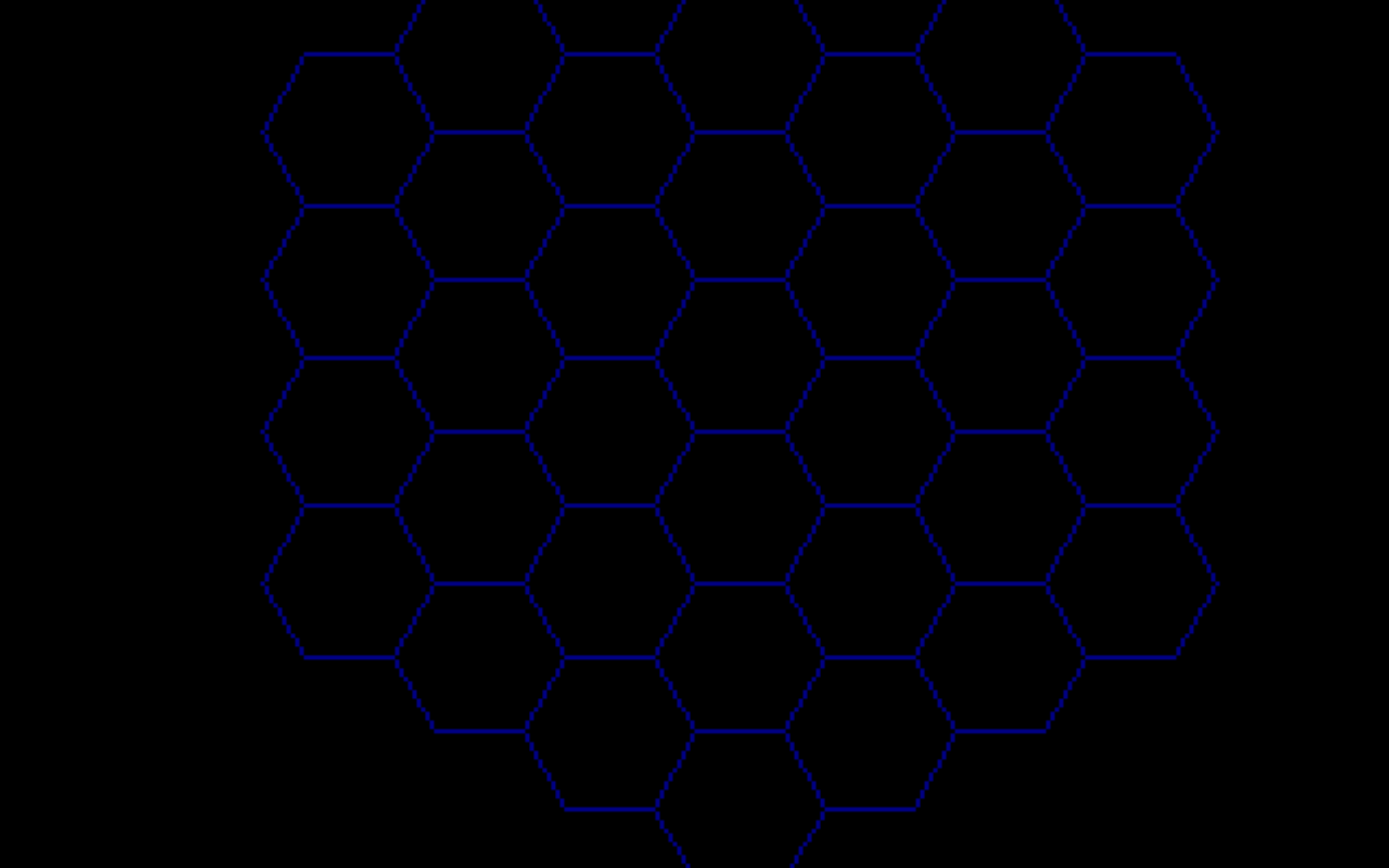
```
(type :question)
make "ans lc rc
(pr :ans)
op :ans = "y
end

to check_for_win
if not (:board = [0 1 2 3 4 5 6 7 8 9])
   [stop]
make "game_over "TRUE
pr []
show :board
pr []
(pr [You've done it in only\ ] :move - 1 [\ moves!])
pr []
end

to reverse :set
; Returns a reversed copy of the input list
if emptyp :set
   [op []]
op (se (list last :set) reverse bl :set)
end

to loop :body#
; Repeat instructions in body (until STOP or OP)
repeat 1 / 0 :body#
end

to until :cond# :body#
; Perform the body until the condition is true
label "loopst
run :body#
if not run :cond#
   [go "loopst]
end

to sink :x
end

to explain_rules
ct
pr [This is the game of REVERSE]
pr []
pr [I will give you a scrambled list of 10 numbers, and you have]
pr [to put them in order, from the smallest to the largest.]
pr []
pr [The tricky part is that the only thing you can do is to reverse]
pr [some or all of the numbers. For example, if you have the list]
pr []
pr [[1 3 2 6 4 9 8 5 0 7]]
pr []
pr [and your reverse the first 5 numbers, the new list will be:]
pr []
pr [[4 6 2 3 1 9 8 5 0 7]]
pr []
pr [If you now reverse the first 3 numbers, the list becomes:]
pr []
pr [[2 6 4 3 1 9 8 5 0 7]]
pr []
pr []
pr [Now, press the RETURN key to begin]
sink rq
end

to init_game
ct
pr [The game of REVERSE]
pr []
make "board shuffle [0 1 2 3 4 5 6 7 8 9]
make "game_over "FALSE
make "move 1
end

to move
loop
  [(pr [List is:] (list :board) [\ move:\ ] :move)
    pr []
   make "n ask [How many items to reverse ?]
   if not numberp :n
      [make "n 11]
```

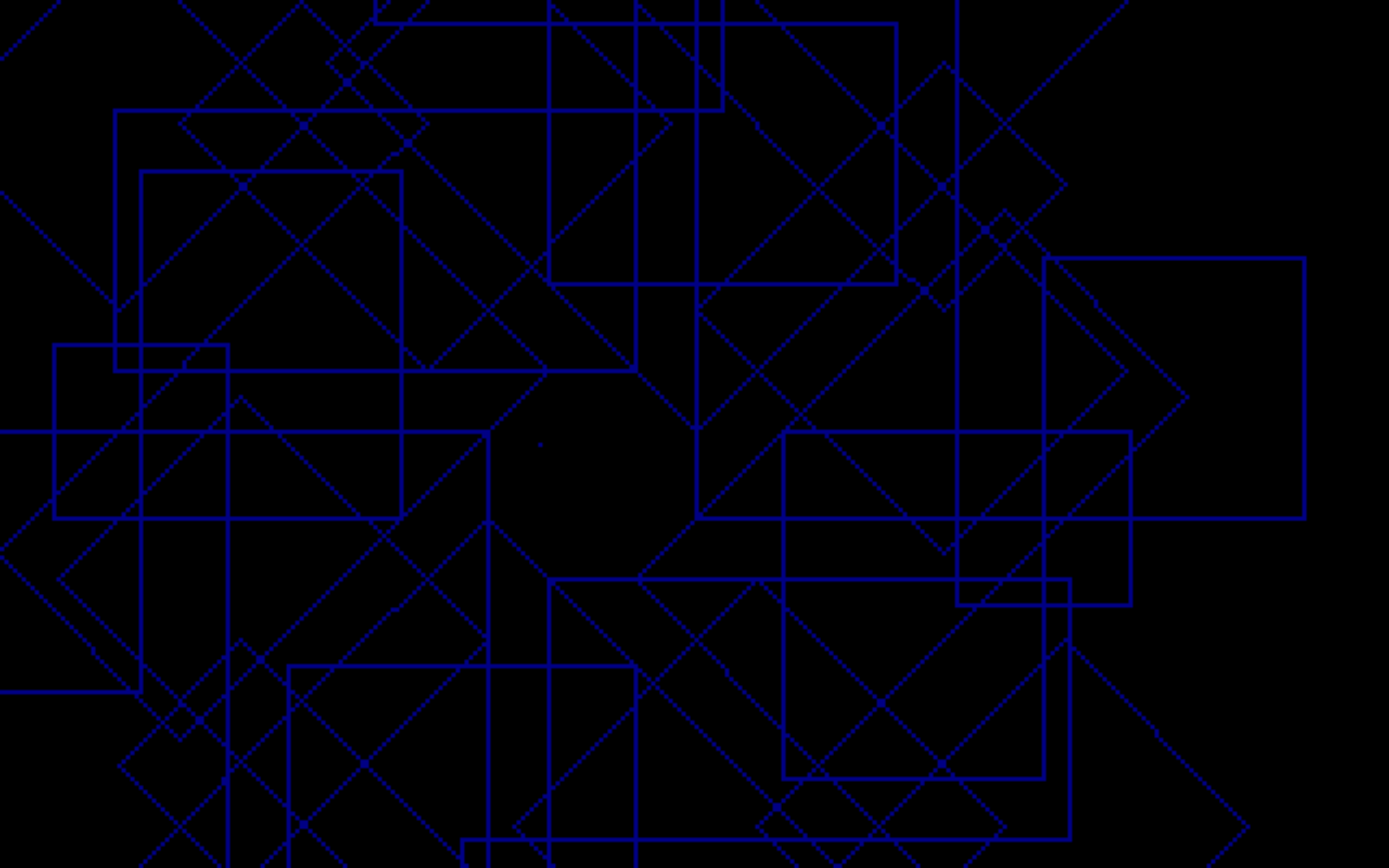
```
    if :n > 10
       [pr [Sorry, I can't do that.]]
       [stop]]
  make "move :move + 1
  if :n < 2
     [stop]
  make "board
  if :n = 10
     [reverse :board]
     [(se reverse piece 1 :n :board piece (:n + 1) 10 :board]
  end

  to play
  ; ...the game of REVERSE
  (local "board "game_over "move "n "m)
  explain_rules
  loop
    [init_game
     until [:game_over][move check_for_win]
     if not askyn
        [Play again (Y/N) ?]
        [stop]]
  end


  EOF
```

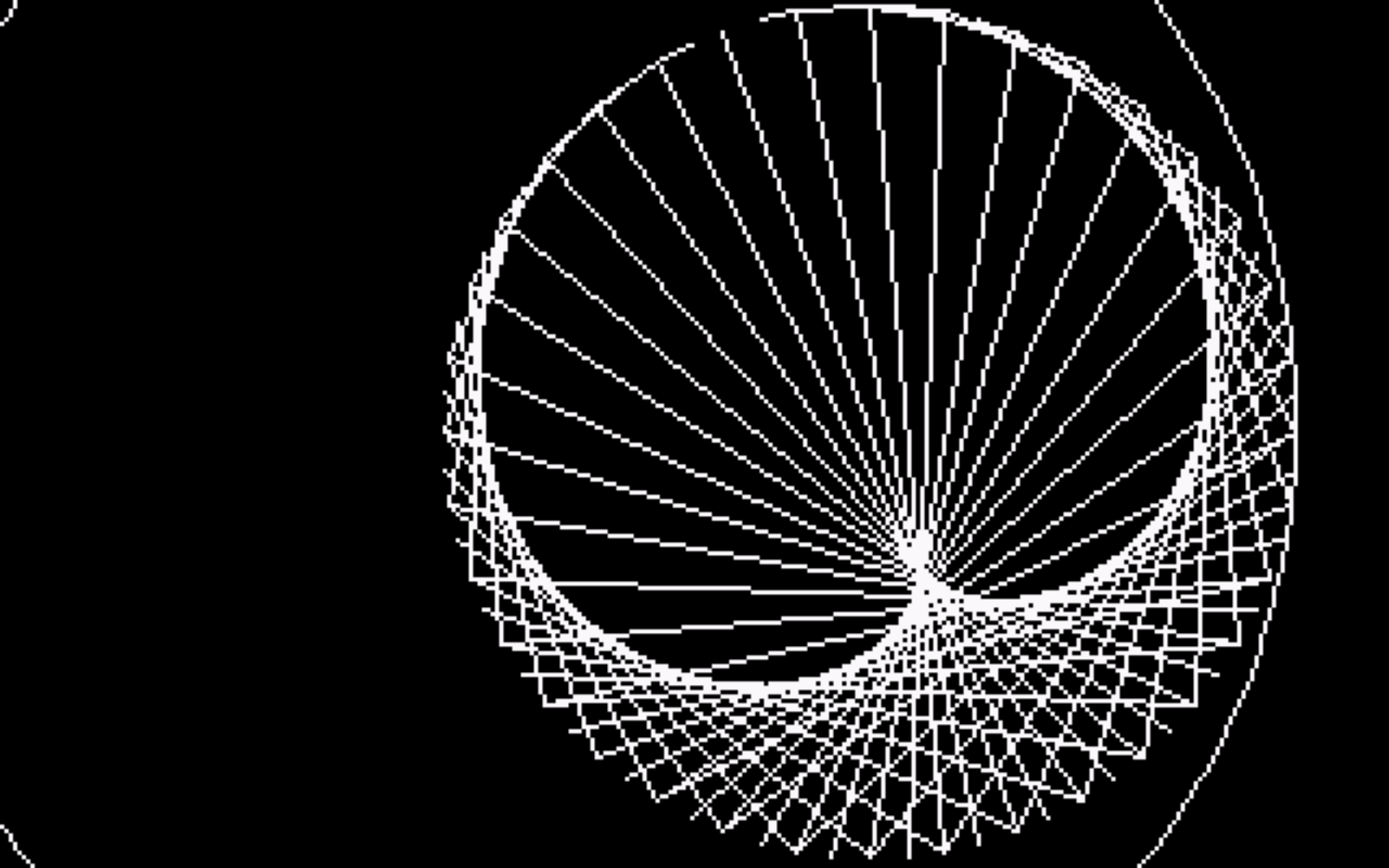
Digital Research presents:

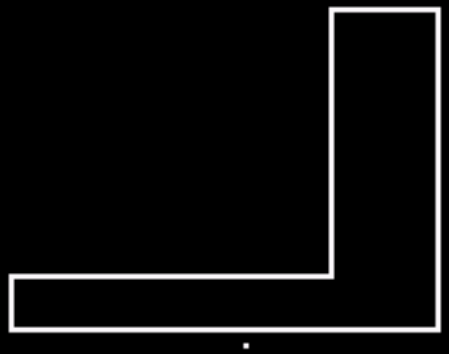
Dr LOGO











```
L is now a shape.
?restore "L
?rotate "L "xz 90
?
```