Logo
PROGRAMMING
WITH MSW Logo

Computer Programming Fundamentals

N V Fitton
Northern Virginia Community College
vfitton@nvcc.edu
www.nvcc.edu/home/vfitton

MSW Logo comes from www.softronix.com/logo.html

Why learn Logo?

* because it's fun

* because it makes us want to think clearly

* because it's real computer science

Logo is easy to learn, as programming languages go, yet it also has enough depth to do virtually anything that can be done in any computer language. There is way more to it than I can show you or tell you, even if we took an entire semester.

I will be happy to help you, and you can get much more help by helping yourself:

* There is a very extensive help system built into the Logo interpreter.

* There are web sites galore, many with program examples.

* My own web site offers code for things not included here.

`www.nvcc.edu/home/vfitton/logo`

Contents

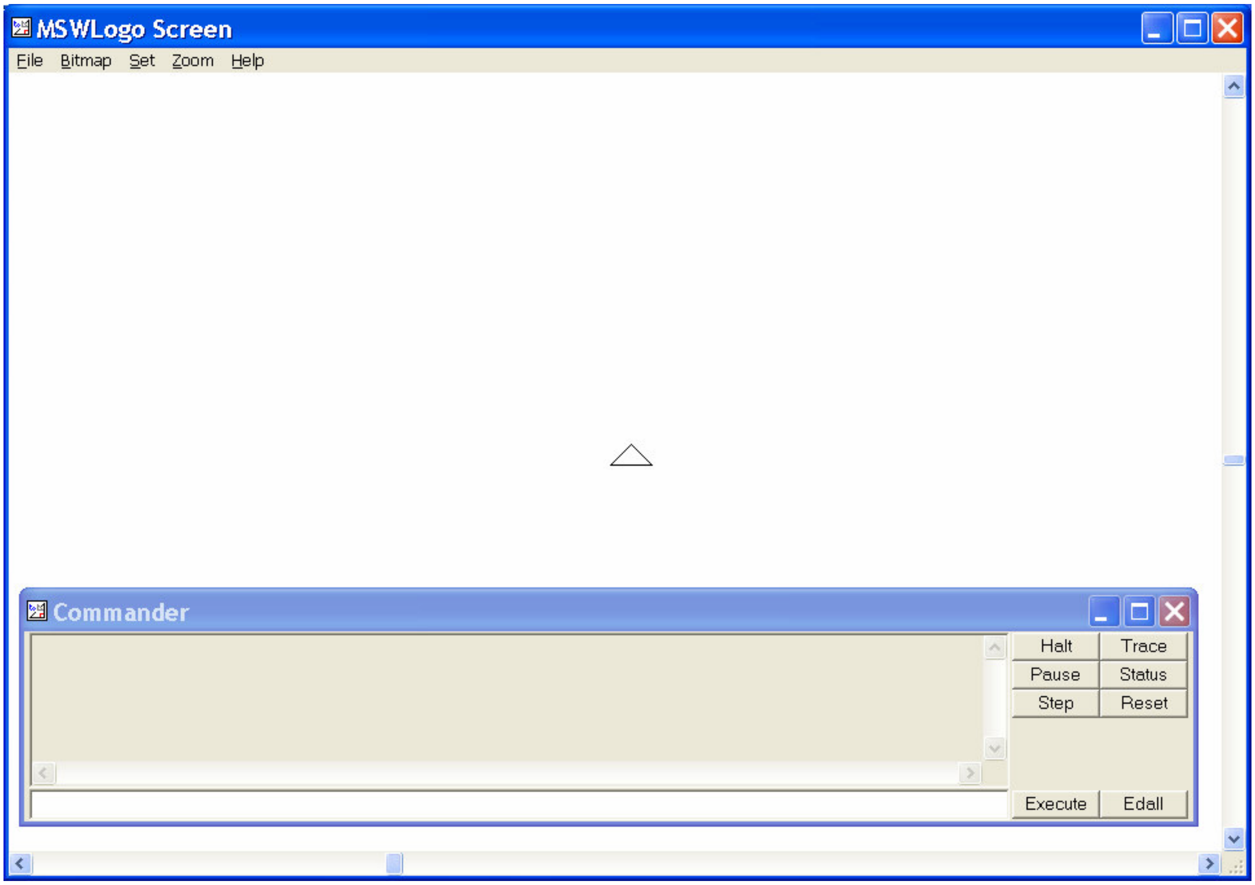
Programming fundamentals

Logo programming

If Logo is not already on your computer, you can get it for free from its makers at
`www.softronix.com/mswlogo.html`

The installation process puts a Logo icon (as on the cover of this document) on your computer desktop.

Here's the MSW Logo screen in two parts:
drawing window above, with triangle-shaped TURTLE in center
Commander window below



Write commands in *command line*, i.e., text box at bottom of Commander window.
Press Enter or click Execute to run command written there.

It's OK to write and run more than one command on line at a time.

Command history appears in gray box.
Click a line in the history to make it jump to the command line,
then make changes (or not) then press Enter or click Execute.

BASICS Don't forget spaces between words!	
`forward 50` *or* *or* `fd 50`	*go forward 50 steps*
`right 90` *or* *or* `rt 90`	*right turn 90 degrees*
`fd 100 rt 90 fd 100 rt 90` *several steps on command line are OK*	*makes two sides of a square*
`left 30` *or* `lt 30`	*left turn 30 degrees*
`back 100` *or* `bk 100`	*go backward 100 steps*
`clearscreen` *or* `cs`	*erases all drawings and sets turtle at center* *useful at *beginning* of multi-step commands!*

PROGRAMMING FUNDAMENTAL 1: Sequence

You have already experienced this one:

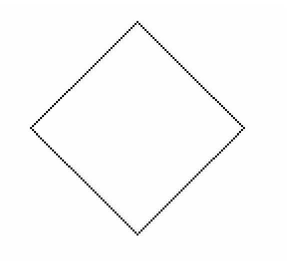
Computers do commands *in sequence.*

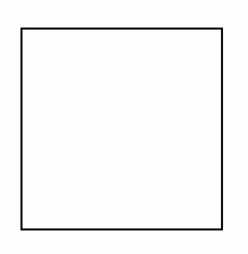
When you're designing a Logo drawing, think of the steps in the order that the computer should do them.

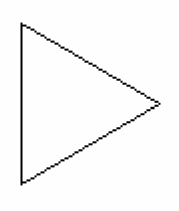
Try to map out all the steps in advance, then enter them at the command line. Find out how accurate your thinking is!

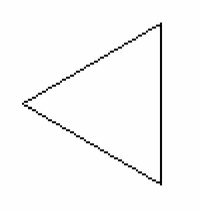
The computer is your faithful servant: it will do exactly what you tell it to do. (And that's all it will do.)

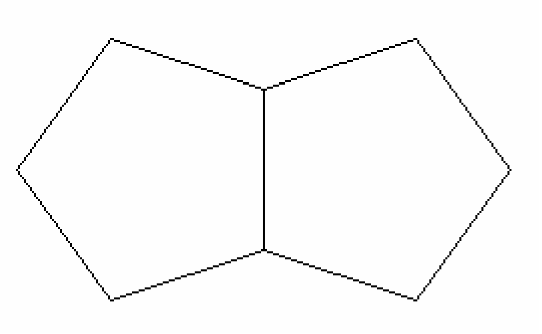
Can you draw these?

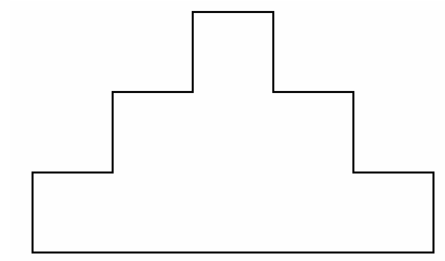




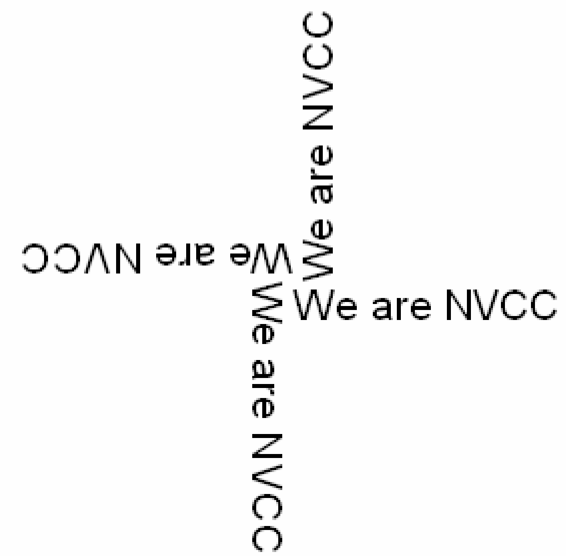


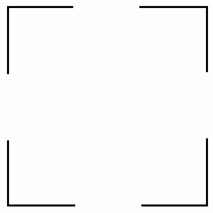


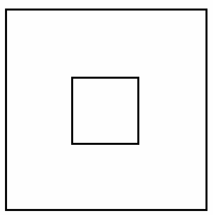




BASICALLY HANDY	
`clearscreen` *or* `cs`	*erases all drawings and sets turtle at center* *useful at *beginning* of multi-step commands!*
`penup` *or* `pu`	*pick pen up,* *so you can move turtle without leaving tracks*
`pendown` *or* `pd`	*put pen down*
`hideturtle` *or* `ht`	*hides the turtle so you can admire your drawing*
`showturtle` *or* `st`	*shows the turtle so you can see what you're doing*
`setpensize [3 3]`	*makes pen larger, easier to see!* *default pen size is [1 1]*
`label [whatever]`	*writes* whatever *at cursor location* *text goes in direction that turtle is pointing*
`wait 20`	*put this between commands to slow turtle down* *so you can see what it's doing, for example:* *fd 100 rt 90 wait 20 fd 100*







PROGRAMMING FUNDAMENTAL 2: Repetition

You already have experience with this one, too:

We often repeat a sequence of commands.

Many of the things that computer programs do, they do over and over. Just about every programming system has a way of doing this repetition, or *iteration*, as computer scientists call it. We tell Logo, for example:

```
repeat 4 [fd 100 rt 90]
```

or

```
repeat 4 [fd 100 rt 90 wait 20]
```

to save ourselves some typing in making a square.

The general form is:

```
repeat number [commands]
```

We must use the keyword `repeat` followed by a `number` and then a sequence of commands in `[square brackets]`.

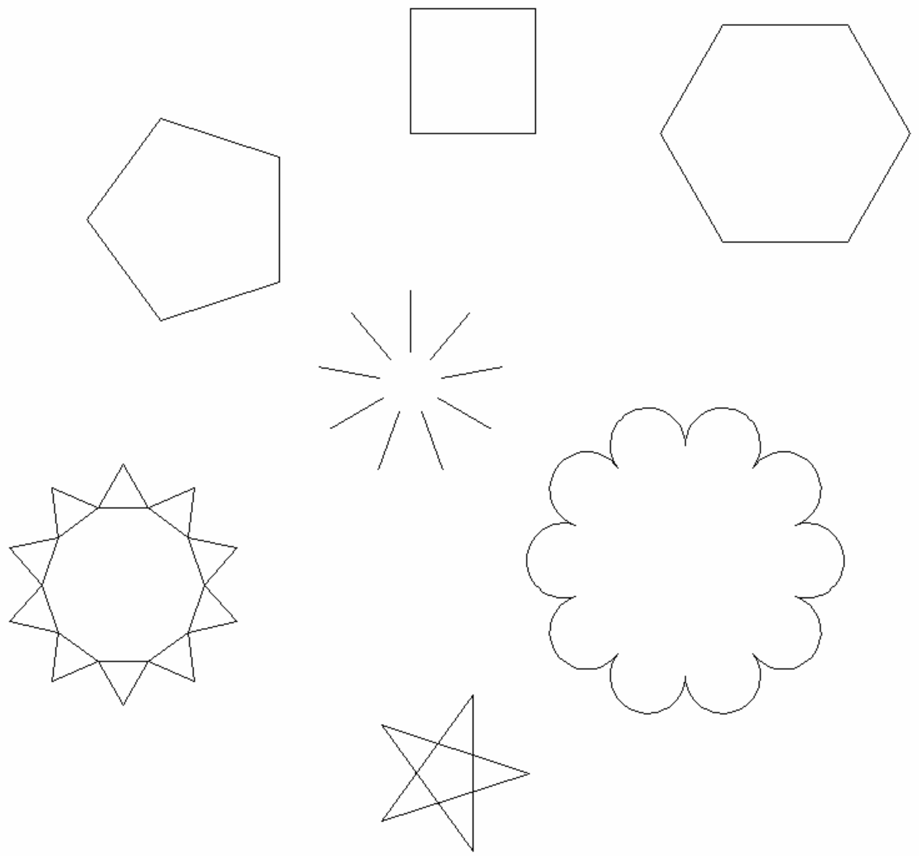
Often we have repeat within repeat. (This is called nesting.) What does this do? Think about it, then try it:

```
repeat 4 [repeat 4 [fd 100 rt 90] rt 90]
```

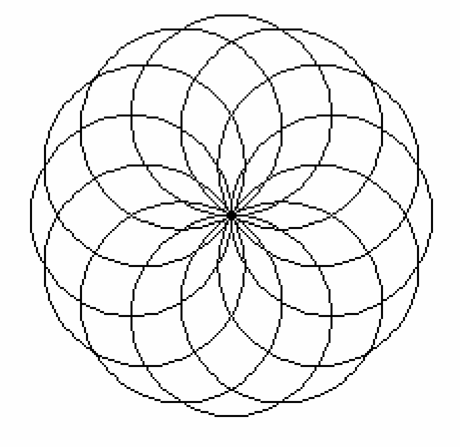
How about this? How is it different?

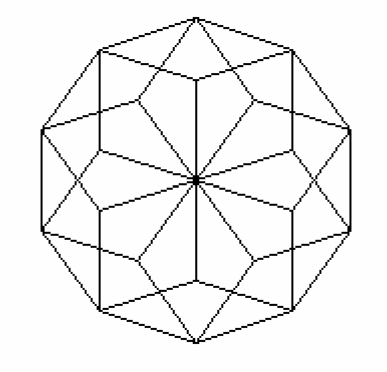
```
repeat 8 [repeat 4 [fd 100 rt 90] rt 45]
```

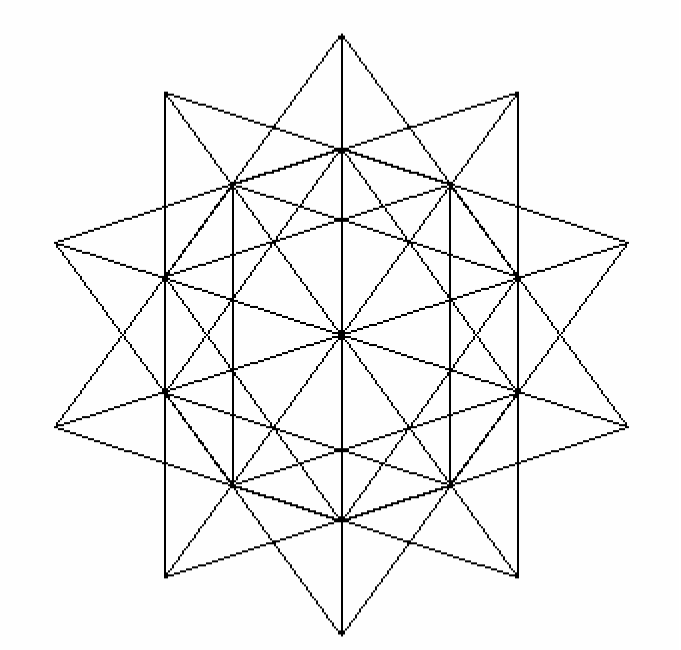
All of these drawings use the *repeat* command.
The burst shape at the center used the *penup* and *pendown* commands.
In between the drawings, I picked up the pen and moved the turtle.

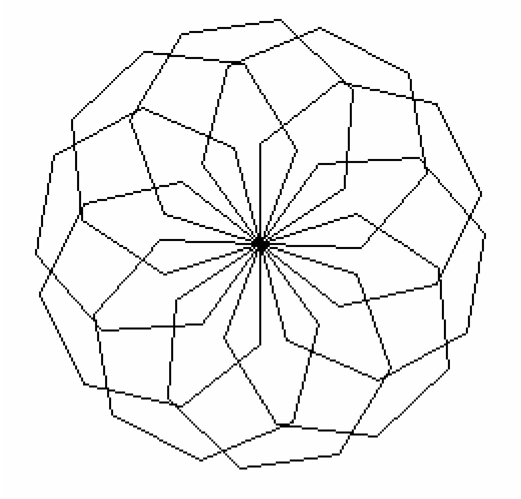


Drawings made using <u>repeat</u> [ ] or <u>repeat</u> [<u>repeat</u> [ ]]









PROGRAMMING FUNDAMENTAL 3: Subprograms

For convenience and concision, we can add a new command to Logo's language.

> A *subprogram* is a named sequence of steps for another program to execute.

Other names for subprograms are *procedures* and *functions.*

In Logo, you tell the computer how *to do* something — for example,

```
to square
repeat 4 [fd 100 rt 90]
end
```

Once you have described your procedure to Logo, you can enter its name on the command line just as you would any of the built-in things Logo knows how to do. In this case, you would say on the command line,

```
square
```

and Logo looks up your commands for doing a square.

Click the button that says Edall (for *edit all)* to bring up Logo's built-in editor. (If your Logo doesn't have an Edall button, write *edall* on the command line.) Here is the required structure of the subprogram:

```
to procedurename
steps of your procedure here
end
```

Your procedure or subprogram must start with the word `to,` then a name you make up. Then come all the same steps you would write on the command line. Your procedure must end with the word `end.`

Can one of your subprogram's steps be another subprogram? Yes!

Think: what will Logo do with this?

```
to flower
repeat 12 [square rt 30]
end
```

PROGRAMMING FUNDAMENTAL 4: Variables

We use the same procedure on different values, or *variables.*

We don't want every square to be the same size — we want variety. In Logo, we create variables whose values we can change.

We'll use the same *square* procedure with a small change:

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

We give Logo a replacement value for `:n` on the command line.

```
square 50
square 75
square 100
```

Logo puts our number wherever the variable `:n` appears. You can call the variable a short abstract name like `:n` or `:x` or a longer, more meaningful one, like `:length` — whichever you prefer, but don't forget the colon : and don't put space between the colon and the variable name.

A procedure can be used with more than one variable. Here's a challenge: Write a subprogram that makes *any* regular polygon — a triangle, square, pentagon, hexagon, — and makes it any size. Your procedure might start like this:

```
to polygon :sides :length
```

To write this subprogram, you must figure out in advance what you will do with both variables. What to do with `:length` is easier. What you will do with `:sides` has to do with the fact that a complete revolution is a turn of 360 degrees. You'll need a little arithmetic along with geometry. Use symbols + and – to add and subtract and * and / to multiply and divide.

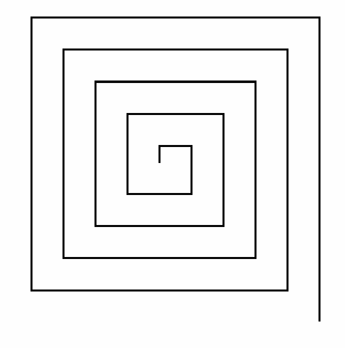
You can draw a reasonable facsimile of a circle with your polygon procedure. How?

PROGRAMMING FUNDAMENTAL 5: Decision-making

A program needs to be able to change course depending on the situation.

Decision-making and variables go together. Here, for example, is a framework for drawing a spiral. It has a loop, a variation on the repetition shown earlier, and the body of the loop is for you to fill in.

```
to spiral
  make "n 1
    while [:n < 20] [
      ; what goes here??
      make "n :n + 1
    ]
end
```



The code above shows several new features of the *syntax* of MSW Logo.

We set a variable to a new value by saying `make`, then the variable's name preceded by a double quote " rather than a colon :

```
make "n 1
```

We use a variable, though, with a colon : in front of its name.

```
while [:n < 20]
```

The code bracketed after `while [condition]` is executed while the condition is true. When it's no longer true, because (in this case) the value of `:n` grows greater than 20, the code following the bracket is executed.

This code has a *comment,* a reminder or notice for human readers, which begins with a semicolon ; Anything on the same line following the semicolon is ignored by Logo, but may be very helpful for our understanding.

Here's a funny bit of code for something called a *random walk.*

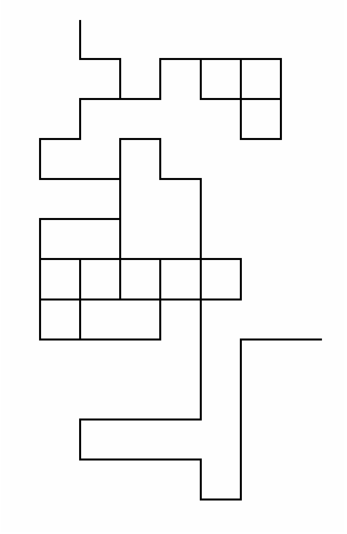
```
to randomwalk

repeat 100 [
  make "r random 3
  if :r = 0 [fd 20]
  if :r = 1 [rt 90 fd 20]
  if :r = 2 [lt 90 fd 20]
]

end
```

This code shows *`if`* statements, that have code executed only when a given condition is true.

It also shows a Logo built-in that generates random numbers. The statement *`random 3`* produces a 0, 1, or 2. The procedure then decides which way to go "at random." Can a random walk through the business pages produce results as good as a stockbroker's?

The statement *`random 6`* produces a number chosen from 0, 1, 2, 3, 4, 5. So what do you say to make Logo roll dice?

Sample output of the random walk above:



For a more random walk than this, see the program *bug* on my web site.

Strings

In computer science, any sequence of characters *like this* goes by the name of *string.* Dealing with strings is fundamental: for example, how does the computer understand the commands that we give it? It has to break them into pieces and figure out what the pieces mean according to what they are and where they are.

Counting the characters is the most basic of all string processes. The answer to the question `howlong "abcdefg` is given by the following procedure:

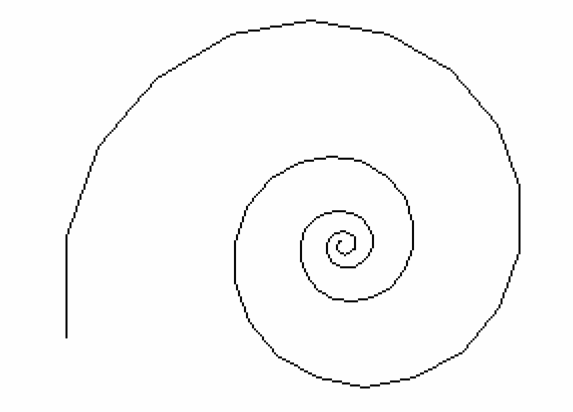
```
to howlong :s

   make "count 0
      ; why zero?

   while [not emptyp :s] [
      make "count :count + 1
      print first :s
         ; it's helpful to see it
      make "s butfirst :s
         ; butfirst means all but
         ; the first of something
   ]
   print (sentence :s "has :c "letters)
end
```

The command *print* writes a result on the command line, and we can see a sequence of results in the command history. When you're writing and troubleshooting a procedure like this, it's useful to show yourself the intermediate values, as we do in the statement *print first :s.* It wouldn't hurt to *print butfirst :s* at this stage, too.

Here's a much more challenging task: use the new ideas shown above to make a procdure that counts the number of occurrences of a character within a string. For example, the command `howmany "a "yabbadabba` would give the result 4.

Recursive procedures

Can a procedure call itself? Why not? When it does, it is called a recursive procedure because the call recurs — there's a recurrence of the procedure within the procedure.



This pleasing picture was produced by the following procedure with the call `spiral 50 :`

```
     to spiralR :n
        if :n < 1 [stop]
        fd :n
        rt 20
        spiralR 0.95 * :n
     end
```

Why does this work? What values does the procedure operate on? Why doesn't it just go on forever? It is a deep fact of computer science that every procedure written with *repeat* can also be written as a recursive procedure.

You might want to rewrite the procedure with variables instead of constants so that you can more easily conduct spiral experiments.

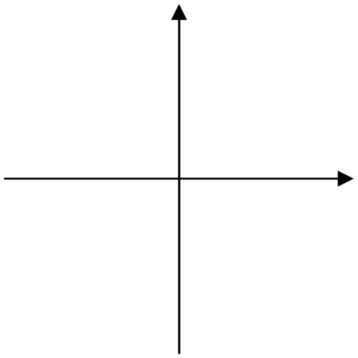
The following is a rewrite of the procedure *howlong* on the preceding page. Some people believe that the recursive version shows a more natural way to think and is easier to understand. (It's certainly shorter.) What do you think?

```
     to howlongR :s
        if emptyp :s [output 0]
        output 1 + howlongR butfirst :s
     end
```

Challenge: Use this procedure as a model for one that computes *n!* or one that concatenates (sticks together) two strings.

Turtle geometry

Many programming systems work on the same kind of two-axis *xy* coordinate plane that you work with in algebra.



x is horizontal
y is vertical

0 0 is the center, or origin
(no comma or parentheses here!)

In its centered, zoom-"normal" state,
Logo's drawing screen shows an area
about 150 points up or down
and 300 points right or left
from the center.

The turtle can be directed with *headings* that correspond to a compass rose,
with 0 or 360 degrees pointing straight up, 90 degrees straight to the right, and so on.
You can set a variable to a number between 0 and 360 and then walk that way.

TURTLE GEOMETRY	
`setx 100`	*set turtle's x-coordinate to +100* *move it 100 points to right of center* *no vertical change*
`setx -200`	*move turtle 200 points to left of center* *no vertical change*
`sety 150`	*set turtle's y-coordinate to 150* *move it 150 points above center* *no horizontal change*
`sety -50`	*move turtle 50 points below center* *no horizontal change*
`setxy 100 100`	*move turtle to xy coordinate 100 100*

`show xcor`
`show ycor`
	report on command line:
	where is the turtle now?

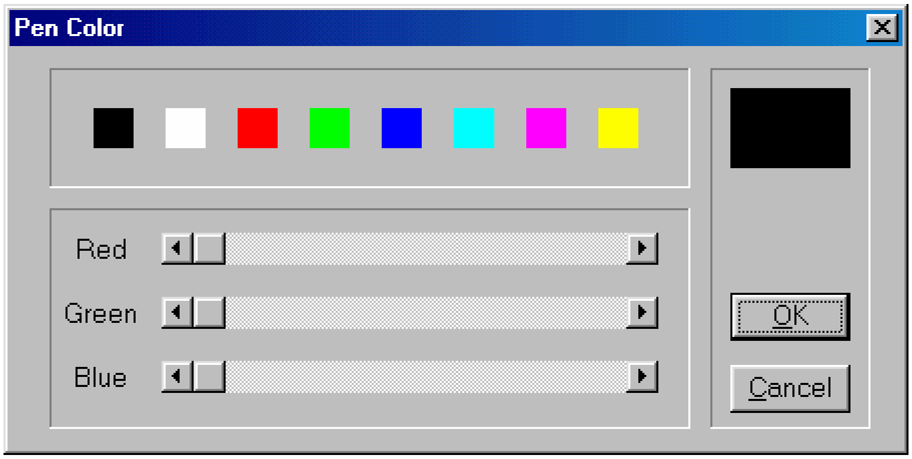
`cs`
`sety 100`
`setx 100`
`sety 0`
`setx 0`
	think about it —
	try it!

`setheading 0` *or* `seth 0`	*point turtle straight up, "high noon," "due north"*
`seth 120`	*four o'clock*

Color in Logo

Computer screens work with red, green, and blue components of light,
so they are sometimes called RGB screens.
(Remember Roy G. Biv?)

On Logo's **Set** menu, you can set the color of three screen elements —
the turtle's pen
the turtle's fill (like a paint bucket for enclosures)
the screen background



You set a color by moving these three sliders left and right.

Recall that black is the absence of all color and white is all colors together.
Mixing light isn't like mixing paint.
If you mix red and green paint, you get a muddy color —
what happens if you mix red and green light?

Since this is a computer, every color has an internal numeric representation.
The left end is of the sliding scale 0, zero.
The right end is 255, which is kind of like 99 to a computer. (It's $2^8 - 1$.)

Thus black is [0 0 0], red is [255 0 0], green is [0 255 0], blue is [0 0 255],
you can make anything in between that you like,
and in all there are 256 * 256 * 256 possible colors.
That's $2^8 * 2^8 * 2^8$, or 24 bits of color — 24 binary digits inside the machine.

These commands give you a big fat red pen:

```
setpensize [5 5]
setpencolor [255 0 0]
```

My colors	color values
red	
green	
blue	
black	
white	
yellow	

When you find a color you like using sliders, you can ask Logo what it is:
choose the pen color, then in the command window, enter

```
show pencolor
```

You can make a colored square like this:

draw the square
pen up
go to a point inside the square
fill

Color- and pen-related commands	

`setpencolor [r g b]`
`setpc [r g b]`
	color for turtle's pen
	r g b are numbers in range [0, 255]

`setfloodcolor [r g b]`
`setfc [r g b]`
	color for an enclosed area

`setscreencolor [r g b]`
`setsc [r g b]`
	color for background

`show pencolor`
`show floodcolor`
`show screencolor`
	tells you what values are right now for [r g b] of named item

`fill`	*dumps a bucket of current floodcolor at cursor's location*
`setpensize [w h]`	*sets width and height of pen* *w h are numbers in range [1, 5]*

A method for teaching color names to Logo

This method comes from the web site of Simone Rudge at Yukon College in Canada. She has published course notes for an entire semester's worth of serious Logo!
`http://www.yukoncollege.yk.ca/~srudge/comp052/notes.html`

Logo has a command `op` (short for `output`) that "produces" what you ask it to.
For example, if you want *blue* and the computer calls blue *`[0 0 255]`*,
you write a procedure for *blue* using `op` to produce the color vector
when you issue the command.

Do colors in this way in the Logo editor:

```
to blue
op [0 0 255]
end

to yellow
op [255 255 0]
end
```

then, when you want, for example, a yellow pen:

```
setpencolor yellow
```

When Logo sees the word *yellow,*
it looks in the list of procedures that it knows,
finds your procedure for yellow,
and sets the pen color accordingly.

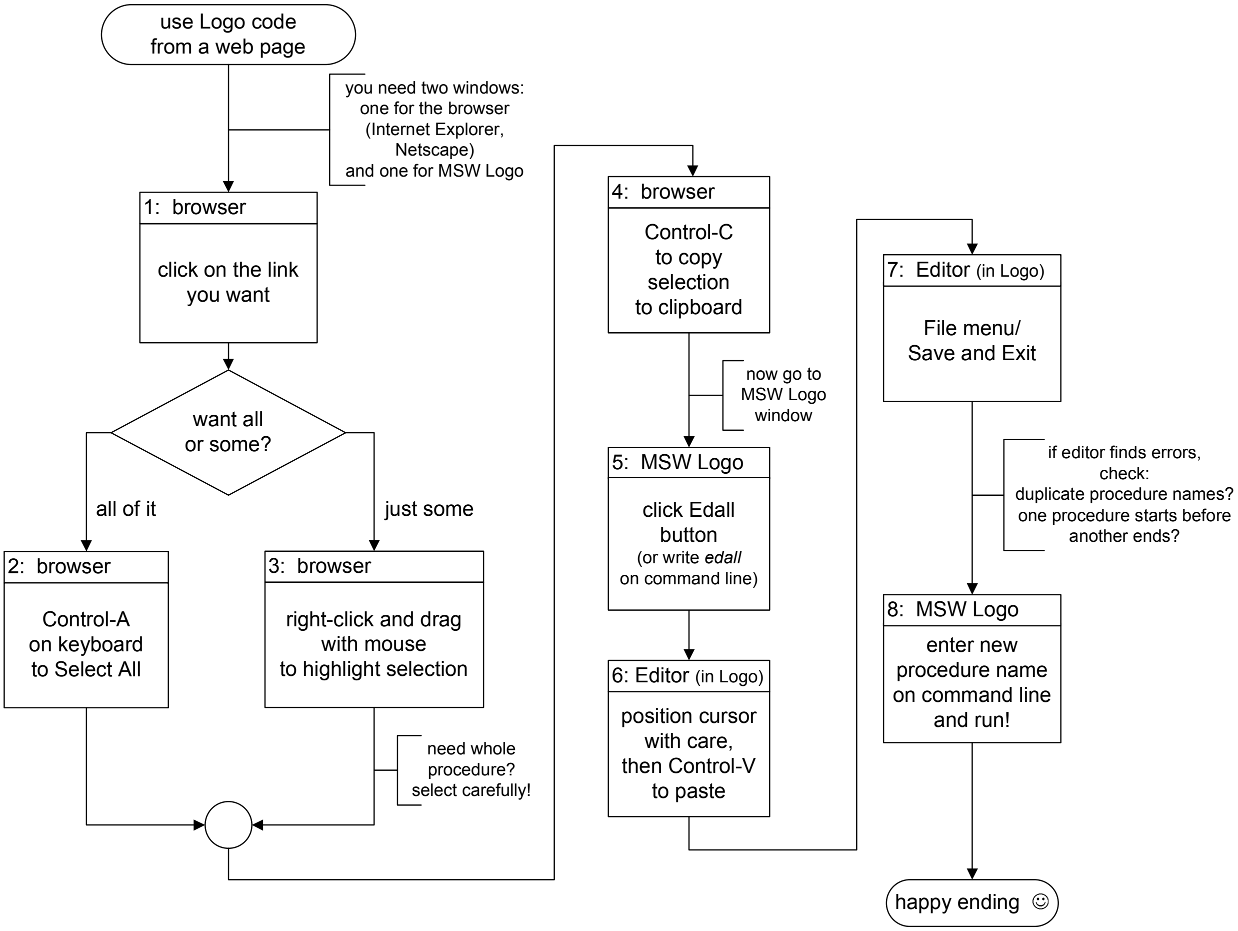
Make a blue square with yellow inside:

```
setpc blue
repeat4 [fd 100 rt 90]
rt 45
penup
fd 25
setfc yellow
fill
```

Using code from other people's programs

Many people, including me, put their code on the Internet and invite you to use it in Logo procedures of your own. This is very easily done using cut-and-paste techniques with the Microsoft Windows clipboard.

Here's one way to use code from somebody else's web page:


use Logo code from a web page
you need two windows: one for the browser (Internet Explorer, Netscape) and one for MSW Logo
1: browser
click on the link you want
want all or some?
all of it
just some
2: browser
Control-A on keyboard to Select All
3: browser
right-click and drag with mouse to highlight selection
need whole procedure? select carefully!
4: browser
Control-C to copy selection to clipboard
now go to MSW Logo window
5: MSW Logo
click Edall button (or write *edall* on command line)
6: Editor (in Logo)
position cursor with care, then Control-V to paste
7: Editor (in Logo)
File menu/ Save and Exit
if editor finds errors, check: duplicate procedure names? one procedure starts before another ends?
8: MSW Logo
enter new procedure name on command line and run!
happy ending ☺

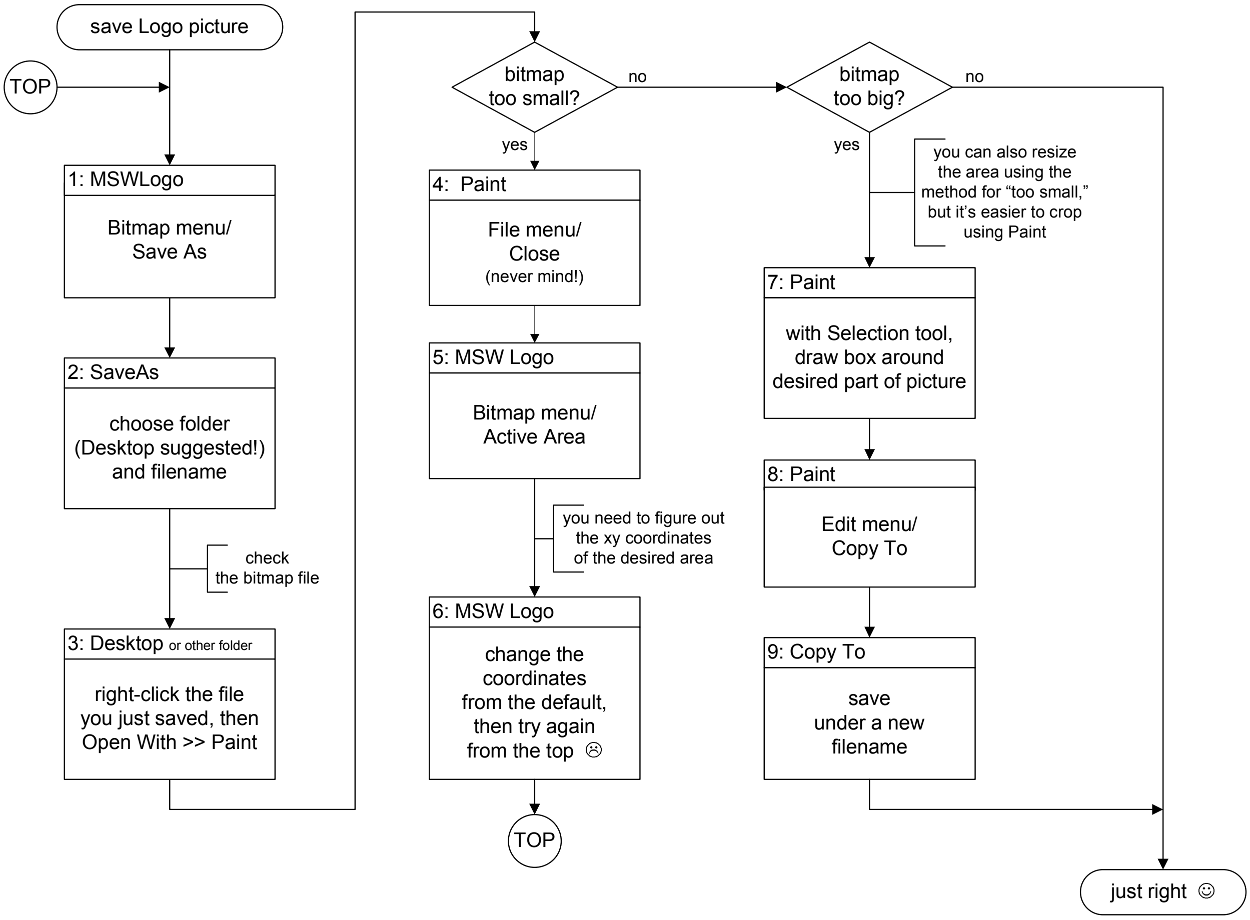
People who know Windows well can think of half a dozen ways to do the same thing, but I find this one reliable and straightforward.

If your Logo doesn't have an Edall button (box 5), write *edall* on the command line.

Saving your pictures and programs

Computer work is saved in computer files, with different kinds of work being saved in files with different names. On a Microsoft Windows system, files have screen icons and filename *extensions* that vary according to file content, and thus Logo pictures and Logo commands are saved in distinct kinds of files.

One type of file for saving a computer picture is called a bitmap. It's a grid of dots, like the computer screen itself, in which every dot's location and color are remembered. You can recognize a bitmap file by its icon (the little picture you double-click to open a file) or by its extension .BMP. Your computer might not be set up to show filename extensions, but you can change that with Folder View or Tools.


save Logo picture
TOP
1: MSWLogo
Bitmap menu/ Save As
2: SaveAs
choose folder (Desktop suggested!) and filename
check the bitmap file
3: Desktop or other folder
right-click the file you just saved, then Open With >> Paint
bitmap too small?
no
yes
4: Paint
File menu/ Close (never mind!)
5: MSW Logo
Bitmap menu/ Active Area
you need to figure out the xy coordinates of the desired area
6: MSW Logo
change the coordinates from the default, then try again from the top ☹
TOP
bitmap too big?
no
yes
you can also resize the area using the method for "too small," but it's easier to crop using Paint
7: Paint
with Selection tool, draw box around desired part of picture
8: Paint
Edit menu/ Copy To
9: Copy To
save under a new filename
just right ☺

Other bitmap formats are more compact; the bitmap files created by Logo are huge. You can conserve space and time, in case you send your drawings over the Internet, by converting your bitmap to another format. Two programs you could use for this are the Windows accessory program Paint and the wonderfully useful IrfanView. Copy and paste, then use the Save As menu of the other program to select a format.

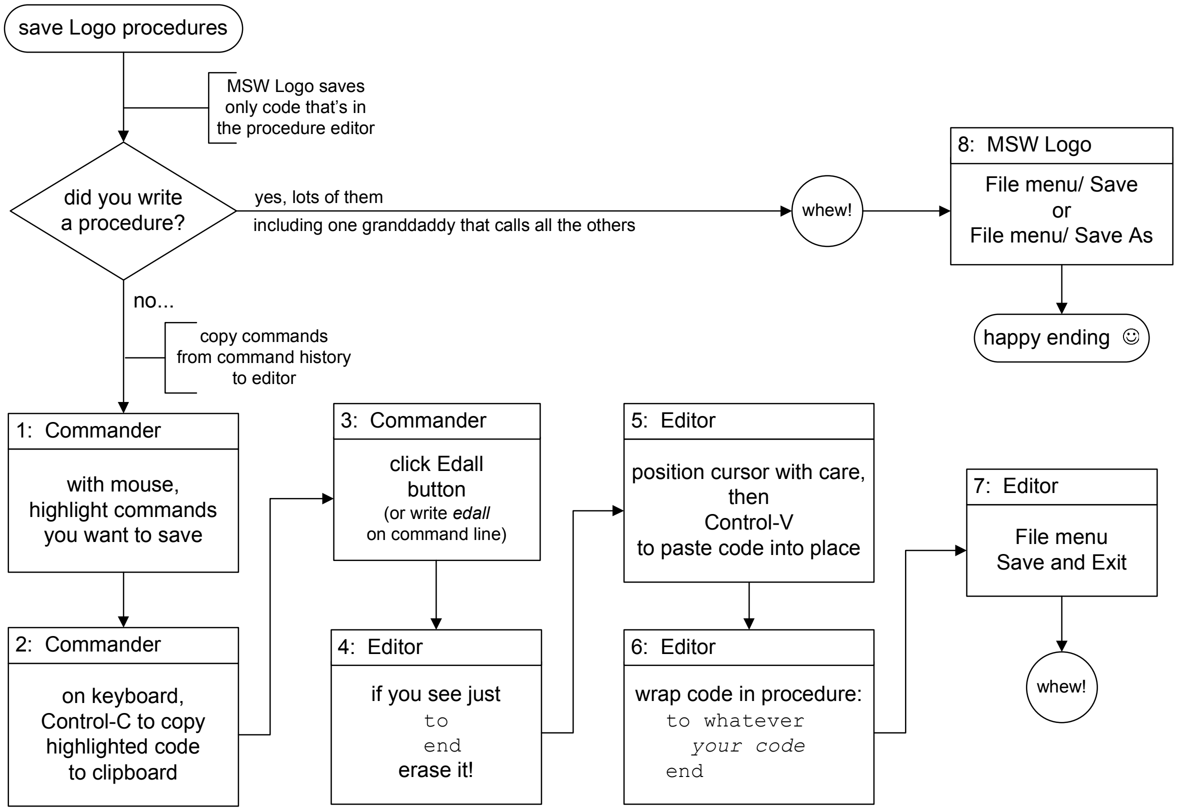
You will want to save the Logo procedures that you write, either to show off for other people or to continue refining them. (Probably both.) Like other programs, Logo saves nothing until you tell it to.

A feature of MSW Logo is that before you save, it remembers your work in two ways, as well as in two distinct locations:

> your command-line commands are remembered in the command history
>
> your procedures, i.e., the words you added to Logo, are remembered in the editor

When you do File menu/ Save, *only the procedures are saved.* If you want to remember the commands as well, then you must copy and paste them into the editor.

.LGO is the filename extension for files of procedures created by MSW Logo.


save Logo procedures
MSW Logo saves only code that's in the procedure editor
did you write a procedure?
yes, lots of them
including one granddaddy that calls all the others
whew!
8: MSW Logo
File menu/ Save or File menu/ Save As
happy ending ☺
no...
copy commands from command history to editor
1: Commander
with mouse, highlight commands you want to save
2: Commander
on keyboard, Control-C to copy highlighted code to clipboard
3: Commander
click Edall button (or write *edall* on command line)
4: Editor
if you see just to end erase it!
5: Editor
position cursor with care, then Control-V to paste code into place
6: Editor
wrap code in procedure: to whatever *your code* end
7: Editor
File menu Save and Exit
whew!

You can also use the Windows clipboard to copy and paste your procedures from the Logo command history or the Logo editor into an e-mail to yourself. Later, you will need to copy and paste them from the e-mail back into Logo's editor.

To recall your Logo code later

From the main MSW Logo screen, do File menu/ Load, then select the file you want. Logo will look for .LGO files by default. (Double-clicking on a file to open it doesn't seem to work in all versions of Windows.)

References

Here are a few of many great sources for more information about Logo.

Harvey, B. *Computer Science Logo Style*, second edition. Cambridge, Massachusetts: MIT Press, 1997.

In this amazing three-volume book, Brian Harvey teaches computer science — for adults, not children — using Logo. This work will be of particular interest to someone who wants to approach computer science using functional programming, as opposed to structural (e.g., Pascal) or object-oriented (e.g., Java). All 1000 pages can be downloaded from the author's web page at www.cs.berkeley.edu/~bh.

Harvey is one of the authors of Berkeley Logo, available on a variety of platforms, including Linux. (MSW Logo is based on Berkeley Logo.) He has also written a beginning computer science book using the functional language Scheme.

And here are some web sites, each with more links. (Use Google if the link is broken.)

MSW Logo, created by George Mills
www.softronix.com/logo.html
Fabulous freeware with links to many other sites.

A Turtle for the Teacher, by Paul Dench
www.ecu.edu.au/pa/ecawa/sig/logo/paul_dench/turtle/
Detailed and lengthy, at an elementary level.

Simone Rudge's college-level course outline
www.yukoncollege.yk.ca/~srudge/comp052/notes.html
Likewise, at a college level.

Logo Art Gallery by Yehuda Katz
www.geocities.com/CollegePark/Lab/2276/
Lots of arty and recursive examples.

Logo Foundation at MIT
el.media.mit.edu/logo-foundation/index.html
MIT's Artifical Intelligence Lab and Professor Seymour Papert have been instrumental in the development of Logo and the "learn by making" philosophy that Logo embodies. The Logo Foundation's *What Is Logo* page suggests the breadth and depth of Logo development around the world.