New & Improved Graph a Mystery Picture #1

Secret Picture 1 Coordinates	Procedures	Secret Picture 2 Challenge

[1 3]
[2 2]
[3 2]
[2 1]
[3 0]
[2 0]
[2 -4]
[1 -6]
[-1 -7]
[0 -8]
[2 -9]
[-1 -9]
[-1 -8]
[-2 -7]
[-3 -7]
[-4 -8]
[-2 -9]
[-5 -9]
[-5 -8]
[-4 -7]
[-7 -5]
[-8 -3]
[-8 0]
[-6 -1]
[-3 -1]
[-2 0]
[-2 2]
[-1 3]
[1 3]
[1 4]
[0 3]
[-1 4]
[-1 3]
 	
These coordinate points will be used in a procedure to get the turtle to plot them for us.
	`to mypicture`
	`pu`
	`setpos [1 3]`
	`pd`
	`setpos [1 3] dot`
	`setpos [2 2] dot`
	`setpos [3 2] dot`
	`setpos [2 1] dot`
	`setpos [3 0] dot`
	`setpos [2 0] dot`
	`setpos [2 -4] dot`
	`setpos [1 -6] dot`
	`setpos [-1 -7] dot`
	`setpos [0 -8] dot`
	`setpos [2 -9] dot`
	`setpos [-1 -9] dot`
	`setpos [-1 -8] dot`
	`setpos [-2 -7] dot`
	`setpos [-3 -7] dot`
	`setpos [-4 -8] dot`
	`setpos [-2 -9] dot`
	`setpos [-5 -9] dot`
	`setpos [-5 -8] dot`
	`setpos [-4 -7] dot`
	`setpos [-7 -5] dot`
	`setpos [-8 -3] dot`
	`setpos [-8 0] dot`
	`setpos [-6 -1] dot`
	`setpos [-3 -1] dot`
	`setpos [-2 0] dot`
	`setpos [-2 2] dot`
	`setpos [-1 3] dot`
	`setpos [1 3] dot`
	`setpos [1 4] dot`
	`setpos [0 3] dot`
	`setpos [-1 4] dot`
	`setpos [-1 3] dot`
	`end`
	
	to dot
	setpensize 3
	pd fd 0
	setpensize 1
	end
		Can you write a new procedure to connect these points?
		
		Be sure that your procedure has a new name!
		
		[-6 -9]
		[-6 -8]
		[-7 -7]
		[-8 -5]
		[-8 -2]
		[-6 0]
		[-7 3]
		[-5 4]
		[-4 3]
		[-5 3]
		[-5 2]
		[-4 0]
		[-3 3]
		[-1 4]
		[0 3]
		[-1 3]
		[-2 2]
		[-2 0]
		[-1 0]
		[0 -1]
		[0 -2]
		[3 -2]
		[3 -5]
		[-2 -8]
		[-2 -9]
		[-6 -9]
		
		Here's a super version of dot for even cooler pictures! What does it do differently?
		
		`to dot`
		`setpensize 3 setc “red`
		`pd fd 0`
		`setpensize 1 setc “black`
		`end`

Mathematician's Mind-boggling Challenge

How can you make the turtle draw a larger version of your connect-the-dots graph?

Getting the Computer to Work for You

Lots of what kids experience while programming is a form of working for the computer. You formalize thinking and communicate precise directions to the computer. Once you develop a bit of programming fluency, the computer can work for you.

In this context, the most obvious shift is in reducing the amount of typing a user must do to get a computer to graph a nonspecific number of coordinate points.

The First Simplification

The `mypicture` procedure has an obvious and immediate need to simplification. Anytime you see a repetitive pattern or set of similar instructions used multiple times, it may be a good time to consider writing a helper procedure to do more of the work. `Setpos [ x y ] dot` is a great example. Let's create a go procedure that takes a position (expressed as a list of two numbers representing *x* and *y*) as input.

```
to go :point
setpos :point
dot
end
```

Now, your `mypicture` procedure can look like this.

```
to mypicture
pu
setpos [1 3]
pd
go [1 3]
go [2 2]
go [3 2]
.
.
.
end
```

That's Still Too Much Work!

Using list processing, we can eat through a list of coordinate points and graph them in sequence, without even knowing how many points there are. The following *recursive* procedure does this.

```
to grapher :list
if empty? :list [stop]
go first :list
grapher bf :list
end
```

`Grapher` takes a list as input, goes to the first set of coordinates in the list, and then runs the same procedure again, but this time passes everything but the first item in the list (a pair of coordinates) back to the `grapher` procedure. The points are a list of lists.

The `if empty? :list [stop]` line is called a *stop rule*. The stop rule tells the computer to stop the procedure as soon as the list of inputs is empty. A programming line such as this is used in countless contexts. Test it out in the command center.

```
Grapher [[0 0] [0 50] [50 50] [50 0]]
```
or
```
Grapher [[0 0] [0 50] [50 50] [50 0][35 35]]
```

The stop rule allows you to use as many or few elements in the list input to the procedure.

My Picture is Too Small!

Now that we've written a procedure to eat through a list of any size and graph the coordinates found in the list, we need a way to change the scale of the drawing. The first obvious improvement to the `grapher` procedure is to add an input for scale.

```
to supergrapher :list :scale
if empty? :list [stop]
.
.
end
```

So far, so good. Our procedure has two inputs and a stop rule, but where's the beef?

We need to take apart our list of coordinates, multiply each element by the scale, and then put the coordinates back together so the turtle may be sent to that new place on the screen.

```
to supergrapher :list :scale
if empty? :list [stop]
setpos list (first first :list) * :scale (last first :list) * :scale
dot
supergrapher bf :list :scale
end
```

If you don't understand what's happening in the setpos line of this procedure, try the following in the command center.

```
show first first [[10 20] [30 40] [50 60]]
show (first first [[10 20] [30 40] [50 60]]) *5
show list (first first [[10 20] [30 40] [50 60]]) * 5 (last first
[[10 20] [30 40] [50 60]]) * 5
show (list (first first [[10 20] [30 40] [50 60]]) * 5 (last first
[[10 20] [30 40] [50 60]]) * 5)
```

What happened?

`list`'s job is to put two items together and report them as a list, like `sentence`, if you have more than two elements, put the entire expression in parentheses (   ) in order to create a list of many items.

Now graph like crazy with the new `supergrapher`!

```
Supergrapher [[-6 -9] [-6 -8] [-7 -7] [-8 -5] [-8 -2] [-6 0] [-7 3]
[-5 4] [-4 3] [-5 3] [-5 2] [-4 0] [-3 3] [-1 4] [0 3] [-1 3] [-2 2]
[-2 0] [-1 0] [0 -1] [0 -2] [3 -2] [3 -5] [-2 -8] [-2 -9] [-6 -9]] 5
```

```
Supergrapher [[-6 -9] [-6 -8] [-7 -7] [-8 -5] [-8 -2] [-6 0] [-7 3]
[-5 4] [-4 3] [-5 3] [-5 2] [-4 0] [-3 3] [-1 4] [0 3] [-1 3] [-2 2]
[-2 0] [-1 0] [0 -1] [0 -2] [3 -2] [3 -5] [-2 -8] [-2 -9] [-6 -9]] -
10
```

If you just want to play with the newly discovered chicken, you could write a procedure like this one.

```
to superchicken :scale
supergrapher [[ 1 3 ] [ 2 2 ] [ 3 2 ] [ 2 1 ] [ 3 0 ] [ 2 0 ] [ 2 -4
] [ 1 -6 ] [ -1 -7 ] [ 0 -8 ] [ 2 -9 ] [ -1 -9 ] [ -1 -8 ] [ -2 -7 ]
[ -3 -7 ] [ -4 -8 ] [ -2 -9 ] [ -5 -9 ] [ -5 -8 ] [ -4 -7 ] [ -7 -5
] [ -8 -3 ] [ -8 0 ] [ -6 -1 ] [ -3 -1 ] [ -2 0 ] [ -2 2 ] [ -1 3 ]
[ 1 3 ] [ 1 4 ] [ 0 3 ] [ -1 4 ] [ -1 3 ]] :scale
end
```

Try

```
cg superchicken 5
cg superchicken 2
cg superchicken 20
```

You may find that specifying the scale after a long list aesthetically displeasing. That's an easypeasy fix. Just reverse the inputs in the procedure and recursive call.

```
to supergrapher :scale :list
if empty? :list [stop]
setpos list (first first :list) * :scale (last first :list) * :scale
dot
supergrapher :scale bf :list
end
```

Now try this new and improved version!

```
Supergrapher -10 [[-6 -9] [-6 -8] [-7 -7] [-8 -5] [-8 -2] [-6 0] [-7
3] [-5 4] [-4 3] [-5 3] [-5 2] [-4 0] [-3 3] [-1 4] [0 3] [-1 3] [-2
2] [-2 0] [-1 0] [0 -1] [0 -2] [3 -2] [3 -5] [-2 -8] [-2 -9] [-6 -
9]]
```

If you want to simplify this process for other users, you can create a visual slider called something like, *magnification*, on the screen (if using Lynx) and then create a button set to run a procedure, such as the following.

```
to sc2
supergrapher coordinates magnification
end
```

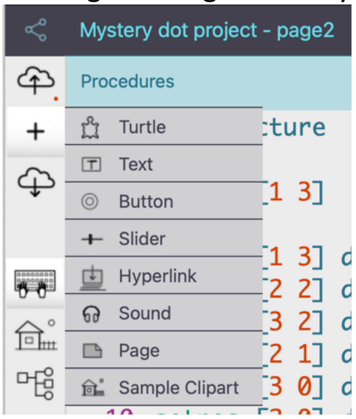
The coordinates procedure is a reporter that just outputs the list of coordinates you're using as an input to other procedures, over and over again.

In this case, the coordinates are the points used to create the original tiny chicken.

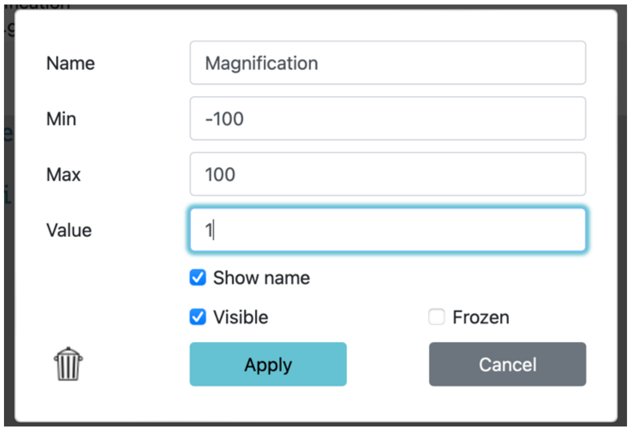
```
to coordinates output [[1 3] [2 2] [3 2] [2 1] [3 0] [2 0] [2 -4]
[1 -6] [-1 -7] [0 -8] [2 -9] [-1 -9] [-1 -8] [-2 -7] [-3 -7] [-4 -8]
[-2 -9] [-5 -9] [-5 -8] [-4 -7] [-7 -5] [-8 -3] [-8 0] [-6 -1]
[-3 -1] [-2 0] [-2 2] [-1 3] [1 3] [1 4] [0 3] [-1 4] [-1 3]]
end
```

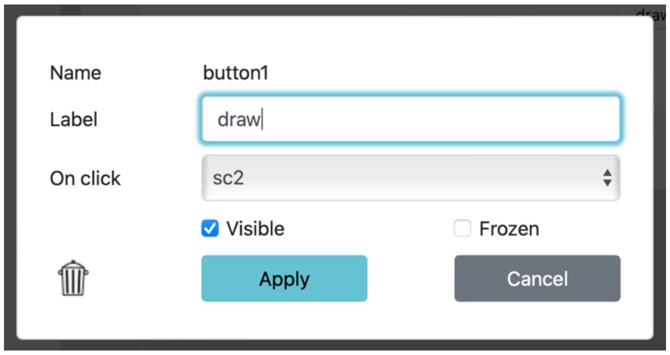
`Magnification` is the name of a slider created by:

- Clicking the + sign in the Lynx tool palette



- Name the slider, set its range, and starting value.


Name: Magnification
Min: -100
Max: 100
Value: 1
☑ Show name
☑ Visible ☐ Frozen


Name: button1
Label: draw
On click: sc2
☑ Visible ☐ Frozen

- Next, create a new button from the same tool palette.
- Label the button `draw`
- Choose the `sc2` procedure to run when the button is clicked

Change the value of the slider by clicking and dragging and then click on the draw button to see what happens. Try this several times.

A version of this nonsense is online in Lynx at https://lynxcoding.club/share/NfV6z86P.

List Processing is Universal!
The sort of code found in `grapher` or `supergrapher` are used in countless contexts! Understand it and you can get the computer to work for you in innumerable ways. Here's an example of manipulating computer music.

Let's imagine that Lynx had a primitive command called, `note`. `Note` takes two inputs, pitch & duration. (Lynx should have simple music primitives and I will try to convince the developers to add them. Sadly, the next example will require you to run the program in your head.)

Let's say that `Note [1 4]` plays a C for a count of 4
`Note [3 4]` plays a D for 4 counts
`Note [5 4]` plays an E for 4 counts

If we had a play procedure that could take a list of notes and durations, we could manipulate the music just like a composer!

`Play` [[1 4] [3 4] [5 4]

```
to play :music
If empty? :music [stop]
note list first first :music last first :music
play bf :music
end
```

Another version of this procedure could speed up or slow down the music, just like we did in `supergrapher`.

```
to play :music :tempo
If empty? :music [stop]
note list (first first :music) (:tempo * (last first :music))
play bf :music :tempo
end
```

`Play` [[1 4] [3 4] [5 4]] 1
Plays at normal tempo
`Play` [[1 4] [3 4] [5 4]] 2
Plays the same notes twice as slow
`Play` [[1 4] [3 4] [5 4]] .5
Plays the same notes twice as fast

Can you add an input to play for *`transposition`*? Its job is to change the pitch by a numerical value (+ or -).

Can you write a procedure that will play a list of notes and durations backwards, or in musical parlance, retrograde?

If you get tired of typing the list of values over and over again, put the computer to work. Create a procedure like the following.

```
to music
output [[1 4] [3 4] [5 4]]
end
```

Then run the following instruction:

```
play music .5
```

or

```
to points
output [[-6 -9] [-6 -8] [-7 -7] [-8 -5] [-8 -2] [-6 0] [-7 3] [-5 4]
[-4 3] [-5 3] [-5 2] [-4 0] [-3 3] [-1 4] [0 3] [-1 3] [-2 2] [-2 0]
[-1 0] [0 -1] [0 -2] [3 -2] [3 -5] [-2 -8] [-2 -9] [-6 -9]]
end
```

Then run the following instruction:

`supergrapher 10 points` or `supergrapher points 10,` depending on which order you decided to use the inputs in your procedure.

The Big Idea

Versions of the grapher/supergrapher procedure are used in music composition, encryption, codes & cryptography, art, and data manipulation of all kinds. If you understand these fundamental list processing techniques of "eating" and "smushing," you can solve a world of problems and put the turtle to work for you.