An Introduction to Scheme Programming Through SchemePaint

For the first 2-3 weeks of this course, we will focus on introducing techniques of functional programming. The language that we will use is called Scheme; this is a dialect of Lisp, the most popular language for artificial intelligence programming in the U.S.

The way that Scheme programming will be introduced in this class is through a graphics application named "SchemePaint." You can think of SchemePaint as a sort of "souped-up paint program": it allows you to draw shapes on the screen by hand (using the mouse), but also to combine those pictures with figures produced by programs that you can write. Using Scheme to write graphics programs is an especially good (and, we hope, enjoyable) way to introduce many of the fundamental concepts involved in programming.

Over time we will focus progressively less on graphics and more on the "classic" areas of Lisp programming—namely, artificial intelligence and symbolic manipulation. By the time we move on to these non-graphics projects, though, you will have been introduced to many of the skills involved in writing sophisticated Lisp programs.

A Note About the Software

We have been given permission by Lightship Software to distribute copies of SchemePaint to students in this course.

PLEASE DO NOT DISTRIBUTE OR COPY THIS SOFTWARE; AND IF YOU USE IT ON A PUBLIC MACHINE, PLEASE DELETE THIS SOFTWARE FROM THE MACHINE WHEN YOU ARE DONE.

In this regard, we ask that you sign and hand in the agreement attached to this handout.

You may find it easiest to use this software by finding a Macintosh—either at home, or in a computer lab—upon which to run it. Over the next several weeks, we will set up MacScheme stations at a couple of publicly available Macs, and if you prefer, you can use the software at those stations. SchemePaint runs on any color Macintosh with at least 16M of RAM.

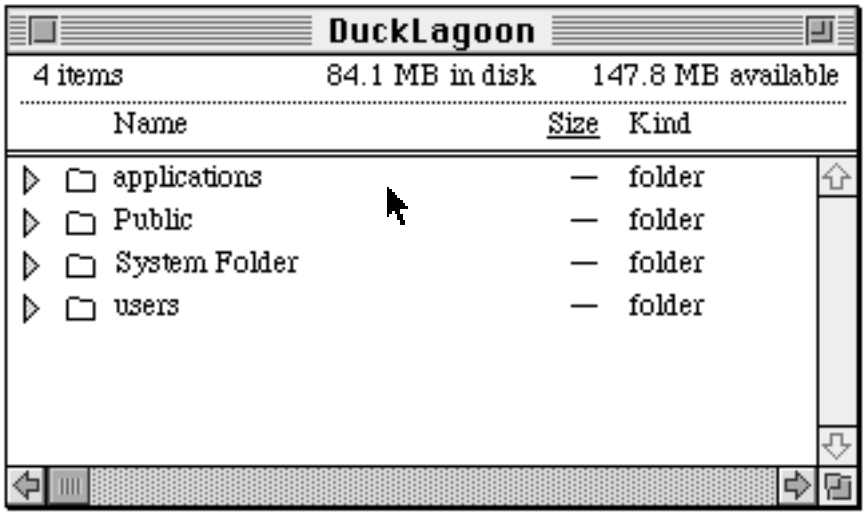
1.1 Starting the SchemePaint system

1.1.1 Copying the SchemePaint system over to a Macintosh

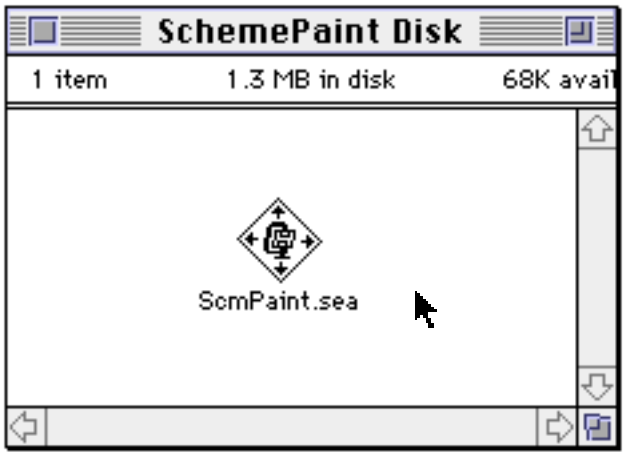
Note: The description below assumes a slight (but nonzero) acquaintance with the Macintosh interface. If you have any questions about how to follow these instructions, you should talk to Mike.

In order to work with SchemePaint, you will have to copy the files on your floppy disk to a Macintosh hard disk. Here's how to go about this:

a. Double-click the mouse button on the icon for the hard disk on the machine you are using. This will open the directory for the hard disk.



b. Insert the disk labelled "SchemePaint Disk" into the Macintosh floppy disk drive. Open the directory for the floppy disk (again by double-clicking).



c. Double-click on the file "ScmPaint.sea". (The suffix ".sea" stands for "self-extracting archive.") You will see a dialog box asking you to create a new folder named "ScmPaint folder" on the hard disk. Select the choice "Save" and a new folder will be created on the hard disk. This new folder will contain two files: one labelled "ScmPaint1.heap" and the other labelled "Toolsmith 4.2 beta".

d. If this is a home Macintosh that you will be able to use on a regular basis, then you needn't erase the SchemePaint folder when you are through working. On the other hand, if this is a public (e.g., library) Mac and is not one of the designated MacScheme workstations, then you should be sure to delete the folder once you are done. The way to do this is to drag the SchemePaint folder (and *only* this folder) onto the trash-can icon, and then select "Empty Trash" from the Special menu.

1.1.2 Starting SchemePaint

a. You can start the SchemePaint system either by double-clicking on the CSSchemePaint.heap icon, or by single-clicking on that icon and dragging it over the MacScheme+TS icon.

b. A large window labelled transcript will come up on your screen. This window will include some startup information which you should read. The system that you have is capable of loading two possible systems—SchemePaint, and a related system named HyperGami. In this course we will only use SchemePaint, so you can ignore that part of the start-up information that is specific to HyperGami.

At the end of the start-up information, the system will pose a question about screen size:

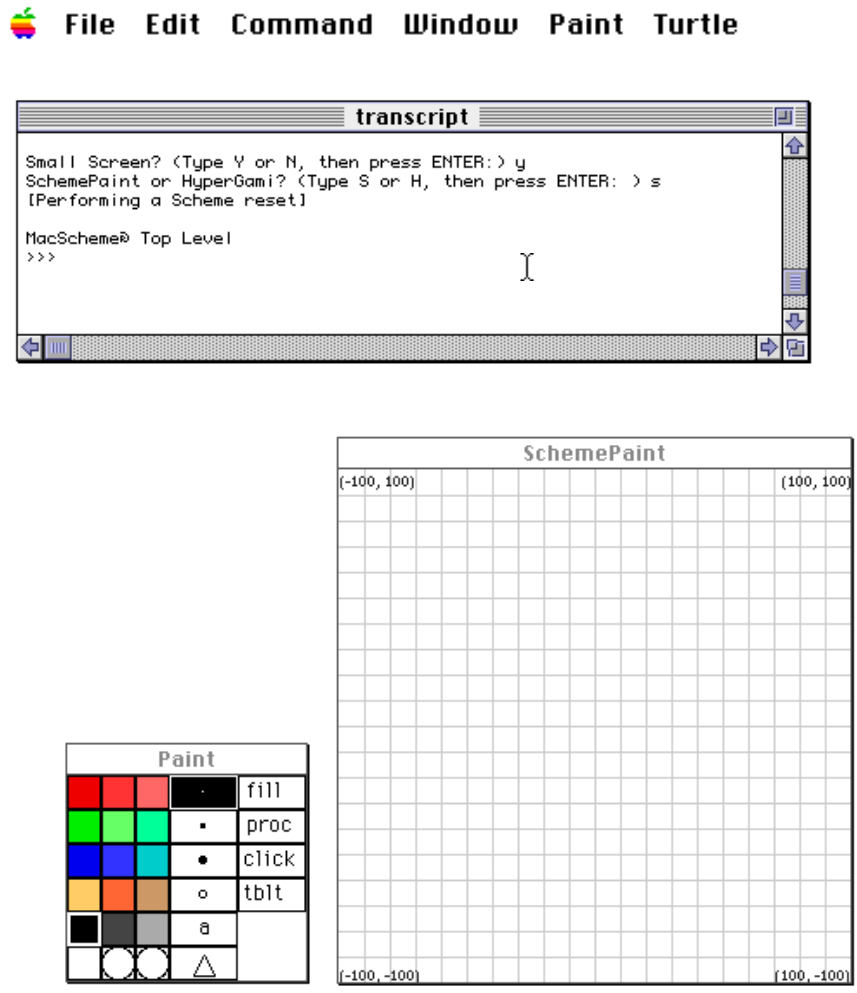
```
Small Screen? (Type Y or N, then press ENTER:)
```

If you are working with a 21-inch screen (lucky!) then type Y, otherwise N. The system will then pose a second question:

```
SchemePaint or HyperGami? (Type S or H, then press ENTER: )
```

In this case, you should type S (in fact, you don't have the complete set of files to run HyperGami, anyhow).

Once you press S, you will see a couple of new windows pop up on the screen (and a couple of new menu titles as well, at the top of the screen). Resize the **transcript** window, and move the SchemePaint and Paint windows so that your screen looks like the figure below:

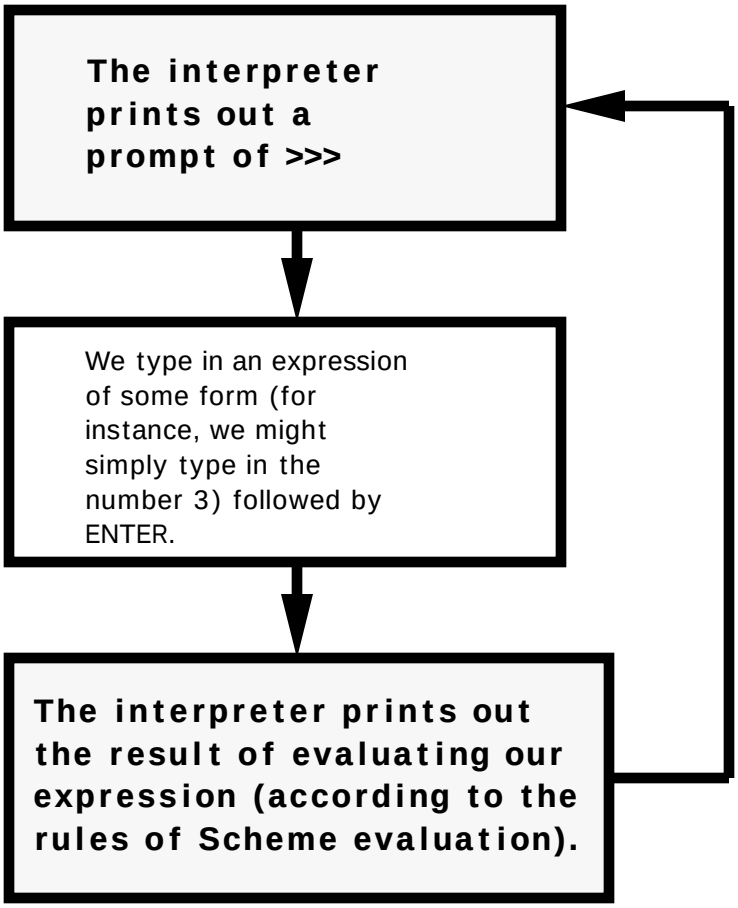


Now you are ready to start working with SchemePaint.

1.2 The Scheme Interpreter

In general, when working with SchemePaint you should begin by setting up the screen so that it looks like the previous figure. For the moment, however, we will forget about the **SchemePaint** and **Paint** windows, and focus on the window labelled **transcript** at the top of the screen. This is the window in which we interact with the Scheme interpreter.

You can think of the Scheme interpreter as an abstract machine that communicates with us via the transcript window. The transcript presents us with a "prompt" (in the case of SchemePaint, the prompt is ">>>"); we type in expressions in response to the interpreter prompt (followed by the ENTER key); and the interpreter replies by finding the value of those expressions. This may sound rather obscure at the moment, but as you work with the interpreter you will get a more textured idea of how it operates.



1.3 Evaluating Numeric Expressions

Numbers evaluate to themselves. (I know this sounds either mysterious or obvious just now, but for the moment bear with me.) To evaluate a numeric expression, simply type the number at the Scheme prompt and then type the **ENTER** (not the **RETURN**) key.

```
>>> 3
3

>>> 4.5
4.5

>>> 1.2e3
1200.0

>>> -4e-3
-0.004
```

The next step, after looking at numbers, is to evaluate a few *compound expressions* (for now, we can just think of these as expressions surrounded by parentheses). Here are a few compound arithmetic expressions:

```
>>> (+ 4 5)
9

>>> (- 16.5 2.2)
14.3

>>> (* 4.2 5)
21.0

>>> (/ 8 3)
2.6666666666666665
```

Note that Scheme uses *prefix notation*: the arithmetic operator appears first, followed by the numeric *arguments* to that operator.

1.4 A Brief Detour: Dealing with Errors

Suppose that instead of typing in a number, we accidentally happen to type in a letter instead:

```
>>> g
Undefined global variable:
g

Entering debugger. Enter ? for help
debug:>
```

What happened? We will go into more detail about this later; but briefly, the Scheme interpreter is telling us that it cannot evaluate the expression "g". A slightly more technical way of putting this is that we have typed in a symbolic expression (here, the symbol g); and the Scheme interpreter does not recognize this particular symbol. In response the Scheme interpreter has given us a prompt for an interactive debugger -- a system used for debugging running programs.

Over time we will see more of the debugger, and learn a bit about how to use it. For now, though, our main concern is in exiting the debugger and going back to the Scheme interpreter. We do this by typing the letter q (for "quit") and then ENTER:

```
debug:> q
[Performing a Scheme reset]

MacScheme Top Level
>>>
```

Now we are once more dealing with the Scheme interpreter.

1.5 Nested Expressions

As you might expect, we can use compound arithmetic expressions as parts of still-larger expressions:

```
>>> (+ 3 (* 6 8))
51

>>> (* 4.2 (+ 21 2))
96.6

>>> (+ (* 2 3) (- 8 6))
8

>>> (* (+ 2.2 3) (- 5 (/ 8 4)))
15.6
```

It is worth pausing at this point to describe in a bit more detail how the Scheme interpreter is dealing with expressions of this type. In essence (though admittedly we're being a little hand-wavy about the details) the interpreter first determines that we have typed in a compound expression beginning with a *primitive procedure* — that is, an expression beginning with a procedure that has been "built-in" to the interpreter. The four arithmetic procedures +, -, *, and / are all primitive procedures.

The interpreter next goes on to evaluate each of the argument subexpressions; the results of these evaluation steps are then passed as arguments to the original primitive procedure.

Here's an example—suppose we type in the following expression:

```
>>> (+ 2 3)
```

The Scheme interpreter first determines that this is a compound expression beginning with the primitive addition procedure. The interpreter then goes on to evaluate each of the two argument subexpressions (here, the expressions 2 and 3). Each of these argument subexpressions is just a number which (as we noted in Section 1.3 earlier) evaluates to itself; so the interpreter now uses the numbers 2 and 3 as arguments to the addition procedure and returns as the final result the number 5.

Now, suppose we type in the following expression:

```
>>> (- 9 (+ 2 3))
```

Again, the Scheme interpreter determines that this is a compound expression beginning with the primitive subtraction procedure. The interpreter goes on to evaluate each of the two argument subexpressions. The first of these is a number (evaluating to 9); the second is itself a compound expression evaluating to the number 5 (as we discovered a moment ago). The interpreter

now uses the numbers 9 and 5 as the arguments to the primitive subtraction procedure, and the result of the overall evaluation process is thus the number 4.

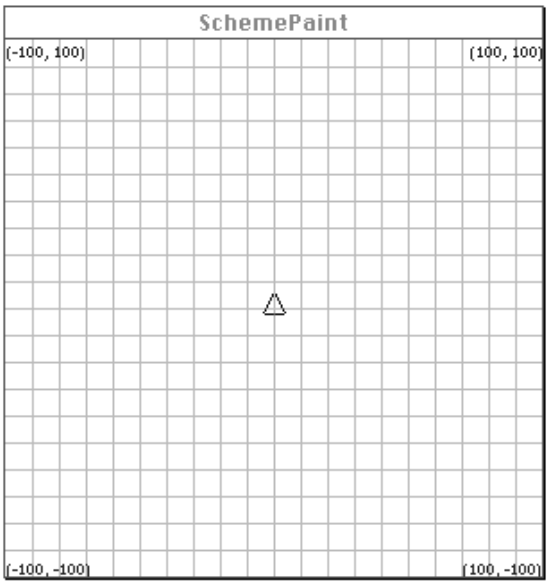
1.6 Using the SchemePaint Interface

We will have much more to say about the Scheme interpreter and its evaluation rules over the next several lectures; but for the time being we will postpone any further discussion of these matters and instead explore some of the basic functionality of the SchemePaint interface.

First, it is worth playing with a few of the menu choices provided in SchemePaint. The **Paint** menu provides (among others) choices allowing us to clear the **SchemePaint** window (the "Clear Windows" command) and to toggle the size of the **Paint** window ("Toggle Palette Size"). The second is useful when you wish to change the number of available default colors (the choice switches back and forth between a larger and smaller number); the first is useful when you wish to refresh the **SchemePaint** window before beginning some new drawing project.

In the following sectionwe will begin to look at some turtle-graphics expressions: the "turtle," as we will see, is a little programmable cursor that draws lines on the screen. Before writing turtle-graphics expressions, though, we need to use a few handy commands from the **Turtle** menu. The "Show or Hide Turtle" command can be used to show (or, if shown, to hide) the turtle on the screen (it appears as a little isosceles triangle pointed upward). The "Pen Down" command is used to tell the turtle to draw lines when it moves (conversely, the "Pen Up" command will tell the turtle to move without drawing any lines on the screen). Finally, the "Center" command can be used to move the turtle to the center of the **SchemePaint** window.

After selecting "Show or Hide Turtle" to show the turtle, the **SchemePaint** window will look as in the figure below.


SchemePaint
(-100, 100)
(100, 100)
(-100, -100)
(100, -100)

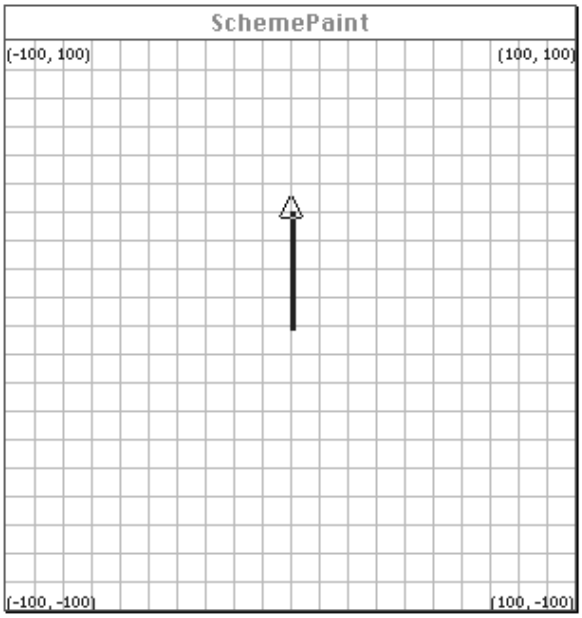
1.7 Turtle Expressions

1.7.1 Moving the turtle forward

We can now begin working with a few initial turtle-graphics expressions in Scheme. If we type in the following expression to the Scheme interpreter:

```
(forward 40)
```

we will see the turtle move forward (in this case, due north) forty units on the screen:



The forward procedure is a primitive procedure—in a sense, very similar to the arithmetic procedures that we saw earlier: it takes a single argument (which should be a number) and uses that argument to tell the turtle how many steps to move in the forward direction. Note that the forward procedure can take a floating point argument; it can take a negative number as argument; and (this should be no surprise) it can take an argument whose value is produced by a compound expression:

```
(forward 33.3)
(forward -25)
(forward (* 5 6))
```

1.7.2 Turning the turtle

If the turtle could only move in one direction, it wouldn't be too interesting. The next step beyond moving the turtle forward is to change its direction. One way to do this is to tell the turtle to turn to its right by a certain number of degrees:

```
(right 90)
```

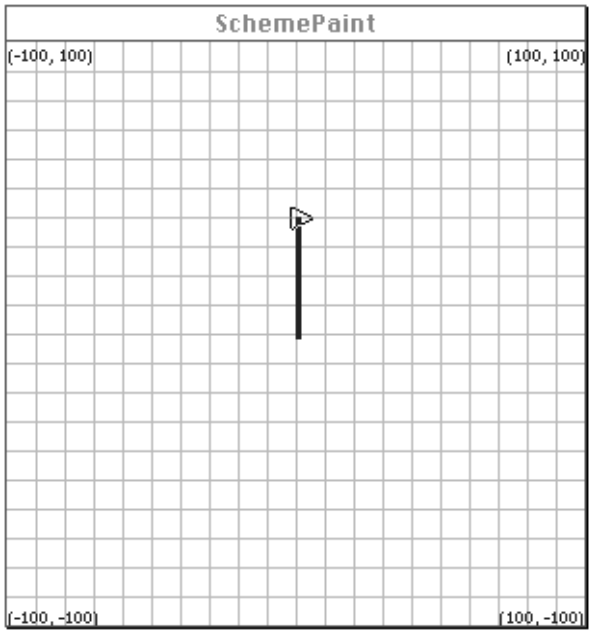
If we type this expression at the Scheme interpreter, the turtle turns to the right by 90 degrees (i.e., it does a "right face" turn). If we were to follow our previous

```
(forward 40)
```

expression with a

```
(right 90)
```

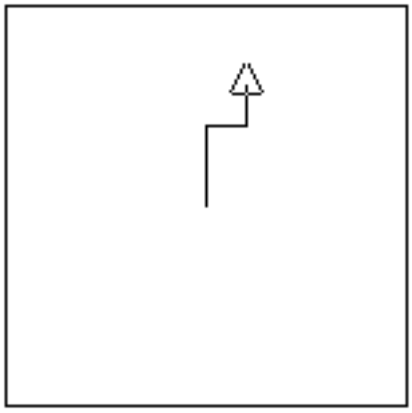
expression, then the screen would look as follows:



We can evaluate a series of Scheme turtle expression in sequence to produce particular shapes. For instance, suppose we first clear the window, center the turtle, and then evaluate the following five expressions:

```
(forward 20)
(right 90)
(forward 10)
(right -90)
(forward 10)
```

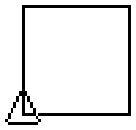
We will see the following path taken by the turtle (for the remainder of these examples we only show the turtle path without the background grid or surrounding **SchemePaint** window):



Or—to take another example—suppose we center the turtle (and clear the window) and evaluate the following eight expressions:

```
(forward 25)
(right 90)
(forward 25)
(right 90)
(forward 25)
(right 90)
(forward 25)
(right 90)
```

The path produced by the turtle will in this case be a square:



1.7.3 Using REPEAT to perform sequences many times

One more addition to our repertoire will now be particularly useful. You may have noticed in the previous example that we in effect repeated four times a sequence of two turtle expressions: that is, we typed in the two expressions:

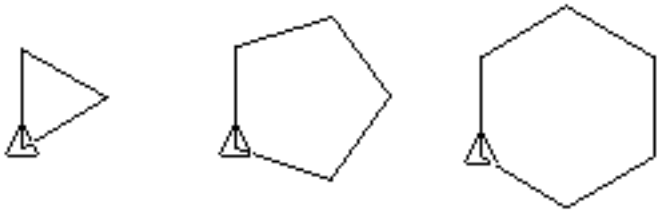
```
(forward 25)
(right 90)
```

four successive times. Rather than go to this trouble, however, we can in fact use a Scheme form named "repeat" to evaluate a sequence of turtle expressions over and over, for some specified number of times. Here is the equivalent repeat formulation of our previous square:

```
(repeat 4 (forward 25) (right 90))
```

Typing in this expression at the Scheme interpreter would produce a square exactly like the one that we drew earlier. In general, the repeat form is followed by (first) a number—indicating the number of times that we wish to repeat some sequence—and then the sequence of one or more expressions that we wish to repeat.

From these humble beginnings, it isn't too hard to start experimenting with other regular polygons. Shown below are three examples—a triangle, pentagon, and hexagon—and the Scheme expressions that generate them:



```
(repeat 3 (forward 25) (right 120))

(repeat 5 (forward 25) (right 72))

(repeat 6 (forward 25) (right 60))
```

Having come this far, we can go one final step further—suppose we repeat expressions as parts of still larger repeat expressions? In other words, we might try repeating, six times, a hexagon-drawing expression followed by a (right 60) expression:

```
(repeat 6 (repeat 6 (forward 25) (right 60)) (right 60))
```

The result is shown below: