Laboratory 8

Using Algorithms for Painting

Objectives

- Learn how to use Palgo, an applet that *P*aints *ALGO*rithmically.
- Learn how to write programs for Palgo.

References

Software needed:

1) A web browser (Internet Explorer or Netscape)
2) Applet from the CD-ROM:
   a) Palgo

Textbook reference: Chapter 8

Background

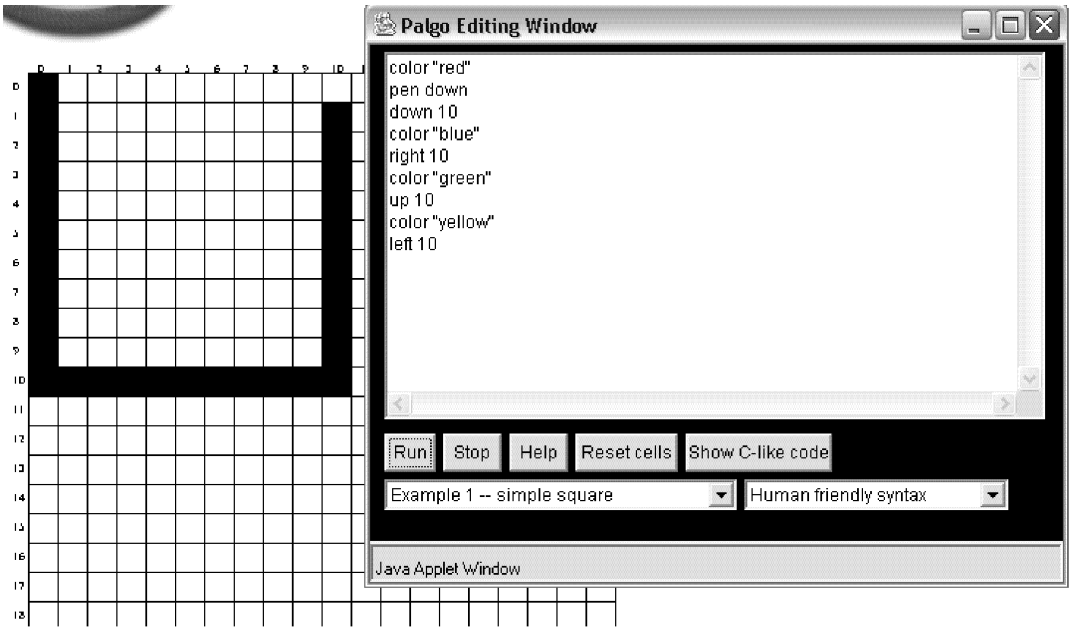
The general background of high-level programming languages is presented in Chapter 8. Palgo uses an imperative programming language to draw pictures. The language constructs are explained in this lab.

Activity

Sometimes it is hard to get a feel for algorithms when solving traditional problems, such as sorting a list or averaging employees' salaries. Pictures are "funner!" Palgo follows in a long tradition, starting with Seymour Papert's Turtle Graphics, of using a programming language to draw pictures. All the basic elements of algorithms can be used and studied: sequence, decision, repetition, and subalgorithm.

To begin, start the "Palgo" applet. The edit window will appear, with the painting window behind it. Select *Example 1* (*Simple Square*) from the pull-down menu, and press the *Run* button. In the painting window you'll see a square, with four different-colored sides (see the screenshot below).

This first program is quite simple, consisting only of a sequence of painting commands. The `color` command changes the color of the invisible pen that is drawing on the square. The `pen down` and the `pen up` commands tell Palgo to start or stop coloring squares with the current color, respectively. (You would need to stop painting, for example, if you wanted the pen to jump to a non-adjacent square without painting the intervening squares.)



The invisible pen starts out pointing to square 0,0 in the upper left corner. (We will denote squares by a comma-separated pair of numbers, where the first number is the column and the second is the row. Thus square 0,1 is the leftmost yellow square in the above picture, and 10,9 is the rightmost blue square.)

Palgo has four commands for moving the pen around: `up`, `down`, `left`, and `right`. If the pen is down, it paints into the square beneath it and any squares it

passes over. If it is up, it can move over squares without painting them. The four direction commands can have a number parameter, such as 10 in the program above. This means to move 10 squares in the direction indicated, all the while painting the squares if the pen is down.

Though a sequential program like the one above can look rather complicated and can draw a pretty picture, the real power of Palgo appears when you use control structures. Click *Reset cells*, then select *Example 4—squares* from the pull-down menu, and click *Run*. Watch what happens in the painting window.

Here's the program, along with an explanation of how it works (the line numbers to the left are for reference only—they do not appear in the actual program):

```
1. define square (n)
2.      pen down
3.      down n
4.      right n
5.      up n
6.      left n
7. end
8. goto 0 0
9. color "red"
10. square(10)
11. goto 2 2
12. color "yellow"
13. square(4)
14. goto 12 12
15. color "blue"
16. square(5)
```

The first seven lines define a subalgorithm called *square*. The identifier n inside parentheses on the define line (line 1) is the parameter to *square*. (The textbook introduces parameters on pp. 260–264. Palgo has only *value* parameters.)

Line 8 directs the pen to go to position 0,0 in the upper left corner. Line 9 sets the current color to "red," and then line 10 *invokes* the subalgorithm named square, telling it to substitute the number 10 for its parameter n. As you might guess from reading the code inside the *square* subalgorithm, the parameter tells Palgo how big to draw the sides of the square.

After changing starting position and color in lines 11–12, the program invokes *square* again, asking it to draw a smaller square in line 13. Then a 5-unit square is drawn by line 16.

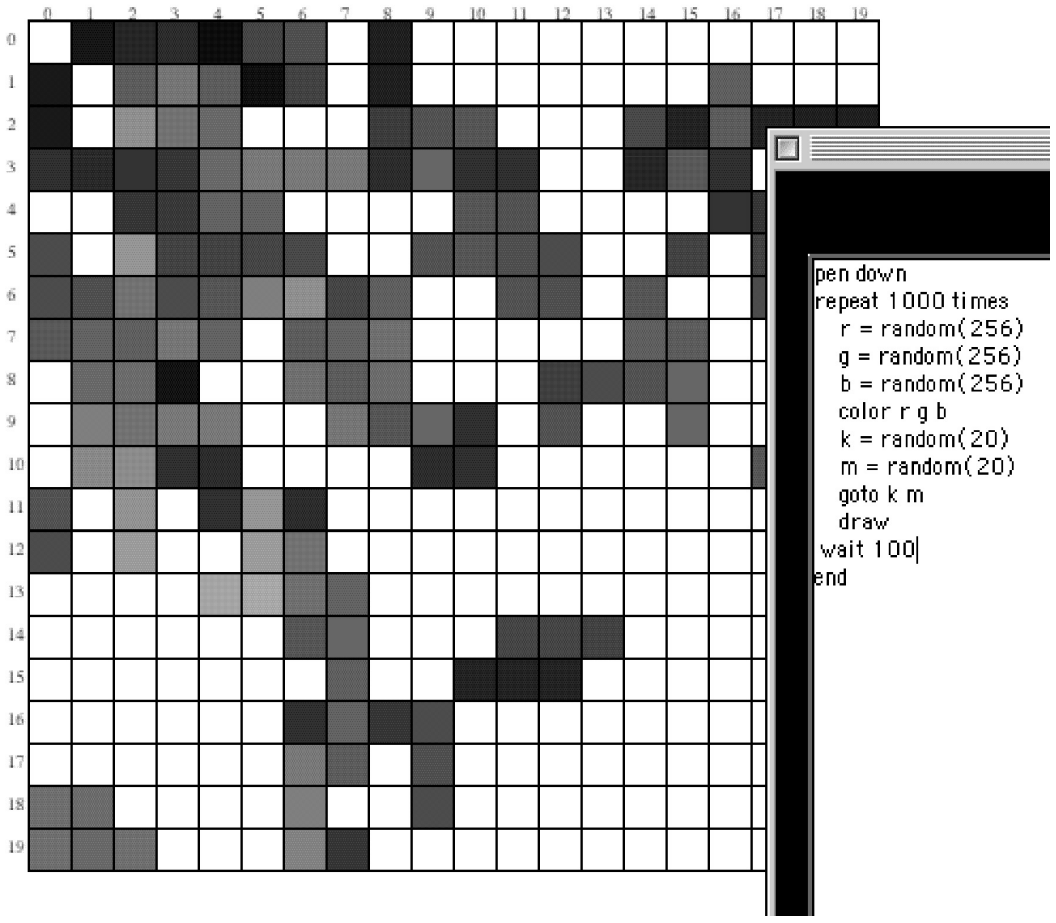
Compare the code in the edit window with the results in the painting window, and make sure you understand how the program works.

Now we'll take a look at a different program. Select *Example 3—random colored dots*. Press *Reset cells*, then press *Run*. Let the program go for a while, noting what happens, then read on.

Palgo has several ways to repeat sections of code. One is to surround the code with `repeat X times` and `end`. (The end keyword is used in Palgo for all control structures, in order to mark the end of their influence. We saw this with the `define` construct above.) To cause one or several lines to be executed many times, simply tuck them inside a repeat loop. The number of times can be either a constant, such as 1000 in this example, or a variable.

The code inside a repeat loop is called the *body* of the loop. This is the code that is performed over and over until the loop stops. *If* statements, as well as subalgorithm *define* statements, also have bodies. Notice that in our examples these bodies are

indented to set them apart from the rest of the code. This indenting doesn't affect the way the code runs, but experienced programmers use indenting to help make their code more readable.



Inside the repeat loop in this example is a sequence of simple Palgo statements, but there could have been `if` statements or even other repeat loops. Early computer scientists must have had a love for biology, even well before bioinformatics caught on, because much of the terminology of computer science is based on animal and plant metaphors. In this case, a repeat loop sitting inside the body of another repeat loop is said to be *nested*.

There are a few interesting things worth noting in the sequential body of our example. First is the use of a built-in function called `random()`. This pulls a random number that is 256 units wide out of a hat. Actually, it creates what is called a pseudo-random number, one that is based on some mathematical principle such as doing complicated and contorted divisions and additions, but that looks pretty random. The parameter to the built-in function, 256, tells Palgo to generate a random number between 0 and 255, inclusive.

Why use 256 as the parameter? As you might recall from Lab 3B, "Colorful Characters," the answer has to do with how computers represent colors. 256 is 2^8, and there are 8 bits per primary color. Each color is represented by three 8-bit numbers, called red, green, and blue. Many millions of colors can be produced by this scheme, which is discussed in the textbook on pages 77–81.

This Palgo program generates three random values for the red, green, and blue components. Then it makes the color by the statement

```
color r g b
```

Remember that since the names `r`, `g`, and `b` are not surrounded by double quotes, Palgo thinks of them as variables that contain integer values. In contrast, the statement `color "red"` sets the current color to red directly. But many of the approximately 16 million possible colors in the red-green-blue scheme do not have simple names like this, so sometimes we specify the color by numbers.

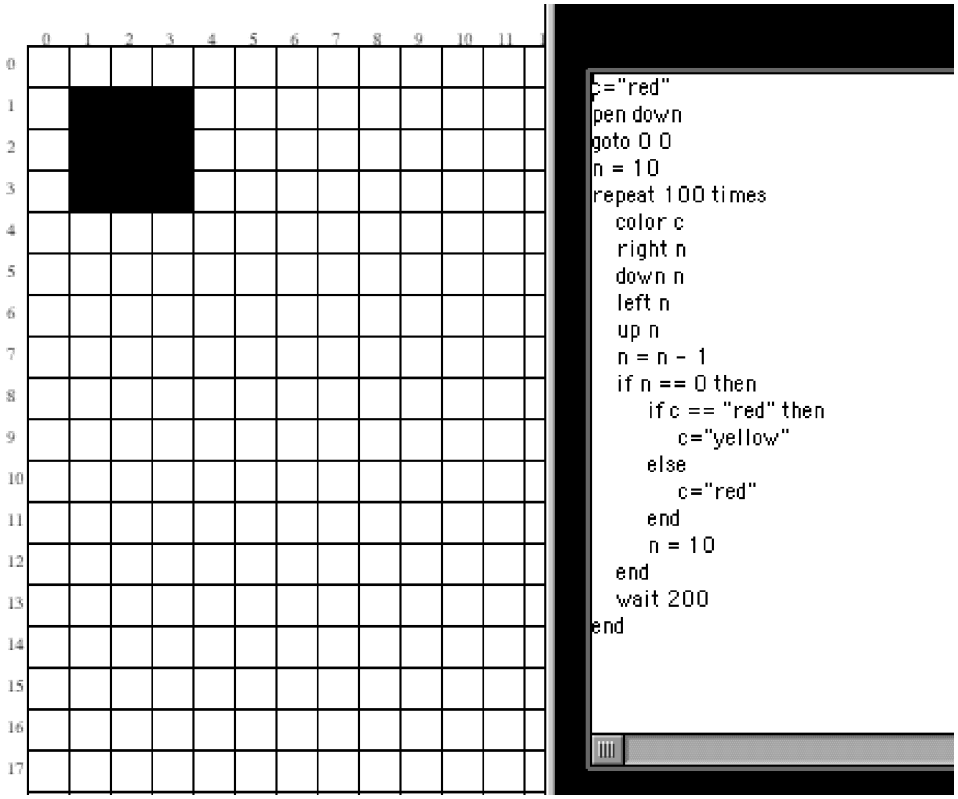
After we generate a random color, we generate a random position. Because our painting window consists of a square 20 cells to a side, we use `random(20)`. Then we go to that newly calculated spot with

```
goto k m
```

and paint the square. This command dips the invisible pen in the magical electronic inkwell and dabs the splash of color onto the screen. Finally, we wait for a short amount of time (100 milliseconds) before starting the whole thing over. Try changing the `wait` value from 100 to 10 and see if your screen doesn't get painted faster when you run the program.

Does it surprise you that seemingly simple painting tasks have rather involved algorithms behind them? This example points out how complex computer programming is, even for "simple" tasks. The computer, an untiring, uncomplaining slave, might make its way through thousands (or even millions) of lines of tedious code just to execute a seemingly simple drawing. (Of course, since this slave might be doing a billion calculations a second, I guess we'd better give it some respect!)

Let's look at a more complicated program, "Example 6 - red and yellow." Here's a screenshot of the program while it's running:



This program, which draws a succession of red and yellow shrinking boxes, has two important variables, whose identifiers are `c` and `n`. (Way too many of us older computer scientists grew up in the ancient FORTRAN era, when one-letter variable names were considered absolutely okay.) `c` holds a string, which is either "red" or "yellow" in this program. When it comes time to dip the invisible pen in the magical inkwell, the statement

```
color c
```

tells Palgo which color to use.

The variable `n` is used to determine the size of one side of the square being drawn. It starts out at 10 and then progressively gets smaller by using the statement

```
n = n - 1
```

An `if` statement inside the body of the loop checks to see if n's value is 0, because if it is, the program needs to swap colors and reset the size of the square side to 10.

One rather strange bit of hieroglyphics is the line

```
if n == 0 then
```

Did the programmer stutter at the keyboard? No, this statement is influenced by another bit of computer history. First, notice the line

```
n = n - 1
```

This looks weird, too, until you realize that this is not an algebraic statement of equality but rather an action command. It tells Palgo to evaluate the right-hand side of the equal sign and come up with a value, then stash that into the variable on the left-hand side. Such an action command is called an *assignment statement*, and the equal sign is the *assignment operator*.

Since = is the assignment operator, it can't be used for ordinary equality because computers, unlike humans, are terrible at using the context to determine what is meant. (No wonder computers are horrible at understanding French, German, or English!) Thus, ==, which is pronounced "equals," is used for ordinary equality.

Equals (==) functions as a *comparative operator*, meaning "is equal to". So the statement `if n == 0 then` translates into "if `n` equals `0` then ...". This is different from =, which as an assignment operator assigns a value to a variable—so in the statement `n = n - 1`, the variable `n` decreases by 1. (You already know several other comparative operators: > for "is greater than" and < for "is less than" are two examples.)

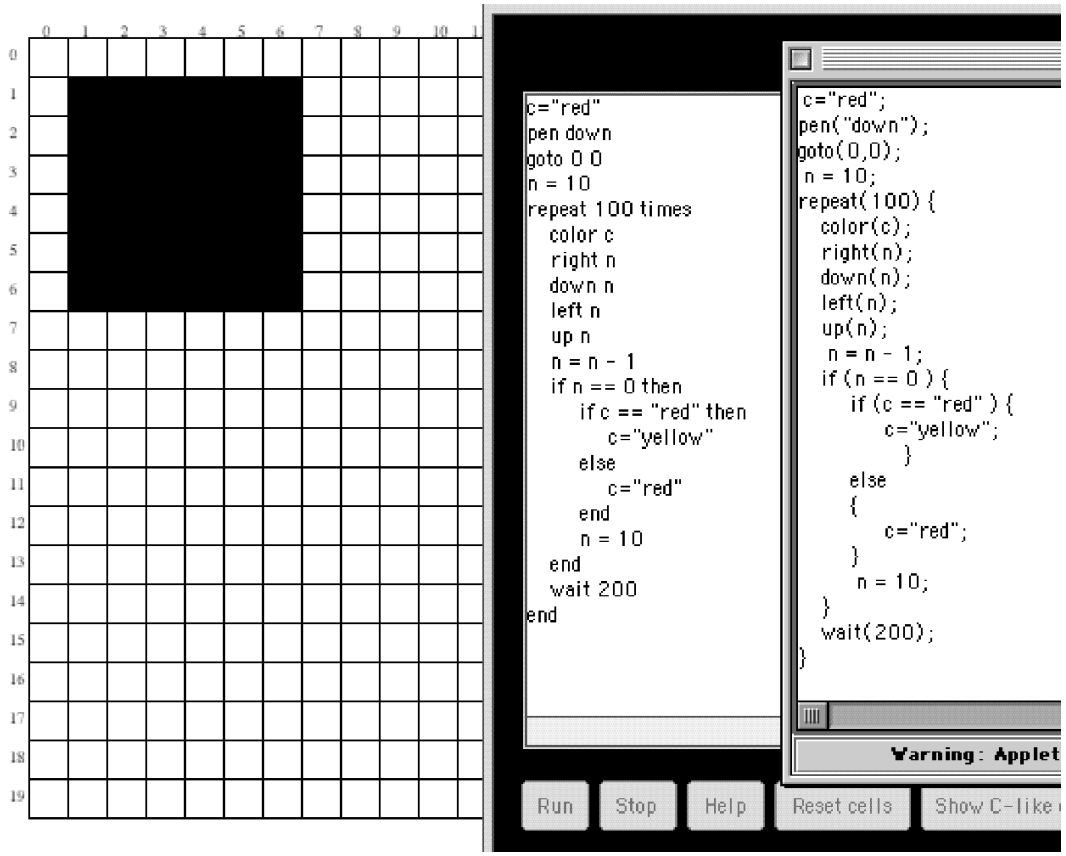
Let's see, didn't we mention some computer science history? Actually, several lines of history intersect here. The assignment operator, first used in FORTRAN, should really have been something more meaningful (like a left-pointing arrow, perhaps), but the IBM 026 keypunch machine commonly in use at the time only had a few symbols on it, and John Backus, who headed the FORTRAN compiler team, had to make do with what they had. In the 1970s, when Brian Kernighan and Dennis Ritchie created the C programming language, they were faced with the same sort of problem and, being familiar with FORTRAN, decided to retain the assignment operator as is. However, they hated the ugly symbol that FORTRAN used for testing equality, which was `.EQ.`, as in:

```
IF(X.EQ.5) GOTO 76
```

so they humanely chose == for equality testing. Their reasoning was clever. They believed that the assignment operator would appear about twice as often as the comparative operator, so it should be half as long. (Some linguists believe that the most frequently used words in human languages are almost always the shortest words.)

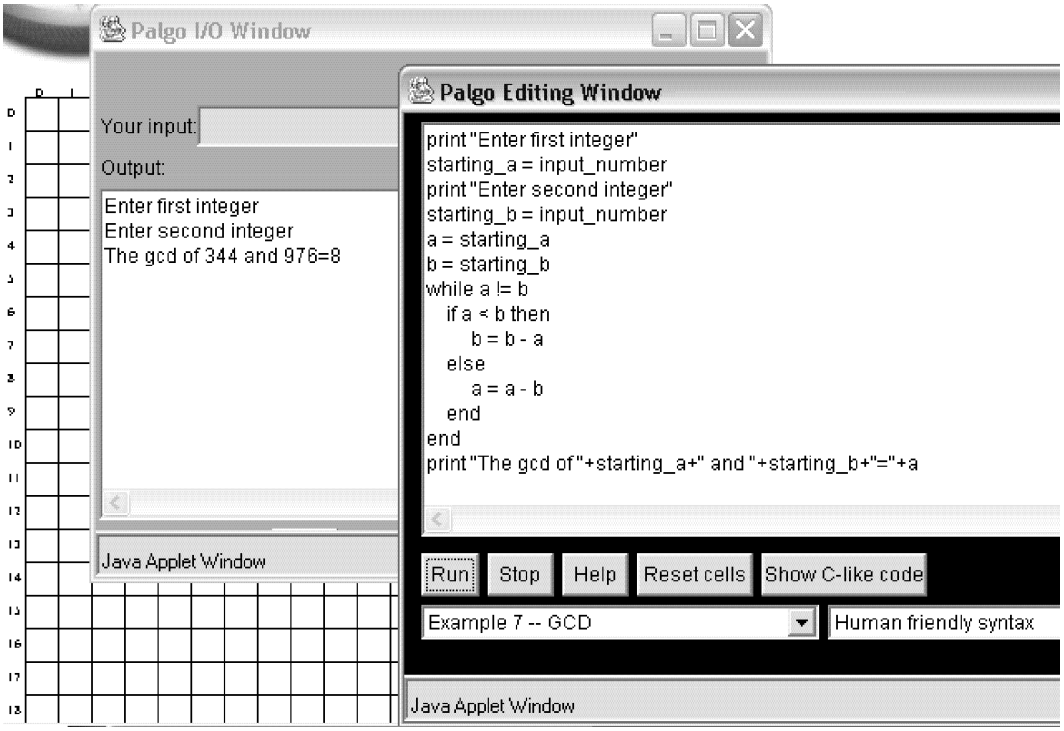
Enough computer science history and lore! There are a few other aspects of Palgo you should become familiar with.

Click the *Show C-like code* button. This displays a different form of the program, in a yellow window you can resize and reposition for more convenient viewing:



Palgo uses software tools to translate and execute programming language code. These tools include a crucial piece called the *parser*. You might be surprised to learn that the original parser was written to interpret a C-like programming language, replete with those scary curly braces and abundant semicolons. The applet author, realizing how cruel it is to force you to program in C, wrote a translator that permits the use of "human-friendly syntax," which is then converted into this C-like language to be used by the parser. (In fact, you can use the C-like syntax if you select *C-style syntax* from the pull-down menu choice at the bottom of the edit window. (You probably won't want to try it, though, unless you have some programming experience or are a glutton for punishment!) This illustrates the power and beauty of software; it is infinitely adaptable, reusable, and malleable.

Finally, as you saw above, Palgo has variables that can be used to compute many things—even some things not involving painting at all. Pull down *Example 7—GCD* and study the code. Notice that it uses no painting commands, but rather computes the greatest common divisor of two numbers that you give it.



The *print* statement and the *input* statement cause an orange window to appear. When the program asks for input, type any value in the small text area at the top and press *Return*. It is *very important* to press Return, because Palgo doesn't know you are done typing until you actually press the *Return* key. Any output appears in the larger text area below.

The Palgo function `input()` needs no parameters, but the empty parentheses are needed so that Palgo can tell this is a function. The `input()` function returns a character string. Inside your program, you must convert that string to a number by using the `atoi()` function (`atoi` stands for "ASCII to Integer" and is another tip of the hat to the C programming language), as shown above. Then the result of these nested functions is stored in a variable:

```
starting_a = atoi(input())
```

So Palgo can be used to do "regular," non-graphical programming too. Of course, you can also mix the two. Sometimes you need to do a lot of "regular" calculations to figure out where to put the pen. Palgo is a relatively bare-bones graphics system, but if you experiment with writing programs in it, you can become quite familiar with the concept of algorithms.

Tip

Palgo can be used as a standalone Java application. If you use Palgo as an application (not as an applet), you can load and save your programs. To run the Java application, navigate to the folder containing the Palgo files and double-click on the **run-application.bat** file.

Exercise 1

Name _____ Date _____

Section _____

As you work on the following steps, refer to the sample programs for models of what your Palgo programs should look like.

1) Start the "Palgo" applet.
2) In the editing window, type in a sequence of Palgo instructions. Here's what they should do:
   a) Set the color to red.
   b) Draw a line from 2,7 to 2,17.
   c) Set the color to green.
   d) Draw a vertical line starting at 2,7 that is 15 cells long, going down the screen.
3) Click on the *Run* button to run the program.
4) Arrange the edit and painting windows so they're both visible, then take a screenshot.

   Hint: To draw a line, go to the beginning cell. Then use right, down, left, or right for as many cells as needed. For example, to draw a vertical line from the bottom of the cell grid to the top in the second column, use the following commands:

   ```
   color "red"
   goto 1 19
   up 20
   ```

   Note: Since the rows and columns are numbered starting at 0 instead of 1, the first column is numbered 0, the second column is numbered 1, and so on.

Exercise 2

Name _____ Date _____

Section _____

1) Start the "Palgo" applet.
2) Select *Example 4—squares* from the pull-down menu.
3) In the editing window, you will alter the program as follows—but make sure to leave the definition of the square subalgorithm at the top unchanged!
4) Here's how you should alter the program:
   a) Remove everything after the square subalgorithm (the subalgorithm ends with the word `end`).
   b) Set the variable k to 1.
   c) Write a repeat loop that executes 10 times.
   d) Inside the loop, set the color to a random r, g, b value. This is shown on p. 106 of this lab manual.
   e) Use the goto statement to move to cell k,k in the grid.
   f) Invoke the square algorithm using variable k. This will create a square of size k. The first time through, when k is 1, the square will be 1×1. The second time through, the square will be 2×2, and so forth.
   g) Add 1 to variable k.
5) Run your program. Take a screenshot showing both the squares and the edit window.

Exercise 3

Name _____ Date _____

Section _____

1) Start the "Palgo" applet or clear it if it is already running.
2) Set the color to blue.
3) Using goto and up, down, right, and left statements, create the following letters side by side:

```
T H E
```

4) Take a screen shot, showing both the Palgo window as well as the cell grid after you run the program.

Exercise 4

Name _____ Date _____

Section _____

1) Start the "Palgo" applet or clear it if it is already running.
2) Type in the following program. (Make sure there are spaces between "draw," "i," and "k")

```
clear
color "orange"
k = 6
for i=3 to 10
     draw i k
     k = k + 1
end
```

3) What kind of line did this draw?

4) Be bold! Change the statement that alters k's value so that it goes up by 2 each time through the for loop. Rerun the program and take a screen shot. Is the new image a line or not?
5) Now change the for loop so that it looks like the following then rerun.

```
for i=3 to 10
     goto i k
     down 2
     k = k + 2
end
```

6) Compare this image to the last one. Which one is more of a line?

7) Most computer screens can't draw freehand like we can. Instead, computers can only paint into little cells, or *pixels*. So drawing lines other than pure horizontal or vertical involves a compromise. If you look at a ragged line from a distance, or if the *resolution* is high (meaning the number of cells in the grid is large), the line will look smoother. Try standing about 10 to 15 feet away from your computer screen and report if the image looks more like a line.

8) Another fun experiment is to get a strong magnifying glass and hold it up to your computer's screen. Look at letters on the screen and report what you see. Also, what do you notice about the colors under the magnifying glass?

9) Palgo lets you change the resolution of your cell grid with the `numcells` command. Here's a similar program that changes the number of cells from 20 on each size of the grid to 80 on each side. Then it draws a longer line. Run it and take a snapshot.

```
clear
numcells 80
color "orange"
k = 6
for i=3 to 100
     goto i k
     down 2
     k = k + 2
end
```

10) Does the line look smooth enough?

Deliverables

Turn in the screenshot showing your program after it finishes running and the edit window with your program in it.

If you are running Palgo as a standalone Java application, save your file and then print it out. Use a word processor or a text application (like Notepad) to print out the file. Your instructor may ask you to hand in the file electronically, too. Consult the instructor for details on how to do this.

Deeper Investigation

Programming is hard work! As you read through Chapters 8 and 9 of your textbook, you might find yourself working hard to absorb the meaning of `while` loops, `if` statements, and subalgorithms. Actually putting your new knowledge into practice and writing real programs is a new, but hopefully fun, challenge. Palgo has the ability to create extremely complicated, program-driven pictures. Play with it and see!

Write a function that draws a filled rectangle. There should be four parameters: starting row, starting column, width, and height. Then cannibalize the random dots program to draw rectangles of random sizes on the screen at random locations, using random colors. (*Cannibalize?* What that means, of course, is to study the code and steal or modify whatever might be useful for your program!)