15-112 Fall 18

Home

Syllabus

Schedule

Staff

Gallery

Piazza

Autolab

OH Queue

CMU 15-112 Fall 2018: Fundamentals of Programming and Computer Science
Homework 4 (Due Sunday 23-Sep, at 5pm)

- Reminder: all problems that are not explicitly marked COLLABORATIVE must be completed individually, as stated in the course syllabus.

- To start:
  1. Create a folder named 'week4'
  2. Download hw4.py to that folder.
  3. Edit hw4.py using Pyzo
  4. When you are ready, submit hw4.py to Autolab. For this hw, you may submit up to 7 times, but only your last submission counts.
- Do not use dictionaries or recursion this week.
- Do not hardcode the test cases in your solutions.
- Remember, we will grade your code based on whether it follows the 15-112 style guide! We may deduct up to 10 points from your overall grade for style errors. Writing your code with good style from the beginning is the best way to avoid losing points!
- If you are having trouble starting a particular problem, try reviewing algorithmic thinking techniques!

---

0. Style Meetings [0pts]
   Attend office hours with a TA and briefly discuss how to improve your programming style (based on your hw3 submission) in order to regain points lost on hw3 because of style errors. To make the best use of this time, come prepared by reading the feedback you received on style from hw3. The TA will discuss the most important deductions on hw3 and how to improve your tyle in general. At the TA's sole discretion, we will return lost points on hw3 due to style errors. Have this feedback and hw3 ready on your laptop when you speak to the TA. You must have this meeting by the main assignment deadline, Sunday at 5pm.

1. List Function Table [5pts]
   There are many different list functions that exist in Python which change the contents of the list. Some of these functions are destructive (changing the list they're called on), others are nondestructive (creating a new list as a result). Next to each of the functions listed below, write whether the function is destructive or non-destructive to the list a. Include your table in a triple-string comment in hw4.py.

   ```
   """
   c = a + b
   a += b
   a.append(x)
   a.insert(0, x)
   b = a[:i] + [x] + a[i:]
   a.remove(x)
   a.pop(0)
   b = a * 3
   a.reverse()
   reversed(a)
   a.sort()
   sorted(a)
   copy.copy(a)
   """
   ```

2. COLLABORATIVE: lookAndSay(a) [10pts]
   First, read about look-and-say numbers here. Then, write the non-destructive function lookAndSay(a) that takes a list of numbers and returns a list of numbers that results from "reading off" the initial list using the look-and-say method, using tuples for each (count, value) pair. For example:
   ```
   lookAndSay([]) == []
   lookAndSay([1,1,1]) == [(3,1)]
   lookAndSay([-1,2,7]) == [(1,-1),(1,2),(1,7)]
   lookAndSay([3,3,8,-10,-10,-10]) == [(2,3),(1,8),(3,-10)]
   ```
   Hint: you'll want to keep track of the current number and how many times it has been seen.

3. inverseLookAndSay(a) [10pts]
   Write the non-destructive function inverseLookAndSay(a) that does the inverse of the function lookAndSay from the previous problem, so that, in general:
   ```
   inverseLookAndSay(lookAndSay(a)) == a
   ```
   Or, in particular:
   ```
   inverseLookAndSay([(2,3),(1,8),(3,-10)]) == [3,3,8,-10,-10,-10]
   ```

4. bestScrabbleScore(dictionary, letterScores, hand) [25pts]
   Background: In a Scrabble-like game, players each have a hand, which is a list of lowercase letters. There is also a dictionary, which is a list of legal words (all in lowercase letters). And there is a list of letterScores, which is length 26, where letterScores[i] contains the point value for the ith character in the alphabet (so letterScores[0] contains the point value for 'a'). Players can use some or all of the tiles in their hand and arrange them in any order to form words. The point value for a word is 0 if it is not in the dictionary, otherwise it is the sum of the point values of each letter in the word, according to the letterScores list (pretty much as it works in actual Scrabble).

   In case you are interested, here is a list of the actual letterScores for Scrabble:

```
letterScores = [
#  a, b, c, d, e, f, g, h, i, j, k, l, m
   1, 3, 3, 2, 1, 4, 2, 4, 1, 8, 5, 1, 3,
#  n, o, p, q, r, s, t, u, v, w, x, y, z
   1, 1, 3,10, 1, 1, 1, 1, 4, 4, 8, 4,10
]
```

   Note that your function must work for any list of letterScores as is provided by the caller.

   With this in mind, write the function bestScrabbleScore(dictionary, letterScores, hand) that takes 3 lists -- dictionary (a list of lowercase words), letterScores (a list of 26 integers), and hand (a list of lowercase characters) -- and returns a tuple of the highest-scoring word in the dictionary that can be formed by some arrangement of some subset of letters in the hand, followed by its score. In the case of a tie, the first element of the tuple should instead be a list of all such words in the order they appear in the dictionary. If no such words exist, return None.
   The dictionary in this problem is a list of words, and thus not a true Python dictionary (which we haven't taught you and you may not use in this assignment)! It is OK to loop through the dictionary, even if the dictionary we provide is large.

   Note 0: You should definitely write helper functions for this problem! In fact, try to think of at least two helper functions you could use before writing any code at all.

   Note 1: You may not use itertools for this problem! In fact, you shouldn't need to create permutations of the letters at all...

5. COLLABORATIVE: drawStar(canvas, centerX, centerY, diameter, numPoints, color, winWidth=500, winHeight=500) [15pts] [manually-graded]
   Write the function drawStar which takes a canvas and the star's center coordinates, diameter, number of points, and color (and optional parameters winWidth and winHeight), and produces a star based on that specification. To draw a star, we need to identify where to place each of the inner and outer points, then draw them all together as a polygon.

   The outer points of the star should be evenly placed on a circle based on the specified diameter, with the first point at a 90 degree angle. The inner points should then be placed on a circle 3/8 the size of the first circle, halfway between the pairs of outer points. (We use this ratio to make a nice-looking five-pointed star. Actually, the best inner circle would be about 38.2% the size of the outer circle; a little trigonometry and problem-solving will tell you why! But 3/8 is close enough.) An example of how these circles work is shown below.

15-112
Fall 18

Home

Syllabus

Schedule

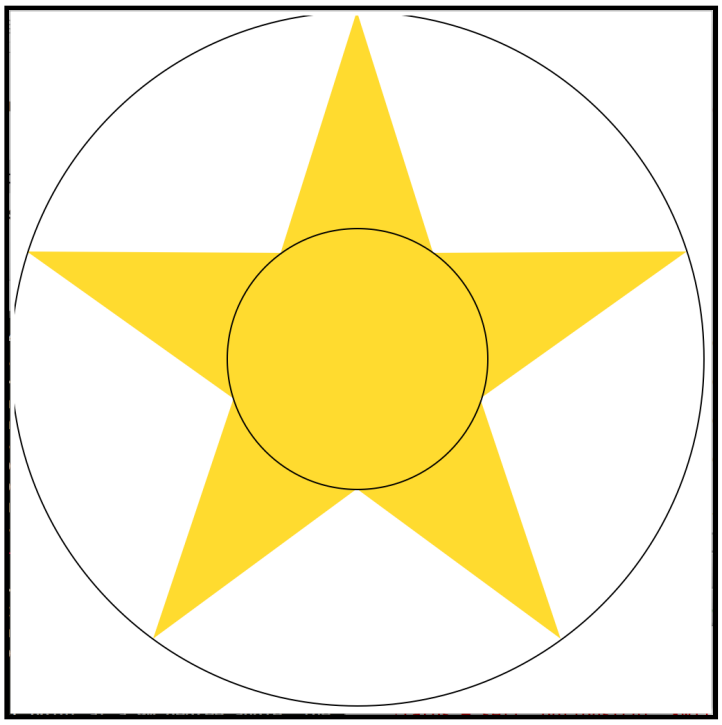
Staff

Gallery

Piazza

Autolab

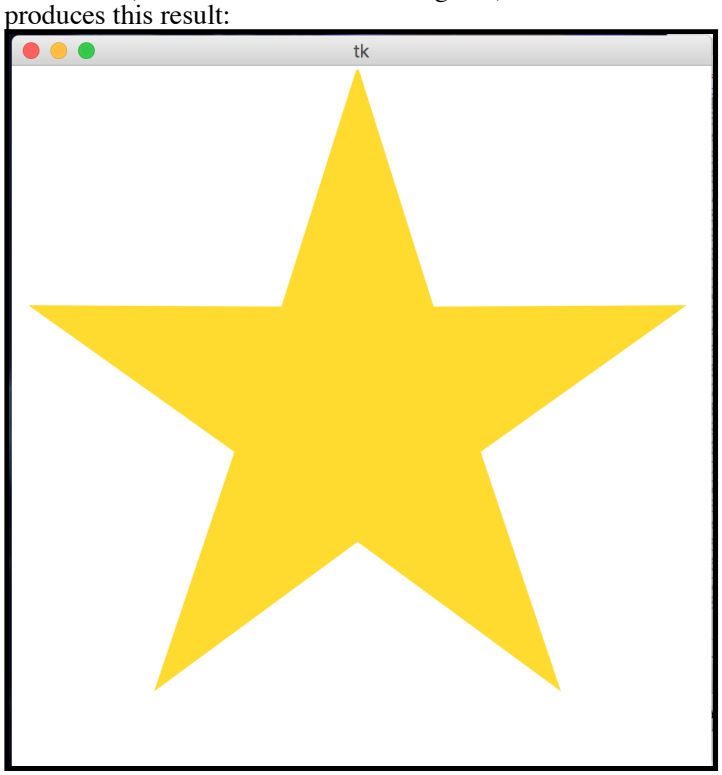
OH Queue



For example, this call:

```
drawStar(canvas, 250, 250, 500, 5, "gold")
```

produces this result:



And this call:

```
drawStar(canvas, 300, 400, 100, 4, "blue")
```

produces this result:

15-112
Fall 18

Home

Syllabus

Schedule

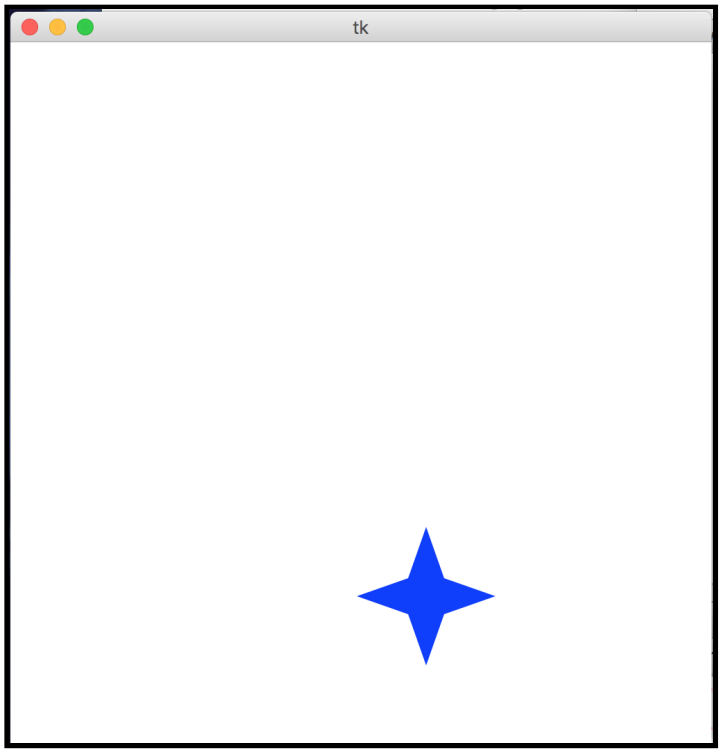
Staff

Gallery

Piazza

Autolab

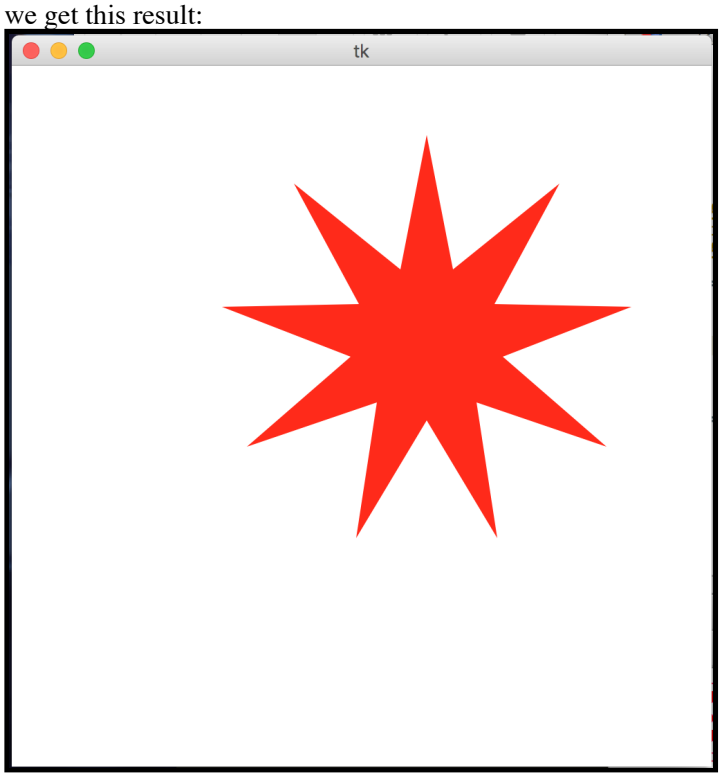
OH Queue



And if we add a few more points:

```
drawStar(canvas, 300, 200, 300, 9, "red")
```

we get this result:



6. drawCirclePattern(n, winWidth=500, winHeight=500) [15 pts] [manually-graded]
   Write the function drawCirclePattern(n, winWidth=500, winHeight=500) that takes a positive int (and optional parameters winWidth and winHeight) and displays a graphics window, inside of which it draws the nxn version of this picture:

15-112
Fall 18

Home

Syllabus

Schedule

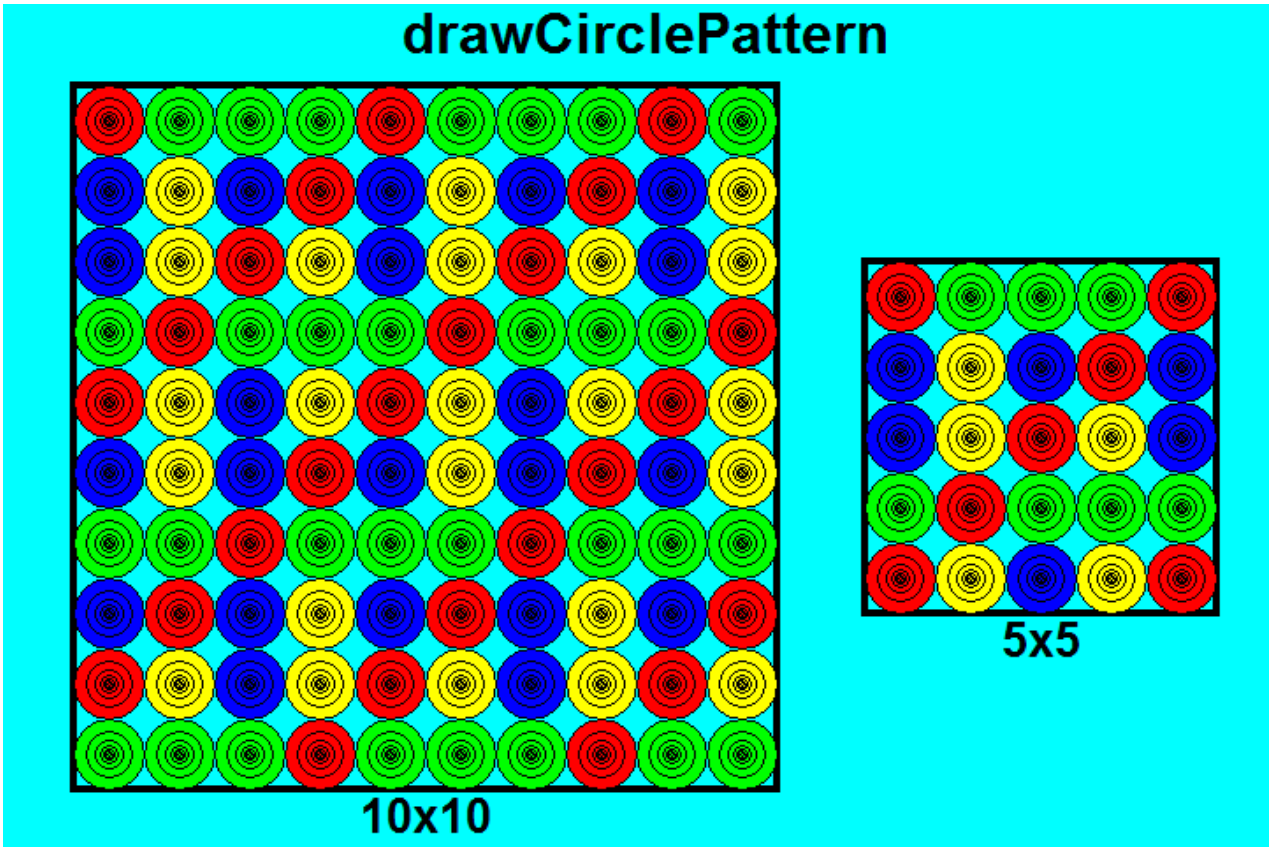
Staff

Gallery

Piazza

Autolab

OH Queue



While the image above shows two circle patterns, you only need to draw one at a time. For example, the left image above was drawn with 10 rows (and 10 columns), and the right image with 5 rows (and 5 columns). The pattern consists of "buttons" (or perhaps "bullseyes"), where each button is really several circles drawn on top of each other. Each circle in a "button" has a radius 2/3rds as large as the next-larger circle, which continues until the radius is less than 1. As for the color of each button, here are the rules to follow:

Rule 1: Starting from the left-top corner, every 4th diagonal is entirely red.
Rule 2: For the non-red buttons, starting from the top row, every 3rd row is entirely green.
Rule 3: For the non-red and non-green buttons, starting from the second-to-leftmost column, every 2nd column is entirely yellow.
Rule 4: Any non-red, non-green, and non-yellow button is blue.

Note that these rules can be fairly easily implemented using a single if-elif-elif-else statement. Inside that statement, you might want to set a variable, say one named "color", based on each of these four conditions. Then you could draw with fill=color.

Note: The drawing should mostly fill the window, and the 10x10 case should be about the same size as the 10x10 case in the image above, but the exact dimensions of the drawing are up to you.

7. COLLABORATIVE: runSimpleTortoiseProgram(program, winWidth=500, winHeight=500) [20 pts, manually-graded]
In addition to the Tkinter, which we all know and love, Python usually comes with another graphics package called "Turtle Graphics", which you can read about here. We will definitely not be using turtle graphics in this problem (and you may not do so in your solution!), but we will instead implement a small turtle-like (or maybe turtle-inspired) graphics language of our own. We'll call it Tortoise Graphics.

   First, we need to understand how Tortoise Graphics programs work. Your tortoise starts in the middle of the screen, heading to the right. You can direct your tortoise with the following commands:

   - color name
     Set the drawing color to the given name, which is entered without quotes, and which can be "red", "blue", "green", or any other color that Tkinter understands. It can also be "none", meaning to not draw.

   - move n
     Move n pixels straight ahead, where n is a non-negative integer, while drawing a 4-pixel-wide line in the current drawing color. If the drawing color is "none", just move straight ahead without drawing (that is, just change the tortoise's location).

- left n
  Turn n degrees to the left, without moving, where n is a non-negative integer.

- right n
  Turn n degrees to the right, without moving, where n is a non-negative integer.

Commands are given one-per-line. Lines can also contain comments, denoted by the hash sign (#), and everything from there to the end-of-line is ignored. Blank lines and lines containing only whitespace and/or comments are also ignored.

With this in mind, write the function runSimpleTortoiseProgram(program, winWidth=500, winHeight=500) that takes a program as specified above and runs it, displaying a window (which is 500x500 by default) with the resulting drawing from running that program. Your function should also display the tortoise program in that window, in a 10-point font, in gray text, running down the left-hand side of the window (10 pixels from the left edge). Don't worry if the program is longer than can fit in the window (no need to scroll or otherwise deal with this). Also, you are not responsible for any syntax errors or runtime errors in the tortoise program.

For example, this call:

```
runSimpleTortoiseProgram("""
# This is a simple tortoise program
color blue
move 50

left 90

color red
move 100

color none # turns off drawing
move 50

right 45

color green # drawing is on again
move 50

right 45

color orange
move 50

right 90

color purple
move 100
""", 300, 400)
```

produces this result in a 300x400 window:

15-112
Fall 18

Home

Syllabus

Schedule

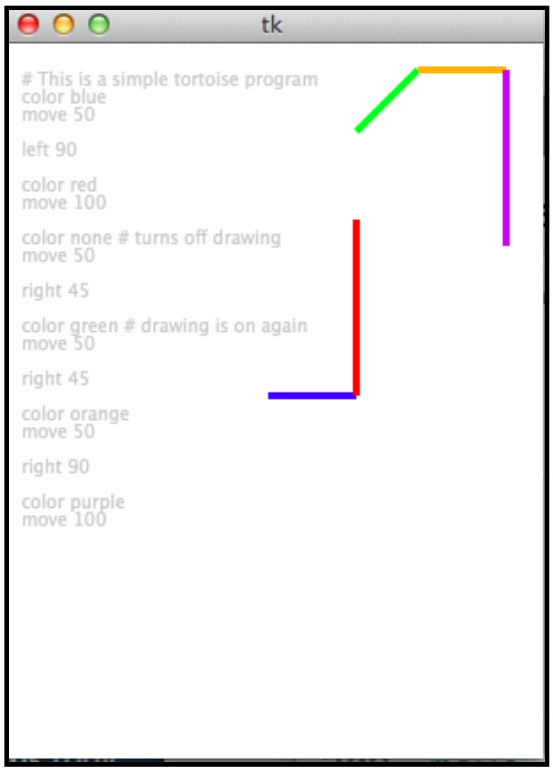
Staff

Gallery

Piazza

Autolab

OH Queue



And this call:

```
runSimpleTortoiseProgram("""
# Y
color red
right 45
move 50
right 45
move 50
right 180
move 50
right 45
move 50
color none # space
right 45
move 25

# E
color green
right 90
move 85
left 90
move 50
right 180
move 50
right 90
move 42
right 90
move 50
right 180
move 50
right 90
move 43
right 90
move 50  # space
color none
move 25

# S
color blue
move 50
left 180
```

15-112
Fall 18

Home

Syllabus

Schedule

Staff

Gallery

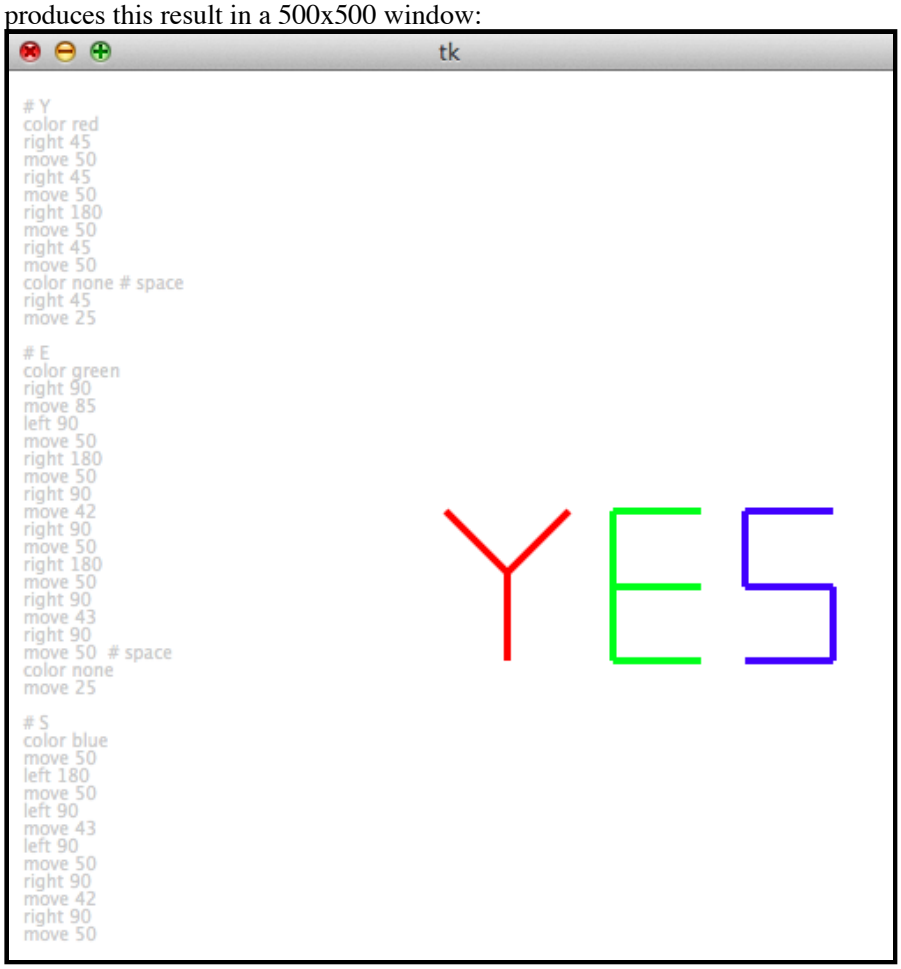
Piazza

Autolab

OH Queue

```
move 50
left 90
move 43
left 90
move 50
right 90
move 42
right 90
move 50
""")
```

produces this result in a 500x500 window:



- Bonus/Optional: drawNiceRobot() [1pt] [manually-graded]
  Write the function drawNiceRobot() that (you guessed it!) draws a nice robot! This is not meant to be very difficult. We just want to see some really cool robots while grading your homework. To get this bonus point, your function must make a drawing using Tkinter that meets the following criteria:
  1. Easily identifiable as a robot
  2. Is in a 500x500 window
  3. Includes at least 10 shapes total, including at least one oval (or circle), rectangle, non-rectangular polygon, and line
  4. Uses at least 4 colors

  Do not use anything we haven't learned in class or in the notes through Week 4! No extra files, no importing anything other than Tkinter! Have fun!

- Bonus/Optional: runSimpleProgram(program, args) [3 pts]
  First, carefully watch this video that describes this problem:

Then, write the function runSimpleProgram(program, args) that works as described in the video, taking a legal program (do not worry about syntax or runtime errors, as we will not test for those cases) and runs it with the given args, and returns the result.

Here are the legal expressions in this language:

- [Non-negative Integer]
  Any non-negative integer, such as 0 or 123, is a legal expression.

- A[N]
  The letter A followed by a non-negative integer, such as A0 or A123, is a legal expression, and refers to the given argument. A0 is the value at index 0 of the supplied args list. It is an error to set arg values, and it is an error to get arg values that are not supplied. And you may ignore these errors, as we will not test for them!

- L[N]
  The letter L followed by a non-negative integer, such as L0 or L123, is a legal expression, and refers to the given local variable. It is ok to get an unassigned local variable, in which case its value should be 0.

- [operator] [operand1] [operand2]
  This language allows so-called prefix expressions, where the operator precedes the operands. The operator can be either + or -, and the operands must each be one of the legal expression types listed above (non-negative integer, A[N] or L[N]).

And here are the legal statements in this language (noting that statements occur one per line, and leading and trailing whitespace is ignored):

- ! comment
  Lines that start with an exclamation (!), after the ignored whitespace, are comments and are ignored.

- L[N] [expr]
  Lines that start with L[N] are assignment statements, and are followed by the expression (as described above) to be stored into the given local variable. For example: `L5 - L2 42` This line assigns (L2 - 42) into L5.

- [label]:
  Lines that contain only a lowercase word followed by a colon are labels, which are ignored except for when they are targets of jump statements.

- JMP [label]
  This is a jump statement, and control is transferred to the line number where the given label is located. It is an error for such a label to not exist, and you may ignore that error.

- JMP+ [expr] [label]
  This is a conditional jump, and control is transferred to the line number where the given label is located only if the given expression is positive. Otherwise, the statement is ignored.

- JMP0 [expr] [label]
  This is another kind of conditional jump, and control is transferred only if the given expression is 0.

- RTN [expr]
  This is a return statement, and the given expression is returned.

Hints:

1. Do not try to translate the program into Python! Even if you could do so, it is not allowed here. Instead, you are going to write a so-called interpreter. Just keep track of the local variables, and move line-by-line through the program, simulating the execution of the line as appropriate.
2. You will find it useful to keep track of the current line number.
3. How long do you run the program? Until you hit a RTN statement! You may assume that will always eventually happen.
4. We used strip, split, and splitlines in our sample solution, though you of course may solve this how you wish.

Finally, here is a sample test function for you. You surely will want to add some addition test cases. In fact, a hint would be to build your function incrementally, starting with the simplest test cases you can think up, which use the fewest expression and statement syntax rules. Then add more test cases as you implement more of the language.

```
def testRunSimpleProgram():
    print("Testing runSimpleProgram()...", end="")
    largest = """! largest: Returns max(A0, A1)
                  L0 - A0 A1
                  JMP+ L0 a0
                  RTN A1
                  a0:
                  RTN A0"""
    assert(runSimpleProgram(largest, [5, 6]) == 6)
    assert(runSimpleProgram(largest, [6, 5]) == 6)

    sumToN = """! SumToN: Returns 1 + ... + A0
                ! L0 is a counter, L1 is the result
                L0 0
                L1 0
                loop:
                L2 - L0 A0
                JMP0 L2 done
                L0 + L0 1
                L1 + L1 L0
                JMP loop
                done:
                RTN L1"""
    assert(runSimpleProgram(sumToN, [5]) == 1+2+3+4+5)
    assert(runSimpleProgram(sumToN, [10]) == 10*11//2)
    print("Passed!")
```

Copy

15-112
Fall 18

Home

Syllabus

Schedule

Staff

Gallery

Piazza

Autolab

OH Queue