CS151 Notes on Parameterized Strings

Stephanie R Taylor

Did anyone try to draw a parallelogram or trapezoid for Project 8 or 9? It wasn't straight-forward because the current design allows for only one angle. Likewise, you had to get rather clever to create a rectangle. What if we could specify the angle for every turn and the distance for every forward move?

Using some additional code, we can do that. This code will "parse" the string of characters to make sense out of it. Parsing is a common idea in computer science – we analyze a string of characters by looking at them one at a time, and build up some values that are related to the string's content.

In lab, we will add support for numeric parameters preceding the F, +, and - symbols. E.g. (5)F means "forward(5*distance)" (i.e. go 5 timfes as far as you would with a regular old F). And (120)+ means "turn right 120 degrees". We will call the values between the parentheses *modifiers*. The modifier for F acts as a scale factor. The modifier for + and - acts as the angle (not a scale to the angle). And here is a rule about the modifiers that will become important later. If we have a modifier, e.g. (5), then it must be followed immediately by a +, - , or F. Or else we won't use it. E.g. (5)<yL>F will behave the same as <yL>F (the 5 can't modify the F unless it immediately precedes it).

This involves only a small amount of code, but the design is worth significant attention, so that is what we will do in class today.

Our goal will be to draw an isosceles triangle, using the string

F(135)−(1.4142135623730951)F(135)−F(90)−

Updating drawString

We need to update TurtleInterpreter.drawString to handle this new option.

The current basic structure of drawString is this:

```
 # preliminaries
stack = []

# for each symbol in dstring
for c in dstring:
        # interpret the symbol as a turtle command
        if c == 'F':
                turtle.forward(distance)
        elif c == '+':
                turtle.left(angle)
        elif c == '-':
                turtle.right(angle)

# postliminaries
turtle.update()
```

We need to augment the code within the loop. Why? Because not every symbol leads directly to a turtle command. In particular, what happens when the symbol is a parenthesis? We learn that we are about to start seeing characters that together make a modifier. That means we need to store some information indicating that we are in "modifier-grabbing mode". This is information about the context or current "state" of the function – sort of like the stack of positions and headings. Then, when we encounter the close parenthesis, we need to leave modifier-grabbing mode and return to turtle-drawing mode.

This means our new structure will look more like this

```
# preliminaries
stack = []
# set-up for parameter handling

# for each symbol in dstring
for c in dstring:
        # If the character is an open parenthesis and we are not
        # in modifier-grabbing mode
                # put us into modifier-grabbing-mode
                # initialize a modifier-string variable to ''
                # continue to the next iteration of the loop
        # elif the character is close parenthesis
                # put us out of modifier-grabbing-mode
                # turn the modifier string into a float and store
                #   the result in a modifier variable
                # continue to the next iteration of the loop
        # elif we are in modifier-grabbing-mode
                # add this character to the modifier string
                # continue to the next iteration of the loop

        # if we get this far, then the symbol was not part of
        # a modifier, so
        # interpret the symbol as a turtle command
        if c == 'F':
                # but we may need to use a modifier here
                turtle.forward(distance)
        elif c == '+':
                # but we may need to use a modifier here
                turtle.left(angle)
        elif c == '-':
                # but we may need to use a modifier here
                turtle.right(angle)

# postliminaries
turtle.update()
```

Let's separate our coding into two steps. First, let's see if we can properly detect the modifiers. We will see if we can do that and still draw the string.

How do we

- Indicate that we are in (or not in) modifier-grabbing mode?

  Use a variable (like stack) named mod_grab. Set it to True if we are in

modifier-grabbing mode and False if we are not.

- Store the modifier string as we are building it up in modifier-grabbing mode?

  Use a variable named mod_string. Set it to '' when we first enter modifier-grabbing mode. Add characters to it while we are in modifier-grabbing mode.

- Store the modifier as a number once we are done grabbing it?

  Use a variable named mod_value. Set it to float(mod_string) when we encounter the close parenthesis.

This means we add these lines of code to the outline

```
# preliminaries
stack = []
# set-up for parameter handling
mod_string = ''
mod_grab = False
mod_value = None

# for each symbol in dstring
for c in dstring:
        # If the character is (
        if c == '(' and not mod_grab:
                # put us into modifier-grabbing-mode
                mod_grab = True
                mod_string = ''
                continue
        elif c == ')':
                # put us out of modifier-grabbing-mode
                mod_grab = False
                # and save the value
                mod_value = float(mod_string)
                continue
        elif mod_grab:
                # add this character to the number string
                mod_string += c
                continue

        # if we get this far, then the symbol was not part of
        # a parameter, so
        # interpret the symbol as a turtle command
        if c == 'F':
                # but we may need to use the modifier here
                turtle.forward(distance)
        elif c == '+':
                # but we may need to use a modifier here
                turtle.left(angle)
        elif c == '-':
                # but we may need to use a modifier here
                turtle.right(angle)

# postliminaries
turtle.update()
```

We can test it with this code:

```
terp = TurtleInterpreter( 500, 500 )
terp.place( 0, 0 )
terp.drawString( "F(135)-(1.4142135623730951)F(135)-F(90)-", 50, 120 )
terp.hold()
```

And it should draw an equilateral triangle. The modifiers will be ignored and the angle is set to 120, so this amounts to an equilateral triangle.

Next, let's use the modifiers. That means we need to update the code that handles the F, +, and - cases.

Out basic strategy will be to have the mod_value variable contain None if there is no modifier for a given turtle command symbol. If the value isn't None, then the F, -, and + code must use it.

How do we

- use the modifier when we have one and ignore it when we don't?

  In the code handling F, -, and +, we need to check to see if the value of mod_value is None. If it is, then use the old code. If it isn't then use code that involves mod_value.

- indicate that a modifier has been used?

  Remember that a modifier must be used right away. So that mean we can assume that if we make it to the bottom of the if-elif statements that handle turtle symbols, then one of those symbols must have used the modifier. And if it didn't, then that's too bad. The modifier must be erased. So we simply set the value of mod_value to None after the if-elif statements.

Here is the updated code:

```
def drawString(self, dstring, distance, angle):
    # preliminaries
    stack = []
    # set-up for parameter handling
    mod_string = ''
    mod_grab = False
    mod_value = None

    # for each symbol in dstring
    for c in dstring:
        # If the character is (
        if c == '(' and not mod_grab:
            # put us into modifier-grabbing-mode
            mod_grab = True
            mod_string = ''
            continue
        elif c == ')':
            # put us out of modifier-grabbing-mode
            mod_grab = False
            # and save the value
            mod_value = float(mod_string)
            continue
        elif mod_grab:
            # add this character to the number string
            mod_string += c
            continue

        # if we get this far, then the symbol was not part of
        # a parameter, so
        # interpret the symbol as a turtle command
        if c == 'F':
            # but we may need to use the modifier here
            if mod_value == None:
                turtle.forward(distance)
            else:
                turtle.forward(distance*mod_value)
        elif c == '+':
            if mod_value == None:
                turtle.left(angle)
            else:
                turtle.left(mod_value)
        elif c == '-':
            if mod_value == None:
                turtle.right(angle)
```

```
            else:
                turtle.right(mod_value)

            # if we made it this far, then we executed a turtle command
            # so we have use the modifier (if there was one). To make
            # sure the next command is not accidentally modified, let's
            # clear the modifier by setting its value to None
            mod_value = None

    # postliminaries
    turtle.update()
```

Now if we test it with the same code, we should see an isosceles triangle.