Assignment 3A: Fractals

Assignment by Chris Gregg, based on an assignment by Marty Stepp and Victoria Kirst. Originally based on a problem by Julie Zelenski and Jerry Cain.

JULY 12, 2017

Outline and Problem Description

*Note: because of the reduced time for this assignment, the Recursive Tree part of the assignment is **optional**.*

This problem focuses on recursion: you will write several recursive functions that draw graphics. Recursive graphical patterns are also called *fractals*. It is fine to write **"helper" functions** to assist you in implementing the recursive algorithms for any part of the assignment. Some parts of the assignment essentially *require* a helper to implement them properly. However, it is up to you to decide which parts should have a helper, what parameter(s) the helpers should accept, and so on. You can declare function prototypes for any such helper functions near the top of your **.cpp** file. (Don't modify the provided **.h** files to add your prototypes; put them in your own **.cpp** file.) We provide a GUI (graphical user interface) for you that will help you run and test your code.

Files and Links:

Project Starter ZIP:

(open **Fractals.pro**)

Turn In

fractals.cpp

Demo Jar

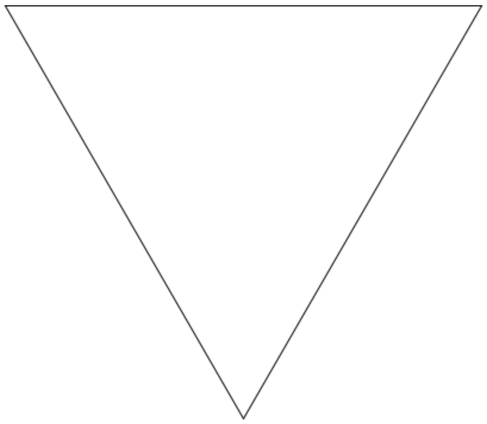
(note: ignore 20 Questions and Fractals flood fill. Also, the jar does not demonstrate Mandelbrot)

Sierpinski Triangle

If you search the web for fractal designs, you will find many intricate wonders beyond the Koch snowflake illustrated in Chapter 8. One of these is the Sierpinski Triangle, named after its inventor, the Polish mathematician Waclaw Sierpinski (1882-1969). The order-1 Sierpinski Triangle is an equilateral triangle, as shown in the diagram below.

For this problem, you will write a recursive function that draws the Sierpinski triangle fractal image. Your solution should not use any loops; you must use recursion. Do not use any data structures in your solution such as a `Vector`, `Map`, arrays, etc.

```
void drawSierpinskiTriangle(GWindow& gw, double x, double y, double size, int order)
```

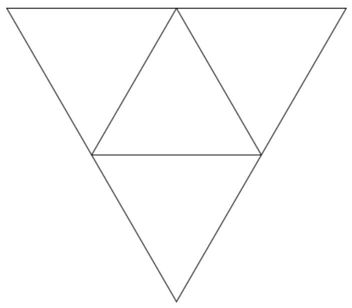


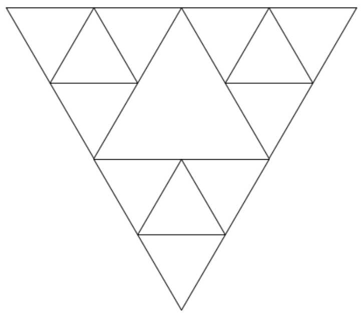
Order 1 Sierpinski triangle

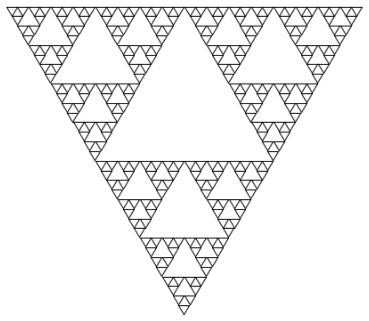
To create an order-*K* Sierpinski Triangle, you draw three Sierpinski Triangles of order *K*-1, each of which has half the edge length of the larger order-K triangle you want to draw. Those three triangles are positioned in what would be the corners of the larger triangle; together they combine to form the larger triangle itself. Take a look at the Order-2 Sierpinski triangle below to get the idea.

Your function should draw a black outlined Sierpinski triangle when passed the following parameters: a reference to a graphical window, the x/y coordinates of the top/left of the triangle (note that the triangles you draw will be inverted), the length of each side of the triangle, and the order of the figure to draw (i.e., 1 for an order-1 triangle). The provided code already contains a `main` function that pops up a graphical user interface to allow the user to choose various x/y positions, sizes, and orders. When the user clicks the appropriate button, the GUI will call your function and pass it the relevant parameters as entered by the user. The rest is up to you.

Some students mistakenly think that some levels of Sierpinski triangles are to be drawn pointing upward and others downward; this is incorrect. The upward-pointing triangle in the middle of the Order-2 figure is not drawn explicitly, but is instead formed by the sides of the other three downward-pointing triangles that are drawn. That area, moreover, is not recursively subdivided and will remain unchanged at every order of the fractal decomposition. Thus, the Order-3 Sierpinski Triangle has the same open area in the middle.


Order-2

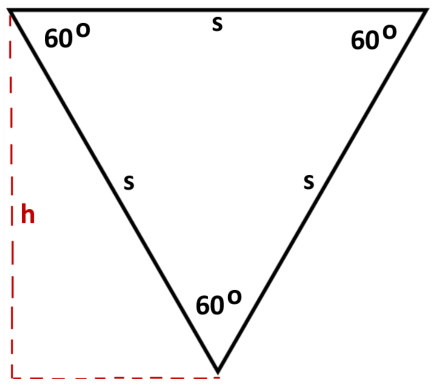

Order-3


Order-6

We have not learned much about the **`GWindow`** class or drawing graphics, but you do not need to know much about it to solve this problem (for your reference, however, here is the complete **`GWindow`** documentation). The only member function you will need from **`GWindow`** for this problem is the **`drawLine()`** function:

Member	**Description**
`gw.drawLine(x1, y1, x2, y2);`	draws a line from point $(x1, y1)$ to point $(x2, y2)$

You may find that you need to compute the height of a given triangle so you can pass the proper x/y coordinates to your function or to the drawing functions. Keep in mind that the height *h* of an equilateral triangle is not the same as its side length *s*. The diagram below shows the relationship between the triangle and its height. You may want to look at information about equilateral triangles on Wikipedia and/or refresh your trigonometry.



Computing an equilateral triangle's height from its side length

One particular type of solution we want you to avoid would be to write a **pair of functions**, one to draw "downward-pointing" triangles and another to draw "upward-pointing" triangles, and to have each function call the other in an alternating fashion. This is a poor solution because it does not capture the self-similarity inherent in this fractal figure.

Another thing you should avoid is **re-drawing** the same line multiple times. If your code is structured poorly, you end up re-drawing a line (or part of a line) that was already drawn, which is unnecessary and inefficient. If you aren't sure whether your solution is redrawing lines, try making a counter variable that is incremented each time you draw a line and checking its value against the number of lines you know your triangle ought to require.

If the order passed is 0, your function should not draw anything. If the x or y coordinates, the order, or the size passed is negative, your function should throw a string **exception**. Otherwise, you may assume that the window passed is large enough to draw the figure at the given position and size.

Expected output: You can compare your graphical output against the image files below, which are already packed into the starter code and can be compared against by clicking the "compare output" icon in the provided GUI shown here:

Please note that due to minor differences in pixel arithmetic, rounding, etc., it is very likely that your output will not perfectly match ours. **It is okay if your image has non-zero numbers of pixel differences from our expected output**, so long as the images look essentially the same to the naked eye when you switch between them.

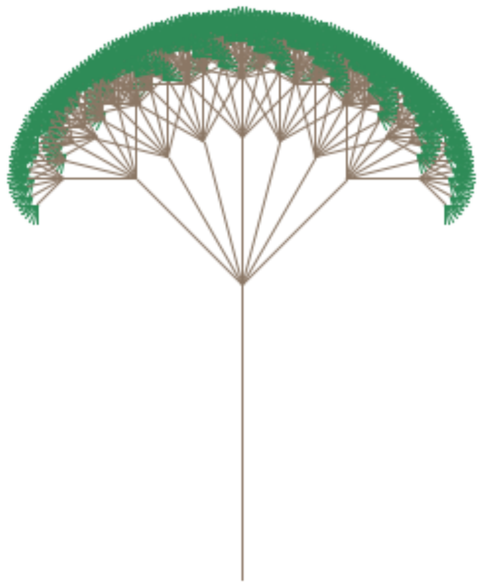
- sierpinski at x=10, y=30, size=300, order=1
- sierpinski at x=10, y=30, size=300, order=2
- sierpinski at x=10, y=30, size=300, order=3
- sierpinski at x=10, y=30, size=300, order=4
- sierpinski at x=10, y=30, size=300, order=5
- sierpinski at x=10, y=30, size=300, order=6

Recursive Tree ***(Optional Extension)***

For this problem, write a **recursive** function that draws a tree fractal image as specified here. Note that your solution is **allowed to use loops** if they are useful to help remove redundancy, but that your overall approach to drawing nested levels of the figure must still be recursive. Do not use any data structures in your solution such as a **`Vector`**, **`Map`**, arrays, etc.

Our tree fractal contains a trunk that is drawn from the bottom center of the applicable area (*x*, *y*) through (*x* + *size*, y + *size*). The trunk extends straight up through a distance that is exactly half as many pixels as *size*.

The drawing below is a tree of order 5. Sitting on top of its trunk are seven trees of order 4, each with a base trunk length half as long as the prior order tree's trunk. Each of the order-4 trees is topped off with seven order-3 trees, which are themselves comprised of seven order-2 trees, and so on.



Order-5 tree fractal

The parameters passed to your function are, in order: the window in which to draw the figure; the x/y position of the top left corner of the imaginary bounding box surrounding the area in which to draw the figure (more details on this just below); the side width/height ofthe figure (i.e., the overall size of the square boundary box; see below); and the number of levels, i.e., the order, of the figure.

```
void drawTree(GWindow& gw, double x, double y, double size, int order)
```

Some of these parameters are somewhat unintuitive. For example, the x/y coordinates passed are not the x/y coordinates of the tree trunk itself, but rather the coordinates of the bounding box area in which to draw the tree. The following diagram attempts to clarify the parameters visually:

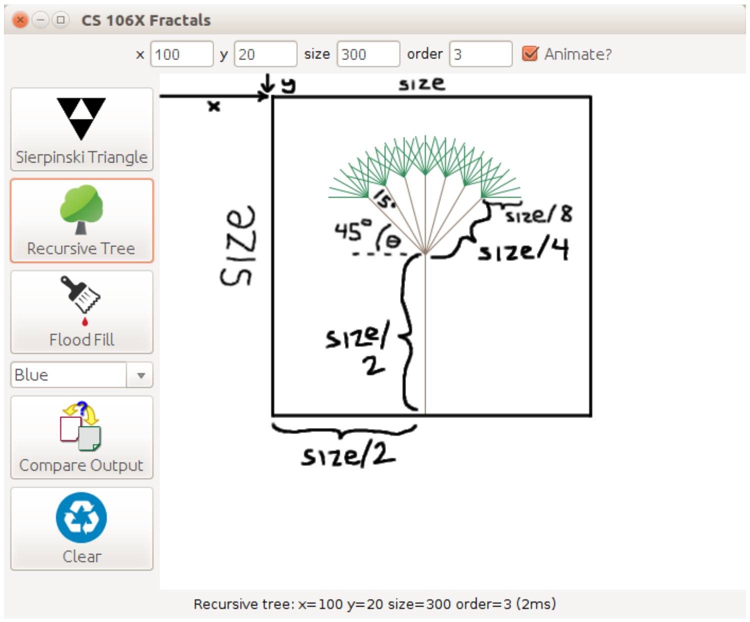


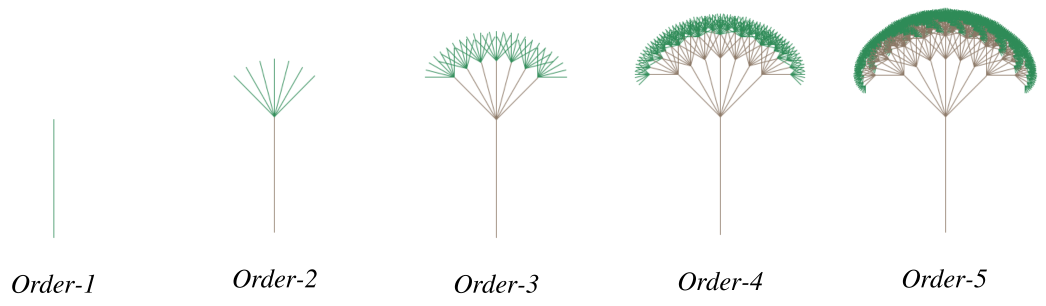
Diagram of **`drawTree(gw, 100, 20, 300, 3);`** call parameters

Lengths: As this drawing also shows, the trunks of each of the seven subtrees are exactly **half** as long of their parent branch's length in pixels.

Angles: The seven subtrees extend from the tip of the previous tree trunk at relative angles of ±45, ±30, ±15, and 0 degrees relative to their parent branch's angle. For example, if you look at the Order-1 figure (below), you can think of it as a vertical line drawn in an upward direction and facing upward, which is a polar angle of 90 degrees. In the Order-2 figure, the seven sub-branches extend at angles of 45, 60, 75, 90, 105, 120, and 135 degrees; etc.

Colors: Inner branches (i.e., branches drawn at order 2 and higher) should be drawn in the color **#8b7765** (r=139, g=119, b=101), and the leafy fringe branches of the tree (i.e., branches drawn at level 1) should be drawn in the color **#2e8b57** (r=46, g=139, b=87).

The images below show the progression of the first five orders of the fractal tree figure:



You should use the **`GWindow`** object's **`drawPolarLine()`** function to draw lines at various angles, and its **`setColor()`** function to change drawing colors, as seen in the Koch snowflake example from lecture.

Member	Description

`gw.drawPolarLine(x0, y0, r, theta);`
`gw.drawPolarLine(p0, r, theta);`
	draws a line the given starting point, of the given length ***r***, at the given angle in degrees ***theta*** relative to the origin

`gw.setColor(color);`	sets color used for future shapes to be drawn, either as a hex string such as `"#aa00ff"` or an RGB integer such as `0xaa00ff`

As in the first problem, if the order passed is 0, your function should not draw anything. Similarly, if the x or y coordinates, the order, or the size passed is negative, your function should throw a string **exception**. Otherwise, you may assume that the window passed is large enough to draw the figure at the given position and size.

Expected output: As above, you can compare your graphical output against the image files below, which are already packed into the starter code and can be compared against by clicking the "compare output" icon in the provided GUI, shown here:

Here too, please note that due to minor differences in pixel arithmetic, rounding, etc., it is very likely that your output will not perfectly match ours. **It is again fine if your image has non-zero numbers of pixel differences from our expected output**, so long as the images look essentially the same to the naked eye when you switch between them.

- tree at x=100, y=20, size=300, order=1
- tree at x=100, y=20, size=300, order=2
- tree at x=100, y=20, size=300, order=3
- tree at x=100, y=20, size=300, order=4
- tree at x=100, y=20, size=300, order=5
- tree at x=100, y=20, size=300, order=6

Mandelbrot Set

For this problem, you will write three functions that work together to generate the "Mandelbrot Set" in a graphical window, as described here. One of your functions will be recursive. Your solution may use loops to traverse a grid, but you must use recursion to determine if a particular point exists in the Mandelbrot Set.

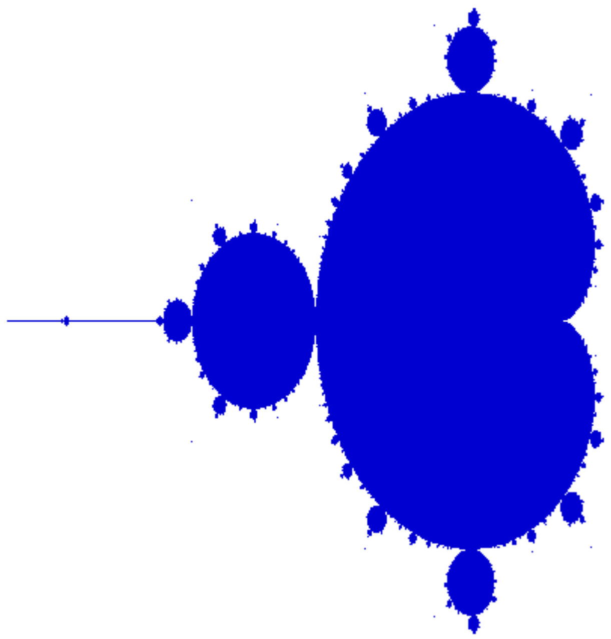
```
void mandelbrotSet(GWindow& gw, double minX, double incX,double minY, double incY, int maxIterations, int color)

int mandelbrotSetIterations(Complex c, int maxIterations)

int mandelbrotSetIterations(Complex z, Complex c, int remainingIterations)
```

The fractal below is a depiction of the "Mandelbrot Set," in tribute to Benoit Mandelbrot, a mathematician who investigated this phenomenon. The Mandelbrot Set is recursively defined as the set of complex numbers, *c*, for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z=0$. In other words, a complex number, c, is in the Mandelbrot Set if it does not grow without bounds when the recursive definition is applied. The complex number is plotted as the real component along the x-axis and the imaginary component along the y-axis.

For example, the complex number 0.2 + 0.5*i* is in the Mandelbrot Set, and therefore, the point (0.2, 0.5) would be plotted on an x-y plane. For this problem, you will map coordinates on an image (via a Grid) to the appropriate complex number plane (note that for this problem, we have abstracted away many details to make the coding easier).



Formally, the Mandelbrot set is the set of values of c in the complex plane that is bounded by the following recursive definition:

$$z_{n+1} = z_n^2 + c$$

$$z_0 = 0, \quad n \rightarrow \infty$$

In order to test if a number, c, is in the Mandelbrot Set, we recursively update z until either of the following conditions are met:

A. The absolute value of z becomes greater than 4 (it is diverging), at which point we determine that c is not in the Mandelbrot Set; or,

B. We exhaust a number of iterations, defined by a parameter (usually 200 is sufficient for the basic Mandelbrot Set, but this is a parameter you will be able to adjust), at which point we declare that c is in the Mandelbrot Set.

Because the Mandelbrot Set utilizes complex numbers, we have provided you with a **Complex** class that contains the following member functions:

Function	Description
`Complex(double a, double b)`	constructor that creates a complex number in the form a + b*i*
`cpx.abs()`	returns the absolute value of the number (a double)
`cpx.realPart()`	returns the real part of the complex number
`cpx.imagPart()`	returns the coefficient of the imaginary part of the complex number
`cpx1 + cpx2`	returns a complex number that is the addition of two complex numbers
`cpx1 * cpx2`	returns a complex number that is the product of two complex numbers

Implementation Details

One of the most interesting aspects of the Mandelbrot Set is that it has an infinitely large self-similarity. In other words, as you zoom into the Mandelbrot Set, you see similar features to an unzoomed Mandelbrot Set. The GUI for this assignment has been set up to allow zooming, based on where the user clicks on the current Mandelbrot Set in the window. The user may click on an area to zoom in, or shift-click on an area to zoom back out. Your **`mandelbrotSet()`** function will be passed in a number of parameters that you will use to determine which pixels in the zoomed window will be in the Mandelbrot Set, the number of iterations you will use in the recursion, and the color(s) of the resulting image. The details of the parameters are described below.

```
void mandelbrotSet(GWindow& gw, double minX, double incX, double minY, double
incY, int maxIterations, int color)
```

The **gw** parameter is the same as in the other two parts. In the starter code, we have already calculated the height and width of the window, created an image for the window, and created a grid based on that image. The grid is composed of individual pixel values (ints) that you can change based on whether or not a pixel is "in the Mandelbrot Set" or not. How to color the pixels is described below.

```
void mandelbrotSet(GWindow& gw, double minX, double incX, double minY, double
incY, int maxIterations, int color)
```

The **minX, incX, minY,** and **incY** parameters define the range of values you will inspect for inclusion in the Mandelbrot Set. **minX** corresponds to the left-most column in your grid, and it forms the real part of the first point you should check. **minY** corresponds to the top-most row in your grid, and it forms the coefficient of the imaginary part of the first point you should check. In other words, the first point you check will be the complex number minX + minY*i*, which you could create as follows:

`Complex coord = Complex(minX, minY)`

If you pass this value into your **`mandelbrotSetIterations()`** function, that function will return a number of iterations needed to determine whether or not the coodinate is unbounded. If the function returns maxIterations, we consider the coordinate to be in the Mandelbrot Set. If it returns a number less than maxIterations, then the coordinate is not in the Mandelbrot Set. The reason we do not simply return a **bool** is because we can use the number of iterations to provide a range of pretty colors, based on the palette (see the section on color below).

The **incX** and **incY** parameters have already been scaled to the size of the GWindow, and they provide the increment amount you should use as you traverse the pixel grid. For example, the pixel at row=3, col=5 would have the following coordinate: **`Complex(minX + 5 * incX, minY + 3 * incY);`** If your **`mandelbrotSetIterations()`** function returns maxIterations for that coordinate, it will be considered in the Mandelbrot Set, and should be set to show on the screen in the color as a parameter (see below).

```
void mandelbrotSet(GWindow& gw, double minX, double incX, double minY, double
incY, int maxIterations, int color)
```

You should use the **maxIterations** parameter in your **`mandelbrotSetIterations()`** function as one of the base cases for your recursion. If you have recursed **maxIterations** number of times and the absolute value of the coordinate remains bounded (less than 4), you should return maxIterations. In the GUI, the "size" value determines the maxIterations, and if you do not enter this value in the GUI, it defaults to 200 iterations. Note that the number of iterations places an O(n^2) complexity on your calculation, so it can be slow. However, the more you zoom, the more iterations you will need to determine whether a point is in the Set, and the more precise your results will be.

```
void mandelbrotSet(GWindow& gw, double minX, double incX, double minY, double
incY, int maxIterations, int color)
```

The **`color`** parameter is used to determine the color of your picture. If the color that is passed in is non-zero, you can simply use that number in your grid to show that a coordinate is in the Mandelbrot Set. In other words, if your **`mandelbrotSetIterations()`** function returns **`maxIterations`** for a coordinate, then you should set the pixel you are working on to **`color`**.

If, however, the color is zero, you should use the palette of colors, based on the result from your **`mandelbrotSetIterations()`** function. The calculation is a simple one:

`pixels[r][c] = palette[numIterations % palette.size()];`

```
int mandelbrotSetIterations(Complex c, int maxIterations)

int mandelbrotSetIterations(Complex z, Complex c, int remainingIterations)
```

These two functions have the same name (they are "overloaded"), but they have different parameters. You should plan on having your **`mandelbrotSet()`** function call the first function, which in turn calls the second function to perform the recursion. In this case, we call the second function a "helper" function, because it performs the recursion and in this case is used to pass along the *z* parameter (which, by the recursive definition, starts at zero). The helper function performs the recursion and returns the number of iterations back to the first function, which in turn passes the result back to the original **`mandelbrotSet()`** function. The **`mandelbrotSet()`** function uses this information to color the grid pixel (or not), based on the result.

Sample expected outputs:

- Basic Mandelbrot Set in blue, maxIterations: 200
- Basic Mandelbrot Set in palette, maxIterations: 200
- Basic Mandelbrot Set in red, maxIterations: 200, clicked on (390,60)
- Basic Mandelbrot Set in purple, maxIterations: 200, clicked on (327,367) twice
- Basic Mandelbrot Set in purple, maxIterations: 500, clicked on (285,195) three times

Style Details

As in other assignments, you should follow our **Style Guide** for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homework 1 and 2 specs, such as those detailing good problem decomposition, parameters, using proper C++ idioms, and commenting. The following are additional points of emphasis and style constraints specific to this problem.

Recursion: Part of your grade will come from appropriately using recursion to implement your algorithm, as we have described. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary. Avoid "arm's length" recursion, which is where the true base case is not found and unnecessary code or logic is inserted into the recursive case. **Redundancy** in recursive code is another major grading focus; avoid repeated logic as much as possible. As we mentioned above, it is fine (and sometimes necessary) to use "helper" functions to assist you in implementing the recursive algorithms for any part of the assignment.

Variables: While not new to this assignment, we want to stress that you should not make any global or static variables (unless they are constants declared with the `const` keyword). Do not use global variables as a way of getting around proper recursion and parameter-passing on this assignment.

Collections: Do not use any collections on any graphical algorithms in this part of the assignment.

Frequently Asked Questions (FAQ)

For each assignment problem, we frequently receive some common questions; you can find the answers to many of those questions by clicking here: **Fractals FAQ**.

Possible Extra Features

Here are some ideas for extra features that you could add to your program for a very small amount of extra credit:

- **Sierpinski colors:** Make your Sierpinski triangle draw different levels in different colors.
- **Mandelbrot Set Gradient:** The Wikipedia page on the Mandelbrot Set discusses ways to make a smooth gradient for the images, as opposed to using a palette. Implement a smooth gradient in your code.
- **Mandelbrot Set iterations**: The number of iterations needed to get a clear picture depends on the zoom. In the GUI, you can manually set this value, but you might want to use the minX, incX, and minY, incY parameters to determine the number of iterations, rather than the maxIterations passed into the function.
- **(Advanced) Mandelbrot Parallelism** Every point in the Mandelbrot set is independent of every other point, and thus this part of the assignment can be parallelized. You might do this using PThreads.
- **Add another fractal:** Add another fractal function representing a fractal image you like. You'll have to do some reconstructive surgery on the GUI to achieve this, and/or swap it in place of one of the existing fractal functions (make sure to turn in your non-extra-feature version first, so as not to lose your solution code).
- **Other:** If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can easily identify this code).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit two versions of your program file: a first one with the standard file name with no extra features added (or with all necessary features disabled or commented out), and a second one whose file name has the suffix **-extra.cpp** with the extra features enabled. Please distinguish them by explaining which is which in the comment header. Our submission system saves every submission you make, so if you make more than one we will be able to view them all; your previously submitted files will not be lost or overwritten.

Assignment 3B: Grammar Solver

Assignment by Marty Stepp and Victoria Kirst. Based on a problem by Stuart Reges of U of Washington.

JULY 12, 2017

Files and Links

Project Starter ZIP

(open **GrammarSolver.pro**)

Turn in:

grammarsolver.cpp

mygrammar.txt

Demo Jar

(note: ignore 20 Questions and Fractals flood fill. Also, the jar does not demonstrate Mandelbrot)

Problem Description

For this part of the assignment you will write a function to generate random sentences from what linguists, programmers, and logicians refer to as a "grammar."

Let's begin with some definitions. A *formal language* is a set of words and/or symbols along with a set of rules, collectively called the *syntax* of the language, that define how those symbols may be used together—programming languages like Java and C++ are formal languages, as are certain mathematical or logical notations. A *grammar,* in this context, is a way of describing the syntax and symbols of a formal language. All programming languages then have grammars; here is a link to a formal grammar for the Java language.

Your task for this problem will be to write a function that accepts a reference to an input file representing a language's grammar (in a format that we will explain below), along with a symbol to randomly generate, and the number of times to generate it. Your function should generate the given number of random expansions of the given symbol and return them as a `Vector` of strings.

```
Vector<string> grammarGenerate(istream& input, string symbol, int times)
```

Reading BNF Grammars

Many language grammars are described in a format called Backus-Naur Form (BNF), which is what we'll use in this assignment. In BNF, some symbols in a grammar are called *terminals* because they represent fundamental words of the language (i.e., symbols that cannot be transformed or reduced beyond their current state). A terminal in English might be "dog" or "run" or "Mariana". Other symbols of the grammar are called *non-terminals* and represent higher-level parts of the language syntax, such as a noun phrase or a sentence in English. Every non-terminal consists of one or more terminals; for example, the verb phrase "throw a ball" consists of three terminal words.

The BNF description of a language consists of a set of derivation *rules*, where each rule names a symbol and the legal transformations that can be performed between that symbol and other constructs in the language. You might think of these rules as providing translations between terminals and other elements, either terminals or non-terminals. For example, a BNF grammar for the English language might state that a sentence consists of a noun phrase and a verb phrase, and that a noun phrase can consist of an adjective followed by a noun or just a noun. Note that rules can be described recursively (i.e., in terms of themselves). For example, a noun phrase might consist of an adjective followed by another noun phrase, such as the phrase "big green tree" which consists of the adjective "big" followed by the noun phrase "green tree".

A BNF grammar is specified as an input file containing one or more rules, each on its own line, of the form:

non-terminal ::= rule|rule|rule|...|rule

A separator of **::=** (colon colon equals) divides the non-terminal from its transformation/expansion rules. There will be exactly one such **::=** separator per line. A **|** (pipe) separates each rule; if there is only one rule for a given non-terminal, there will be no pipe characters. Each rule will consist of one or more whitespace-separated tokens.

To illustrate how to read a BNF grammar, let's consider a very simple example. Suppose that we had the BNF input file below, which describes a language comprising a tiny subset of English. The non-terminal symbols **<s>**, **<pn>**, and **<iv>** are short for grammatical elements: sentences, proper nouns, and intransitive verbs (i.e., verbs that do not take an object). Note that in the following two examples, non-terminal elements are contained in angle brackets, while terminal elements (here, valid English words, such as "Lupe", "Camillo", "laughed", etc.) are not.

```
<s>::=<pn> <iv>

<pn>::=Lupe|Camillo

<iv>::=laughed|wept
```

To transform a symbol into a valid output string (that is, into a sentence described by this grammar's rules), we recursively expand the non-terminal symbols contained in its transformation rules until we have an output made up entirely of terminals—at that point our transformation is complete, and we're done. So in our trivial example of a grammar, if we are given the symbol **<s>**, which represents the non-terminal for "sentence", we examine the rules that describe how to expand the sentence non-terminal, and discover that it is comprised of a proper noun followed by an intransitive verb (**<s>::=<pn> <iv>**). We can begin by transforming the **<pn>** symbol, which we do by choosing at random from among the available transformations provided on the right-hand side of the **<pn>** rule (here, "Lupe" and "Camillo"). After this transformation, our sentence-in-progress might look like this:

```
        Camillo <iv>
```

At each step of our transformation, we perform a single transformation on one of the remaining non-terminal symbols. In this simple example, there is only one non-terminal left, **<iv>**. We accordingly transform that symbol by choosing randomly among the transformations provided for it, which could produce the following output:

```
Camillo laughed
```

Since there are no more non-terminal symbols remaining, our transformation is complete. Note that other valid sentences in this language (i.e., other valid transformations of the symbol **`<s>`**) include the sentences “Camillo wept”, “Lupe laughed”, and “Lupe wept”. Because we can choose at random from the transformations provided for each non-terminal symbol, all of these variations are possible and valid for this very simple language.

Now let’s look at an example of a more complex BNF input file, which describes a slightly larger subset of the English language. Here, the non-terminal symbols **`<np>`**, **`<dp>`**, and **`<tv>`** are short for the linguistic elements noun phrase, determinate article, and transitive verb.

```
<s>::=<np> <vp>

<np>::=<dp> <adjp> <n>|<pn>

<dp>::=the|a

<adjp>::=<adj>|<adj> <adjp>

<adj>::=big|fat|green|wonderful|faulty|subliminal|pretentious

<n>::=dog|cat|man|university|father|mother|child|television

<pn>::=John|Jane|Sally|Spot|Fred|Elmo

<vp>::=<tv> <np>|<iv>

<tv>::=hit|honored|kissed|helped

<iv>::=died|collapsed|laughed|wept
```

*Sample input file **sentence.txt***

Following the rules provided by this input file for expanding non-terminals, this grammar's language can represent sentences such as "The fat university laughed" and "Elmo kissed a green pretentious television". This grammar cannot describe the sentence "Stuart kissed the teacher", because the words "Stuart" and "teacher" are not part of the grammar. It also cannot describe "fat John collapsed Spot" because there are no rules that permit an adjective before the proper noun "John", nor an object after intransitive verb "collapsed".

Though the non-terminals in the previous two example languages are surrounded by **`< >`**, this is not required. By definition any token that **ever** appears on the left side of the **`::=`** of any line is considered a non-terminal, and any token that appears **only** on the right-hand side of **`::=`** in any line(s) is considered a terminal (so for the above example, as we have noted, **`<np>`** is a non-terminal, but “John” is a terminal). Do not assume that non-terminals will be surrounded by **`< >`** in your code. Each line's non-terminal will be a non-empty string that does not contain any whitespace.

You may assume that individual tokens in a rule are separated by a single space, and that there will be no outer whitespace surrounding a given rule or token.

Your **`grammarGenerate`** function will perform two major tasks:

1. **read an input file** with a grammar in Backus-Naur Form and turns its contents into a data structure; and
2. **randomly generate elements** of the grammar (**recursively**).

You may want to separate these steps into one or more **helper function**(s), each of which performs one step. It is important to separate the recursive part of the algorithm from the non-recursive part.

You are given a client program that handles the user interaction. The **`main`** function supplies you with an input file stream to read the BNF file. Your code must read in the file's contents and break each line into its symbols and rules so that it can generate random elements of the grammar as output. When you generate random elements, you store them into a **`Vector`** to be returned. The provided main program loops over the vector and prints the elements stored inside it.

Example Logs of Execution

Your program should exactly reproduce the format and general behavior demonstrated in these logs, although it may not exactly recreate these particular scenarios because of the randomness inherent in your code. (Don't forget to use the course File → Compare Output... feature in the console, or the course web site's Output Comparison Tool, to check output for various test cases.)

```
Symbol to generate (Enter to quit)? <np>
How many to generate? 5

 1: a wonderful father
 2: the faulty man
 3: Spot
 4: the subliminal university
 5: Sally
```

```
Symbol to generate (Enter to quit)? <s>
How many to generate? 7

 1: a green green big dog honored Fred
 2: the big child collapsed
 3: a subliminal dog kissed the subliminal television
 4: Fred died
 5: the pretentious fat subliminal mother wept
 6: Elmo honored a faulty television
 7: Elmo honored Elmo
```

```
Grammar file name? sentence.txt
Symbol to generate (Enter to quit)? <dp>
How many to generate? 3

 1: the
 2: the
 3: a
```

Expected output: Here are some additional expected output files to compare. As we noted, it's hard to match the expected output exactly because it contains randomness. But your function should return valid random results as per the grammar that was given to it. Your program's graphical Console window has a File → Compare Output feature for checking your output.

- test #1 (sentence.txt)
- test #2 (sentence2.txt)
- test #3 (expression.txt)

We have also written a **Grammar Verifier** web tool where you can paste in the randomly generated sentences and phrases from your program, and our page will do its best to validate that they are legal phrases that could have come from the original grammar file. This isn't a perfect test, but it is useful for finding some common types of mistakes and bugs.

Grammar Verifier Tool (click to show)

Now that you've seen an example of the program's behavior, let's dive into the implementation details of your algorithm.

Part 1: Reading the Input File

For this program you must store the contents of the grammar input file into a **`Map`**. As you know, maps keep track of key/value pairs, where each key is associated with a particular value. In our case, we want to store information about each non-terminal symbol, such that the non-terminal symbols become keys and their rules become values. Other than the **`Map`** requirement, you are allowed to use whatever constructs you need from the Stanford C++ libraries. You don't need to use recursion on this part of the algorithm; just loop over the file as needed to process its contents.

One problem you will have to deal with early in this program is breaking strings into various parts. To make it easier for you, the Stanford library's **`"strlib.h"`** library provides a **`stringSplit`** function that you can use on this assignment:

```
Vector<string> stringSplit(string s, string delimiter)
```

The string split function breaks a large string into a **`Vector`** of smaller string tokens; it accepts a delimiter string parameter and looks for that delimiter as the divider between tokens. Here is an example call to this function:

```
string s = "example;one;two;;three";

Vector<string> v = stringSplit(s, ";"); // {"example", "one", "two", "", "three"}
```

The parts of a rule will be separated by whitespace, but once you've split the rule by spaces, the spaces will be gone. If you want spaces between words when generating strings to return, you must concatenate those yourself. If you find that a string has unwanted spaces around its edges, you can remove them by calling the **`trim`** function, also from **`"strlib.h"`**:

```
string s2 = "  hello there  sir  ";

s2 = trim(s2);                          // "hello there  sir"
```

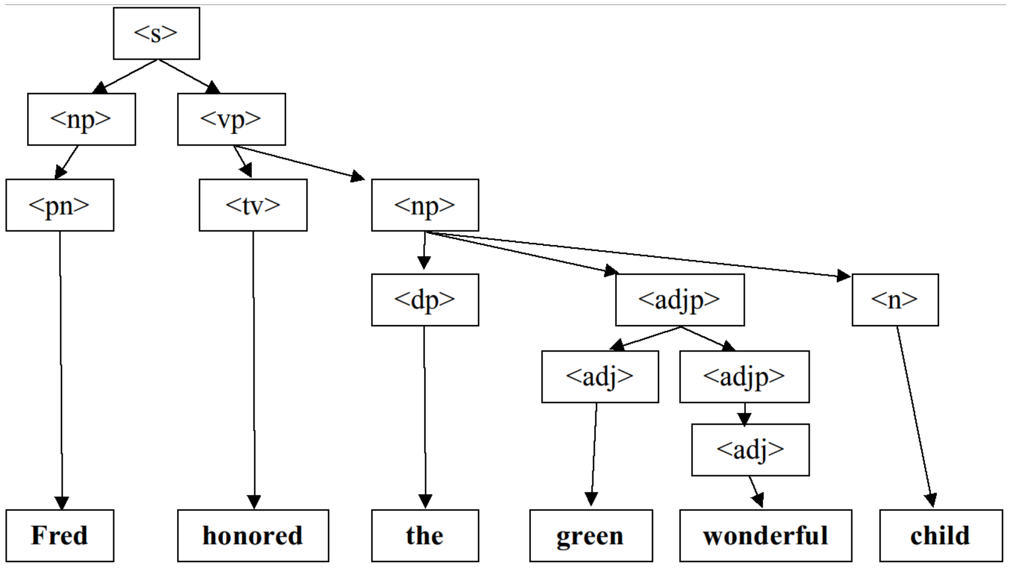
Part 2: Generating Random Expansions from the Grammar

As we mentioned, producing random grammar expansions is a two-step process. The first step, which we've just outlined, requires reading the input grammar file and turning it into an appropriate data structure (non-recursively). The second step requires your program to recursively walk that data structure to generate elements by successively expanding them.

The **recursive algorithm** your program will use to generate a random occurrence of a symbol *S* should have the following logic:

- If *S* is a terminal symbol, there is nothing to do; the result is the symbol itself.
- If *S* is a non-terminal symbol, choose a random expansion rule *R* for *S*, and for each of the symbols in that rule *R*, generate a random occurrence of that symbol.

For example, the example grammar given in the Problem Description above could be used to randomly generate an **`<s>`** non-terminal for the sentence, **`"Fred honored the green wonderful child"`**, as shown in the following diagram.



Random expansion from sentence.txt grammar for symbol <s>

If the string passed to your function is a non-terminal in your grammar, use the grammar's rules to *recursively* expand that symbol fully into a sequence of terminals. For example, using the grammar on the previous pages, a call of **`grammarGenerate("<np>")`** might potentially return the string, **`"the green wonderful child"`**.

Generating a non-terminal involves picking one of its rules at random and then generating each part of that rule, which might involve more non-terminals to recursively generate. For each of these you pick rules at random and generate each part, etc.

When you encounter a terminal, simply include it in your string. This becomes a base case. If the string passed to your function is not a non-terminal in your grammar, you should assume that it is a terminal symbol and simply return it. For example, a call of **`grammarGenerate("green")`** should just return **`"green"`** (without any spaces around it).

Special cases to handle: You should throw a string **exception** if the grammar contains more than one line for the same non-terminal. For example, if two lines both specified rules for symbol **`"<s>"`**, this would be illegal and should result in the exception being thrown. You should throw a string exception if the symbol parameter passed to your function is empty, **`""`**.

Testing: The provided input files and **`main`** may not test all of the above cases; it is your job to come up with tests for them.

Your function may assume that the input file exists, is non-empty, and is in a proper valid format. If the number of times passed is 0 or less, return an empty vector.

Implementation Details

The hardest part of this problem is the recursive generation, so make sure you have read the input file and built your data structure properly before tackling the recursive part.

Loops are *not* forbidden in this part of the assignment. In fact, you should definitely use loops for some parts of the code where they are appropriate. For example, the directory crawler example from class uses a for-each loop. This is perfectly acceptable; if you find that part of this problem is easily solved with a loop, please use one. In the directory crawler, the hard part was traversing all of the different sub-directories, and that's where we used recursion. For this program the hard part is following the grammar rules to generate all the parts of the grammar, so that is the place to use recursion. If your recursive function has a bug, try putting a **`cout`** statement at the start of your recursive function to print its parameter values; this will let you see the calls being made. As a finaly tip: look up the **`randomInteger`** function from **`"random.h"`** to help you make random choices between rules.

Style Details

(These are the same as in the Fractals problem.)

As in other assignments, you should follow our **Style Guide** for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homework 1 and 2 specs, such as those detailing good problem decomposition, parameters, using proper C++ idioms, and commenting. The following are additional points of emphasis and style constraints specific to this problem:

Recursion: Part of your grade will come from appropriately using recursion to implement your algorithm, as we have described. We will also grade on the elegance of your recursive algorithm; don't create special cases in your recursive code if they are not necessary. Avoid "arm's length" recursion, which is where the true base case is not found and unnecessary code or logic is inserted into the recursive case. **Redundancy** in recursive code is another major grading focus; avoid repeated logic as much as possible. As we mentioned above, it is fine (and sometimes necessary) to use "helper" functions to assist you in implementing the recursive algorithms for any part of the assignment.

Variables: While not new to this assignment, we want to stress that you should not make any global or static variables (unless they are constants declared with the **`const`** keyword). Do not use global variables as a way of getting around proper recursion and parameter-passing on this assignment.

Creative Component, **mygrammar.txt**

Along with your code, submit a file **mygrammar.txt** that contains a BNF grammar that can be used as input. For full credit, the file should be in valid BNF format, contain at least 5 non-terminals, and should be your own work (do more than just changing the terminal words in **sentence.txt**, for example). This is worth a small part of your grade.

Frequently Asked Questions (FAQ)

For each assignment problem, we frequently receive some common questions; you can find the answers to many of those questions by clicking here: **Grammar Solver FAQ**.

Possible Extra Features

Here are some ideas for extra features that you could add to your program:

- **Robust grammar solver:** Make your grammar solver able to handle files with excessive whitespace placed between tokens, such as:

```
"  <adjp> ::=   <adj> |    <adj>    <adjp>  "
```

- **Inverse grammar solver:** Write code that can verify whether a given expansion could possibly have come from a given grammar. For example, "The fat university laughed" could come from **sentence.txt**, but "Marty taught the curious students" could not. To answer such a question, you may want to generate all possible expansions of a given length that could come from a grammar, and see whether the given expansion is one of them. This is a good use of recursive backtracking and exhaustive searching. (Note that it’s tricky.)
- **Other:** If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can easily identify this code).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit two versions of your program file: a first one with the standard file name with no extra features added (or with all necessary features disabled or commented out), and a second one whose file name has the suffix **-extra.cpp** with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Our submission system saves every submission you make, so if you make more than one we will be able to view them all; your previously submitted files will not be lost or overwritten.

Fractals FAQ

Q: I'd like to write a helper function and declare a prototype for it, but am I allowed to modify the .h file to do so?
A: Function prototypes don't have to go in a .h file. Declare them near the top of your .cpp file and it will work fine.
Q: The spec says that I need to throw an exception in certain cases. But it looks like the provided `main` program never goes into those cases; it checks for that case before ever calling my function. Doesn't that mean the exception is useless? Do I actually have to do the exception part?
A: The exception is not useless, and yes you do need to do it. Understand that your code might be used by other client code other than that which was provided. You have to ensure that your function is robust no matter what parameter values are passed to it. Please follow the spec and throw the exceptions in the cases specified.
Q: Is the Sierpinski code supposed to be slow when I put in a large, value for N like 15?
A: Yes; keep in mind that the program is drawing roughly 315 triangles if you do that. That is a lot of triangles. We aren't going to test you on such a high order fractal.
Q: Are the Sierpinski triangles supposed to exactly overlay each other? Mine seem to be off by ~1 pixel sometimes.
A: The off-by-one aspect is because the triangle sizes are rounded to the nearest integer. You don't have to worry about that, as long as it is only off by a very small amount such as 1 pixel.
Q: Is the flood fill code supposed to paint so slowly?
A: Yes; the Stanford graphics library has poor graphics performance.
Q: When I run the sample solution, the flood fill pixels paint in a certain order. Does my function need to fill in that same order?
A: No; the order the pixels are filled does not matter, as long as you fill the right pixels.
Q: What does it mean if my program "unexpectedly finishes"?
A: It might mean that you have infinite recursion, which usually comes when you have not set a proper base case, or when your recursive calls don't pass the right parameters between each other. Try running your program in Debug Mode (F5) and viewing the call stack trace when it crashes to see what line it is on when it crashed. You could also try printing a message at the start of every call to make sure that you don't have infinite recursion.
If you are certain that you do not have infinite recursion, it is possible that there are just too many calls on the call stack because you are filling a very large area of the screen. If you tried to fill the whole screen with the same color, it might crash with a "stack overflow" even if your algorithm is correct. This would not be your fault.
Q: Can I use one of the STL containers from the C++ standard library?
A: No.
Q: I already know a lot of C/C++ from my previous programming experience. Can I use advanced features, such as pointers, on this assignment?
A: No; you should limit yourself to using the material that was taught in class so far.
Q: Can I add any other files to the program? Can I add some classes to the program?
A: No; you should limit yourself to the files and functions in the spec.
Q: My Mandelbrot output doesn’t exactly line up with the output in the program!
A: This may be because the window size is a bit different than when we generated our example output. Don’t worry too much about it — it should look very similar and be positioned close. The exact pixels do not need to perfectly overlap.

Grammar Solver FAQ

Q: I'd like to write a helper function and declare a prototype for it, but am I allowed to modify the .h file to do so?
A: Function prototypes don't have to go in a .h file. Declare them near the top of your .cpp file and it will work fine.
Q: What does it mean if my program "unexpectedly finishes"?
A: It probably means that you have infinite recursion, which usually comes when you have not set a proper base case, or when your recursive calls don't pass the right parameters between each other. Try running your program in Debug Mode (F5) and viewing the call stack trace when it crashes to see what line it is on when it crashed.
Q: The spec says that I need to throw an exception in certain cases. But it looks like the provided `main` program never goes into those cases; it checks for that case before ever calling my function. Doesn't that mean the exception is useless? Do I actually have to do the exception part?
A: The exception is not useless, and yes you do need to do it. Understand that your code might be used by other client code other than that which was provided. You have to ensure that your function is robust no matter what parameter values are passed to it. Please follow the spec and throw the exceptions in the cases specified.
Q: I am confused about grammars; what this assignment is about?
A: In terms of this assignment, a grammar is a set of rules that corresponds to a sybmol. For instance, the **adj** grammar relates to many different adjectives like purple, loud, sticky, and fake. To put it simply, one grammar has many rules. This relationship is also known as **non-terminal** to **terminal**. For this assignment, we want you to navigate through these grammars and create them. Notice that some grammars have grammars in their rule sets as well. Upon encountering this situation, remember that it is a grammar that needs to be generated as well.

Take a look at the decision tree in the assignment specifications, this should help give you a visualization of the problem at hand.
Q: What type of data should my `Map` store?
A: Unforutantely, this aspect of the assignment is up to you to figure out. Remember the functions you're going to have to call, along with the idea of a grammar and how it relates to its rules.
Q: I'm having trouble breaking apart the strings.
A: Use the provided **`stringSplit`** function. Print lots of debug messages to make sure that your split calls are doing what you think they are.
Q: How do I spot nonterminal symbols if they don't start/end with "<" and ">"?
A: The "<" and ">" symbols are not special cases you should be concerned with. Treat them as you would any other symbol, word, or character.
Q: Spaces are in the wrong places in my output. Why?
A: Try using other characters like "_", "~", or "!" instead of spaces to find out where there's extra whitespaces. Also remember that you can use the **`trim`** function to remove surrounding whitespace from strings.
Q: How do I debug my recursive function?
A: Try using print statements to find out which grammar symbols are being generated at certain points in your program. Try before and after your recursive step as well as in your base case.

Q: In my recursive case, why doesn't it join together the various pieces correctly?
A: Remember that setting a string = to another variable within a recursive step will not translate in all instances of that function. Be sure to actively add onto your `string` instead of reassigning it.

Q: My recursive function is really long and icky. How can I make it cleaner?
A: Remind yourself of the strategy in approaching recursive solutions. When are you stopping each recursive call? Make sure you're not making any needless or redundant checks inside of your code.

Q: When I run expression.txt, I get really long math expressions? Is that okay?
A: Yes, that's expected.

Q: What does it mean if my program "unexpectedly finishes"?
A: It might mean that you have infinite recursion, which usually comes when you have not set a proper base case, or when your recursive calls don't pass the right parameters between each other. Try running your program in Debug Mode (F5) and viewing the call stack trace when it crashes to see what line it is on when it crashed. You could also try printing a message at the start of every call to make sure that you don't have infinite recursion.
If you are certain that you do not have infinite recursion, it is possible that there are just too many calls on the call stack because you are filling a very large area of the screen. If you tried to fill the whole screen with the same color, it might crash with a "stack overflow" even if your algorithm is correct. This would not be your fault.

Q: Can I use one of the STL containers from the C++ standard library?
A: No.

Q: I already know a lot of C/C++ from my previous programming experience. Can I use advanced features, such as pointers, on this assignment?
A: No; you should limit yourself to using the material that was taught in class so far.

Q: Can I add any other files to the program? Can I add some classes to the program?
A: No; you should limit yourself to the files and functions in the spec.