Stanford University, CS 106B, Homework Assignment 3
Recursion Problems

Thanks to Eric Roberts, Julie Zelenski, Jerry Cain, Keith Schwarz, and Cynthia Lee for the first four problems, and to Stuart Reges for Grammar Solver.

This assignment focuses on **recursion**. The assignment has several parts. Parts A-D are individual functions intended to help you practice recursion. Part E (Grammar Solver) is larger and requires more work and explanation.

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. 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 **recursionproblems.cpp** file. (Don't modify the provided **.h** files to add your prototypes; put them in your own **.cpp** file.)

Files:

We will provide you with a **ZIP archive** containing a starter version of your project. Download this archive from the class web site and finish the code. Turn in only the following files:

- **recursionproblems.cpp**, the C++ code to solve all parts of the assignment
- **mygrammar.txt**, your own unique Part E input file representing a grammar to read as your program's input

The ZIP archive contains other files and libraries; you should not modify them. When grading your code, we will run your file with our own original versions of the support files, so your code must work with them.

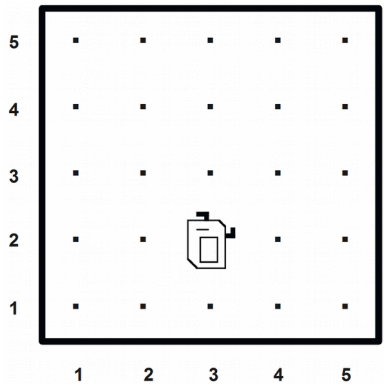
Optional Warm-up Problem: Karel Goes Home

(This is an optional problem that will not be graded. You may skip it if you like.)

For this optional "warm-up" problem, write a recursive function that returns the number of paths Karel could take back to the origin from the specified starting position, subject to the condition that Karel doesn't want to take any unnecessary steps and can therefore only move west or south (left or down in the diagram).

```
int countKarelPaths(int street, int avenue)
```

If you took CS 106A at Stanford, you know that Karel the Robot lives in a world composed of horizontal streets (rows) and vertical avenues (columns) laid out in a regular rectangular grid that looks like the diagram at right.



Suppose that Karel is sitting on the intersection of 2nd Street and 3rd Avenue as shown in the diagram and wants to get back to the origin at 1st Street and 1st Avenue. Even if Karel wants to avoid going out of the way, there are still several equally short paths. For example, in this diagram there are three possible routes:

1. Move left, then left, then down.
2. Move left, then down, then left.
3. Move down, then left, then left.

You should assume that Karel doesn't want to take any unnecessary steps and can therefore only move west or south (left or down in the diagram). If the street or avenue passed is less than 1, throw a string **exception**., such as:

```
throw "illegal street or avenue!";
```

Your solution should not use any loops or data structures; you must solve the problem using recursion.

Part A: String Manipulation

(For this part of the assignment, write the two functions `convertStringToInteger` *and* `isBalanced` *as described below.)*

`convertStringToInteger`: The Stanford C++ library includes a handy `stringToInteger` function that converts a `string` value such as `"1234"` to the equivalent `int` value of `1234`. But what if we did not have access to such a function? Write your own function that accepts a `string` parameter and returns an equivalent integer value.

```
int convertStringToInteger(string exp)
```

For example, the call of `convertStringToInteger("42693")` returns the `int` value `42693`. Your function should work for negative numbers; for example, the call of `convertStringToInteger("-574")` returns the `int` of `-574`.

If the string is invalid, throw a string exception. That is, the only characters that should be allowed in the string `exp` are numeric digits from `'0'` through `'9'`, possibly preceded by a single minus sign `'-'` at the start if the number is negative. A string that contains any other characters in it is considered invalid. You may assume that if the string contains valid characters, then it represents an integer that can be successfully stored in an `int`; that is, its value will not exceed 2^{31}. If the empty string is passed, your function should return `0`.

Your solution should not use any loops or data structures; solve the problem using recursion. Also, this is probably obvious, but you may not call the Stanford `stringToInteger` function in your solution, nor any other existing C++ library function that converts strings into integers (such as a `stringstream` or `atoi`); do the conversion yourself. Also, you may not use the `pow` function to raise an integer to an exponent; solve the problem in a different way.

Note: A common bug in this problem is confusion between `char` and `int` values. For example, the character `'0'` is not the same value as the integer `0`. You can convert between characters and numbers by addition and subtraction; for example, the character `'4'` minus the character `'0'` equals the integer `4`.

`isBalanced`: In our class discussion of stacks, we went through an example (similar to one from Chapter 5) that uses stacks to check whether the bracketing operators in an expression are properly nested. You can do a similar thing recursively. For this problem, write a recursive function that checks if a string has properly "balanced" characters:

```
bool isBalanced(string exp)
```

You may **assume that the string contains *only* parentheses, brackets, less/greater-than, and curly braces**. You should return `true` or `false` according to whether that expression has properly balanced operators. Notice that this is *not* exactly the same problem we solved with a `Stack` in lecture; in that problem we looked at strings containing other characters in addition to parentheses and brackets.

Your function should operate recursively by making use of the recursive insight that such a string is balanced if and only if one of the following two conditions holds:

1. The string is empty.
2. It contains `"()"`, `"[]"`, `"<>"`, or `"{}"` as a substring *and* is still balanced if you remove that substring.

For example, you can show that the string `"[(){}]"` is balanced by the following recursive chain of reasoning:

`isBalanced("[()<{}>]")` is `true` because
  `isBalanced("[<{}>]")` is `true` because
    `isBalanced("[<>]")` is `true` because
      `isBalanced("[]")` is `true` because
        `isBalanced("")` is `true` because the argument is empty.

You must verify exactly the definition of balance described above. The above definition implies that a string such as `"([)]"` or `")("` is not balanced, because it does not meet condition 1 or 2 above.

You may take advantage of various member functions of `string`s such as `length`, `find`, `substr`, `replace`, and `erase`. But your solution should not use any loops or data structures; you must solve the problem using recursion. Your solution should avoid **redundancy**; if you are repeating similar code, extract it into a helper function.

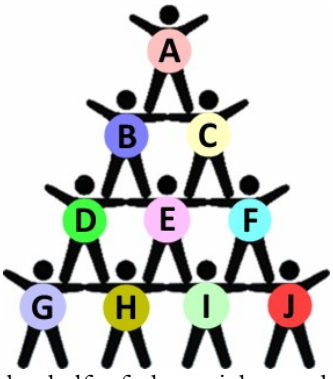
Part B: Human Pyramid

(problem by Cynthia Lee and Keith Schwarz)

For this problem, write a recursive function to compute the weight supported by a person in a "human pyramid":

```
double weightOnKnees(int row, int col, Vector<Vector<double> >& weights)
```

A *human pyramid* is a formation of people where participants make a horizontal row along the ground, and then additional rows of people climb on top of their backs. Each row contains 1 person fewer than the row below it. The top row has a single person in it. The image at right depicts a human pyramid with four rows.



Your task is to write a function to compute the weight being supported by the knees of a given person in the human pyramid. We will define the weight on a given person *P* 's knees recursively to be *P* 's own weight, plus half of the weight on the knees of each of the two people directly above *P*. For example, in the pyramid figure at right, the weight on the knees of person I below is I's own weight, plus half of the weight on the knees of persons E and F.

The weight on the knees of persons E and F can be computed recursively using the same process; for example, the weight on the knees of person E is E's own weight, plus half of the weight on the knees of person B, plus half of the weight on the knees of person C. If a given person does not have two people directly above them, any "blank" or "missing" persons should be ignored. For example, the weight on the knees of person F in the figure at right is F's own weight, plus half of the weight on the knees of person C. No rounding occurs during any of these recursive calculations.

To represent the pyramid we will use a a 2-dimensional vector of vectors, where each person's own weight in kilograms is stored as a real number. So for example, `weights[0][0]` refers to the weight of the person at the top of the pyramid, and `weights[weights.size() - 1][0]` refers to the weight of the bottom-left person in the pyramid. The table below illustrates which person's weight from the above-right figure would be stored in which index of the vector. You can think of it as a left-aligned version of the human pyramid figure.

```
    col   0  1  2  3
row
 0      {{A},
 1       {B, C},
 2       {D, E, F},
 3       {G, H, I, J}}          vector of weights for the above pyramid
```

Our provided code will create the nested vector of weights and pass it to your function. **You may assume** that the vector passed to your function is valid and matches the structure described above, such that the pyramid will always be fully filled in with values in the proper indexes. That is, if there are *n* people on the bottom row, then there are *n*-1 people on the next row up, and *n*-2 people on the next row above that, and so on. Use *exactly* the signature shown above. Do not modify the vector passed in. If the row/column passed is outside the bounds of the vector, **return 0**.

Our starter code provides the overall program and a loop to prompt for the pyramid's size. Here is a sample output:

```
How many people are on the bottom row? 4
Each person's own weight:
51.18
55.90 131.25
69.05 133.66 132.82
53.43 139.61 134.06 121.63

Weight on each person's knees:
51.18
81.49 156.84
109.80 252.82 211.24
108.32 320.92 366.09 227.25
```

Your solution should not use any loops or additional data structures; you must solve the problem using recursion.

Part C: Sierpinski Triangle

(a variation of book exercise 8.18)

For this problem, write a recursive function that draws the Sierpinski triangle fractal image.

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

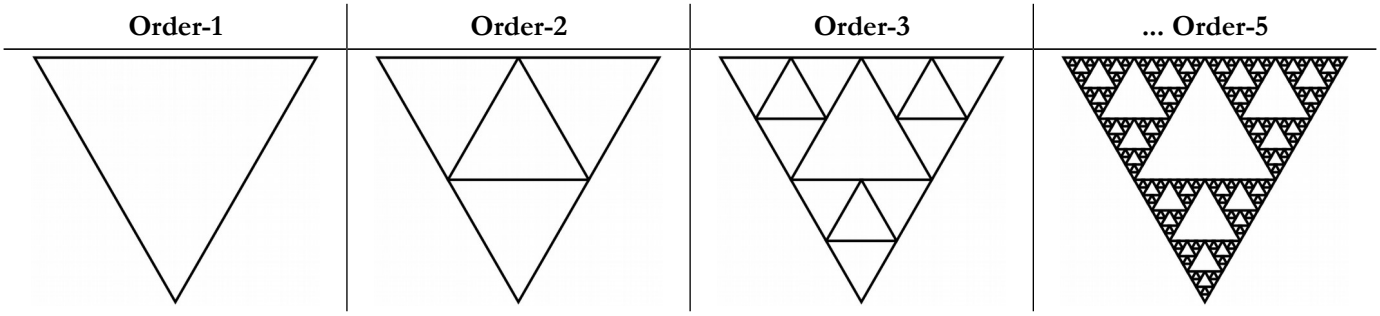
Your function should draw a black outlined Sierpinski triangle when passed a reference to a graphical window, the *x*/*y* coordinates of the top/left of the triangle, the length of each side of the triangle, and the order of the figure to draw (such as `1` for Order-1, etc.). The provided files already contain a `main` function that constructs the window and prompts the user to type an order, then passes the relevant information to your function. The rest is up to you.

If the order passed is 0, your function should not draw anything. If the x, y, order, or 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.

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.

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 original. Those three triangles are placed in the corners of the larger triangle. Take a look at the Order-2 Sierpinski triangle below to get the idea.

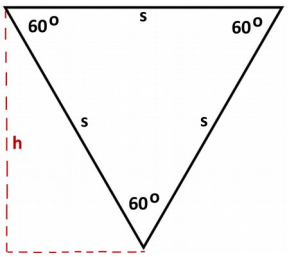
The upward-pointing triangle in the middle of the Order-2 figure is not drawn explicitly, but is instead formed by the sides of the other three triangles. That area, moreover, is not recursively subdivided and will remain unchanged at every order of the fractal decomposition. Thus, the Order-3 Sierpinski Triangle has the same open area in the middle.



We have not really learned much about the `GWindow` class, but you do not need to know much about it to solve this problem. The only member function you will need from the `GWindow` is its `drawLine` function:

`gw.drawLine(x1, y1, x2, y2);`	draws a line from point (*x*1, *y*1) to point (*x*2, *y*2)

Note: You may find yourself needing to compute the height of a given triangle so you can pass the right 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 at right 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.



- http://en.wikipedia.org/wiki/Equilateral_triangle

Your solution should not use any loops or data structures; you must use recursion.

A particular style of solution we want you to avoid is the "**pair of functions**" solution, where you write one function to draw "downward-pointing" triangles and another to draw "upward-pointing" triangles, and each calls the other in an alternating fashion. That is a poor solution that 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 drawing a line again (or part of a line again) that was already drawn, which is unnecessary and inefficient.

Part D: Flood Fill

(a variation of book exercise 9.4)

For this problem, write a recursive function to perform a "flood fill" on a graphical window as described below.

```
int floodFill(GBufferedImage& image, int x, int y, int color)
```

Most computer drawing programs make it possible to fill a region with a solid color. Typically you do this by selecting the "fill" tool and selecting a color, then clicking the mouse somewhere in your drawing. When you click, the paint color spreads outward from the point you clicked to every contiguous part of the image that is the same color as the pixel where you clicked. For example, if you select the Fill tool and choose Red as your color, then click on a purple part of the image, the Fill operation will cause that part of the image, along with any purple part that is touching it, to become red. The screenshots below provide an illustration:

Think of the image as being composed of a 2-D grid of tiny square dots called *pixels* (picture elements). When the user clicks, whatever color that pixel was, it should be replaced by the fill color outward in all four directions: up, down, left, and right. For example, if the initial pixel color is purple, change it to red, then explore one pixel up, down, left, and right from there, looking for more purple pixels and changing them to red. If your exploration leads you to a pixel that is a different color (such as yellow rather than purple in our example), stop exploring in that direction.

Drawing pixels: The Sierpinski Triangle problem asks you to interact with a `GWindow` object, but that class is not built for accessing individual pixels. So we will use a different object called `GBufferedImage` to color pixels here. You can use the following member functions of the buffered image object to get and set the color of individual pixels.

`image.getRGB(x, y)`	returns the color of the given pixel as an integer
`image.setRGB(x, y, color)`	sets the color of the given pixel to the given color as an integer
`image.getWidth(), image.getHeight()`	returns the width or height of the image in pixels
`image.inBounds(x, y)`	returns `true` if pixel (*x*, *y*) is within the bounds of the image

Your code should take care not to access any pixel outside of the bounds of the graphical image, (0, 0) through (width-1, height-1). If you try to get/set a pixel color that is out of bounds, your program will crash.

`int` color: The colors of pixels are returned as `int` values. The exact values that map to various colors don't especially matter to your code, though you can see what color maps to what integer value by looking at the **recursionmain.cpp** file. If you want to print a color out on the console for debugging, it is easier to view it in hexadecimal (base-16) mode. To do that, you'd issue a special '`hex`' command to `cout` to make it print an integer in hexadecimal rather than decimal (base-10), such as the following:

```
cout << hex << color << endl;
```

Optimization: For efficiency, your function is required to implement a particular optimization. If the caller passes the same fill color as the current pixel's existing color, do not color any pixels and immediately return `0`. For example, if the user has the Fill color set to Blue and clicks a blue area, your function should immediately return `0`. You can test whether your function is implementing this correctly by watching the **console's output** while clicking.

Return value: Your function must return the total number of pixels that changed color. For example, if your code ends up filling a 50x30 rectangular region, your function would return 1500. If no pixels change color, return `0`.

If the x or y value passed is negative or is outside the bounds of the given image, throw a string **exception**. You may assume that the color passed is a valid RGB color that can be drawn on the screen.

Your solution should not use any loops or data structures; you must use recursion.

On some machines, filling a very large area of pixels can crash your program because of too many nested function calls ("stack overflow"). This is out of your control. You may ignore this issue, so long as your algorithm is correct.

Part E: Grammar Solver

This part is a larger problem that requires more explanation.
Our expectation is that this part will take you the longest to complete and will provide the greatest challenge.

For this part you will write a function for generating random sentences from a grammar.

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

that accepts a reference to an input file representing a language grammar, 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 `string`s. A more thorough description of the task follows.

A **formal language** is a set of words and/or symbols along with a set of rules, collectively called the *syntax* of the language, defining how those symbols may be used together. A **grammar** is a way of describing the syntax and symbols of a formal language. Many language grammars are described in a format called Backus-Naur Form (*BNF*).

Some symbols in a grammar are called **terminals** because they represent fundamental words of the language. A terminal in English might be "boy" or "run" or "Jessica". Other symbols of the grammar are called *non-terminals* and represent high-level parts of the language syntax, such as a noun phrase or a sentence. 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. 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. Rules can be described *recursively* (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 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. The following is a valid example BNF input file describing a small subset of the English language. Non-terminal names such as `<s>`, `<np>` and `<tv>` are short for linguistic elements such as sentences, noun phrases, and transitive verbs.

Sample input file `sentence.txt`

```
<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
```

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 language 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. 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.

In this task you will write a function named `grammarGenerate` to 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 *(use **recursion** to implement this part)*

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

You are given a client program that does the user interaction. The `main` 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` which is returned. The provided main program loops over the vector and prints the elements stored inside it.

Your program should exactly reproduce the format and general behavior demonstrated in this log, although you may not exactly recreate this scenario because of the randomness that your code performs. (Don't forget to use the course web site's Output Comparison Tool to check output for various test cases.)

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

```
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
```

Part 1: Reading the Input File

For this program **you must store the contents of the grammar 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. So 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** provides a `stringSplit` function that you can use on this assignment:

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

The `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:

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

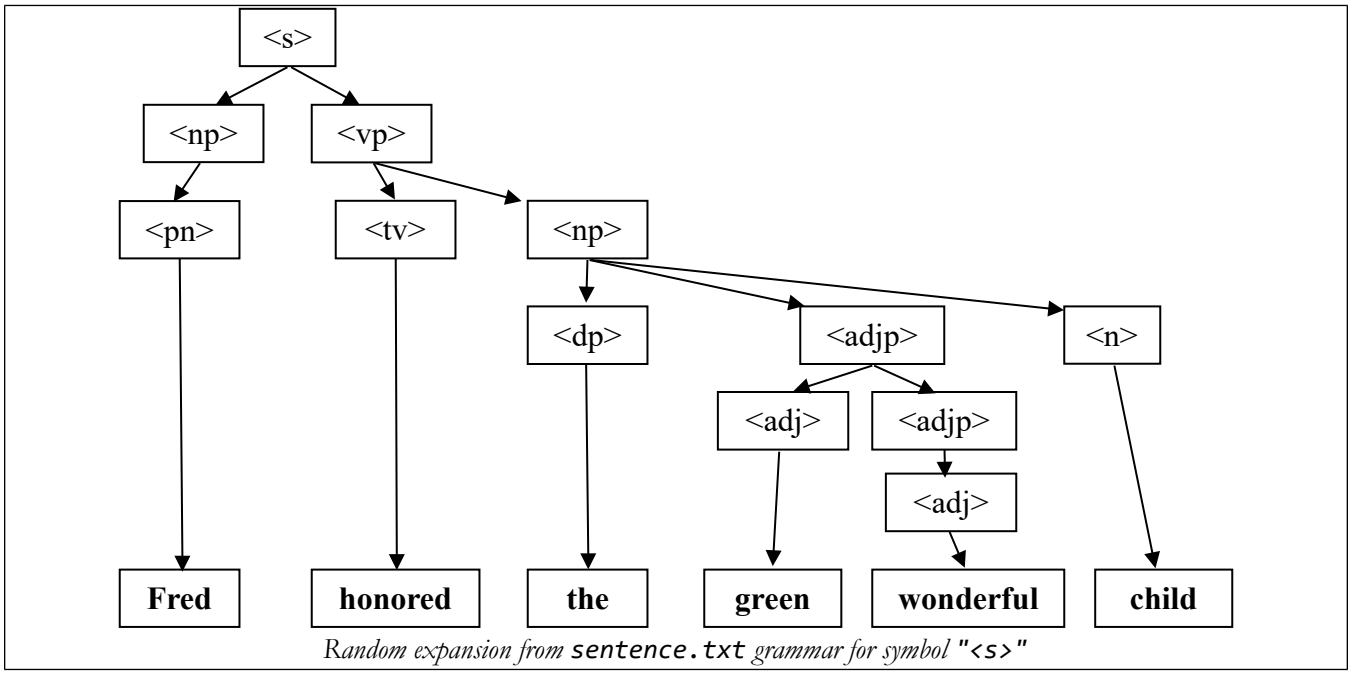
Part 2: Generating Random Expansions from the Grammar

As mentioned previously, producing random grammar expansions is a two-step process. The first step involves reading the input grammar file and turning it into an appropriate data structure (non-recursively). The second step involves *recursively* walking that data structure to generate elements by successively expanding them.

Generate elements of a grammar with a **recursive algorithm**. To generate a random occurrence of a symbol *S*:

- 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*.
  For each of the symbols in the rule *R*, generate a random occurrence of that symbol.

For example, the grammar on the previous page could be used to randomly generate a `<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, `""`.

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.

Creative Aspect (mygrammar.txt):

Along with your program, 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.

Development Strategy and Hints:

The hardest part is the recursive generation, so make sure you have read the input file and built your data structure properly before tackling the recursive part. The directory crawler program from lecture is a good guide.

Loops and data structures are forbidden in the other parts of this assignment, but not in the grammar part. In fact, you should definitely use loops for some parts of the code where they are appropriate. 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 print statement at the start of your recursive function that prints its parameter values, to see the calls being made.

Look up the `randomInteger` function from **random.h** to help you make random choices between rules.

Style Guidelines and Grading:

Items mentioned in the "Implementation and Grading" from the previous assignment(s) specs also apply here. Please refer to those documents as needed. Note the instructions in the previous assignments about procedural decomposition, variables, types, parameters, value vs. reference, and commenting. Don't forget to **cite any sources** you used in your comments. Refer to the course **Style Guide** for a more thorough discussion of good coding style.

Part of your grade will come from appropriately utilizing **recursion** to implement your algorithm as described previously. 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. **Redundancy** is another major grading focus; avoid repeated logic as much as possible. As mentioned previously, it is fine (sometimes necessary) to use "helper" functions to assist you in implementing the recursive algorithms for any part of the assignment.

As for **commenting**, place a descriptive comment header on each file you submit. Next to your function prototypes, place detailed comment headers next to every member explaining its purpose, parameters, what it returns, any exceptions it throws, assumptions it makes, etc. You should also place inline comments as needed on any complex code inside the function bodies.

Please remember to follow the **Honor Code** on this assignment. Submit your own work; do not look at others' solutions. Cite sources. Do not give out your solution; do not place a solution on a public web site or forum.

Possible Extra Features:

- **String to integer bases:** Make `convertStringToInteger` accept an optional base rather than assuming base-10. For example, passing string `"0x3f9"` and base `16` would return 3*16*16 + 15*16 + 9 = `1017`.
- **Sierpinski colors:** Make your Sierpinski triangle draw different levels in different colors.
- **isBalanced non-braces:** Make your `isBalanced` function work for strings that contain characters other than parentheses and braces. For example, `isBalanced("a bc[()d<{e}>]f")` would return `true`.
- **Memoized weightOnKnees:** A naive implementation of `weightOnKnees` will end up calculating the weight value for some people in the pyramid many times over. If you like, you can reduce the runtime of your recursive algorithm using **memoization**. Modify the signature of your recursive function so that `weights` will be `Vector<Vector<Person>>&` where `Person` is a `struct` for storing the person's individual weight and the memoization value of the weight on that person. Remember to change the function prototype.
- **Add colors to flood fill:** The existing client program has a small number of fixed colors that it uses. Go reverse-engineer it and add more colors of your own. Or make it possible to choose any color you want!
- **Flood fill gradient:** Make your flood fill algorithm slightly "fade" the color as it spreads further from the original click point. You can do this by slightly decreasing the brightness in R/G/B of pixels based on their distance from the original click point. This makes it harder to tell which squares to color, so stay within a nearby color range so that you can back-calculate what original color the faded pixels must have come from.
- **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.) *(This is very tricky.)*
- **Non-recursive versions:** Though the whole point of this assignment is to practice recursion, it can be very interesting to try writing the same algorithm in both a recursive and non-recursive fashion. As an extra feature, try to write all of the functions from this assignment without using any recursion. For some, like `convertStringToInteger`, it may be as straightforward as using a loop. For others, like `floodFill` or `grammarGenerate`, you may find that the non-recursive version is trickier than you thought. You might need to keep auxiliary data structures such as a `Stack` or `Queue` of points to fill, and so on. Note that doing this extra feature does *not* substitute for solving the problems in the normal recursive way; you *must* do the recursive versions to get credit on this assignment.
- **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 look at their code easily).

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 extra 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 files**: a first one where the file is named **recursionproblems.cpp** without any extra features added (or with all necessary features disabled or commented out), and a second one named **recursionproblems-extra.cpp** with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Please distinguish them in by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previous files will not be lost or overwritten.