Stanford University, CS 106B, Autumn 2013
Homework Assignment 4: Recursion

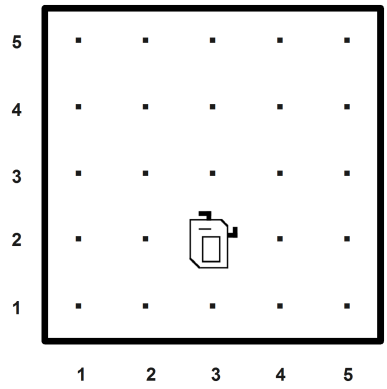
due Wednesday, November 6, 2013, 2:00pm

Thanks to Eric Roberts, Julie Zelenski, and Jerry Cain for the first three problems, and to Stuart Reges for Grammar Solver.

This assignment focuses on recursion. The assignment has four parts. The first three are small problems intended to help you "warm up" with recursion. The fourth part is a bit larger and requires more work and explanation.

Part A: Karel Goes Home (turn in RecursionWarmups.cpp)

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.

For this problem, write a recursive function:

```
int countPaths(int street, int avenue)
```

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). Do not use any loops or data structures; you must solve the problem using recursion.

If the street or avenue passed is less than 1, throw a string exception.

Part B: Balancing Parentheses (turn in RecursionWarmups.cpp)

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 the same thing recursively. For this problem, write a recursive function:

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

that accepts a string consisting only of parentheses, brackets, and curly braces and returns `true` or `false` according to whether that expression has properly balanced operators. Your function should operate recursively by making use of the recursive insight that such a string is balanced if and only one of the following conditions holds:

1. The string is empty.
2. The string contains `"()"`, `"[]"`, or `"{}"` as a substring and *is 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 the argument is empty.

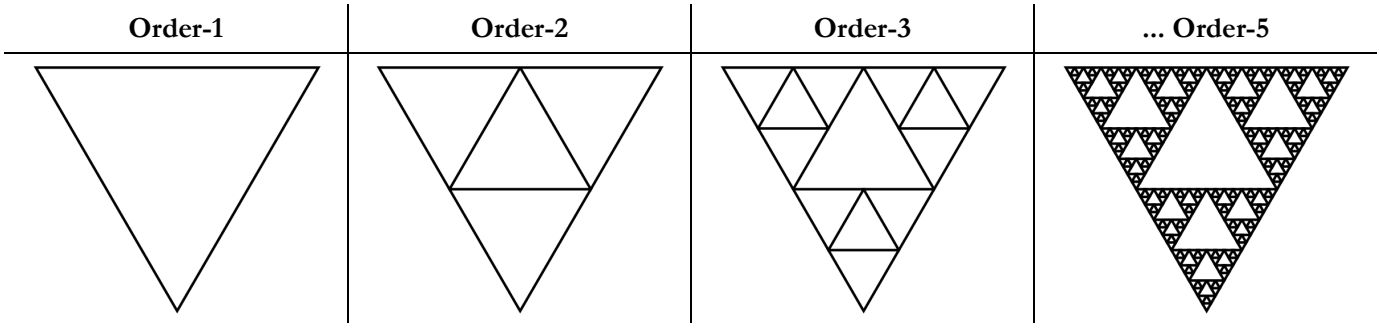
You may take advantage of various useful member functions of `string`s such as `length`, `find`, `substr`, `replace`, and `erase`. But your solution should not use any loops or data structures; solve the problem using recursion.

Part C: Sierpinski Triangle (turn in RecursionWarmups.cpp)

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.



For this problem, write a recursive function:

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

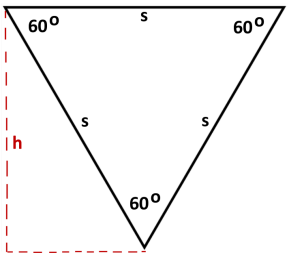
that draws a black outlined Sierpinski triangle when passed a reference to a graphical window, the *x*/*y* coordinates of the top/left of the triangle, the length of each side of the triangle, and the order of the figure to draw (such as `1` for Order-1, etc.). The provided skeleton of the program already contains 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 or less, your function should draw nothing.

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)

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 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 skills.



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

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

Part D: Grammar Solver (turn in GrammarSolver.h/.cpp and mygrammar.txt)

Unlike the previous three problems, this one is a larger program that requires more explanation. Our expectation is that this part will take you the longest to complete and will provide the greatest challenge.

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 can be 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 the English language 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.

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
```

The language described by this grammar 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. The grammar 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.

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.

Program Description:

In this part you will complete a program that reads an input file with a grammar in Backus-Naur Form and allows the user to randomly generate elements of the grammar. Use **recursion** to implement the core of your algorithm.

You are given a client program `grammarmain.cpp` that does the file processing and user interaction. Write a class called `GrammarSolver` that manipulates a grammar. 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.

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 and shuffling that your code performs.

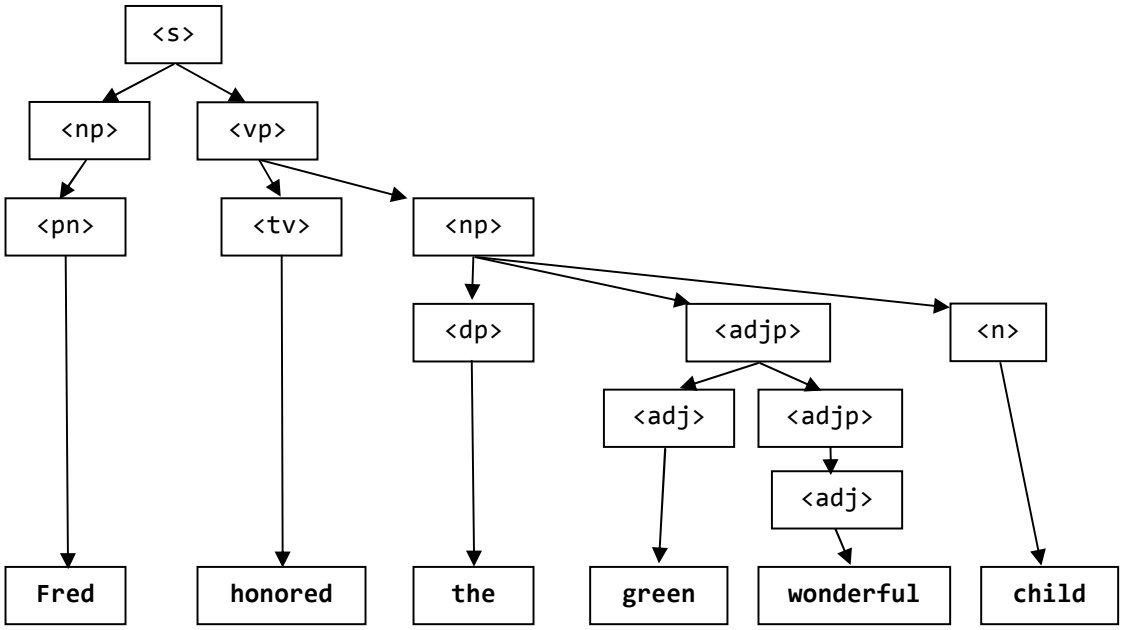
```
Welcome to the CS 106B random sentence generator!
This program recursively generates random sentences
based on a grammar in Backus-Naur Format (BNF).

Grammar file name (Enter for sentence.txt)? sentence.txt

Available symbols to generate are:
<adj> <adjp> <dp> <iv> <n> <np> <pn> <s> <tv> <vp>
What do you want to generate (Enter to quit)? <dp>
How many to generate? 3

the
the
a

Available symbols to generate are:
<adj> <adjp> <dp> <iv> <n> <np> <pn> <s> <tv> <vp>
What do you want to generate (Enter to quit)? <np>
How many to generate? 5

a wonderful father
the faulty man
Spot
the subliminal university
Sally

Available symbols to generate are:
<adj> <adjp> <dp> <iv> <n> <np> <pn> <s> <tv> <vp>
What do you want to generate (Enter to quit)? <s>
How many to generate? 10

a pretentious dog hit Elmo
a green green big dog honored Fred
the big child collapsed
a subliminal dog kissed the subliminal television
Sally laughed
Fred wept
Fred died
the pretentious fat subliminal mother wept
Elmo honored a faulty television
Elmo honored Elmo

Available symbols to generate are:
<adj> <adjp> <dp> <iv> <n> <np> <pn> <s> <tv> <vp>
What do you want to generate (Enter to quit)?
Exiting.
```

*(Don't forget to use the course web site's **Output Comparison Tool** to check your output for various test cases.)*

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 diagram on the next page:



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.

Required Public Members:

Your `GrammarSolver` class must have the following public members. See the provided `.h` file for more details.

`GrammarSolver(`***`input`***`)`	In this constructor you should initialize a new grammar solver over the given BNF grammar input file's data, where each rule appears as one line of text as shown in the file on the previous page. Your constructor should break apart the rules and store them into a `Map` so that you can later look up parts of the grammar efficiently. You may assume that the input file exists, is non-empty, and is in a proper valid format. 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.
`contains(`***`symbol`***`)`	In this member function you should return `true` if the given symbol is a *non-terminal* in the grammar and `false` otherwise. For example, when using the grammar described previously, you would return `true` for a call of `contains("<s>")` and `false` for a call of `contains("<foo>")` or `contains("green")` (`"green"` is a terminal in the language). You should throw a string **exception** if the string has a length of 0.
`getSymbols(`***`set`***`&)`	In this member function you must fill the given `Set`, passed by reference, with all non-terminal symbols of your grammar (the keys of your map). For example, when using the previous grammar, `getSymbols()` would fill the set with the ten elements `{"<adj>", "<adjp>", "<dp>", "<iv>", "<n>", "<np>", "<pn>", "<s>", "<tv>", "<vp>"}`.
`generate(`***`symbol`***`)`	In this function you should use the grammar to generate a random occurrence of the given symbol and return it as a `string`. If the string passed 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 `generate("<np>")` might potentially return the string, `"the green wonderful child"`. If the string passed 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 `generate("green")` should return `"green"`. (Note that there is *not* a space before/after `"green"`.) You should throw a string **exception** if the string has a length of 0.

Development Strategy and Hints:

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.

One problem you will have to deal with early in this program is breaking strings into various parts. To make it easier for you, we are providing a file **split.h/cpp** that contains a global `split` function that you can use on this assignment:

```
Vector<string> split(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 = split(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 s4 = "  hello there  sir  ";
s4 = trim(s4);                              // "hello there  sir"
```

The hardest function is `generate`, so write it last. The directory crawler program from lecture is a good guide. In that program, the recursive function has a for-each loop. This is perfectly acceptable; if you find that part of this problem is easily solved with a loop, 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 `generate` 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. Your class may have other member functions besides those specified, but any you add should be `private`. It is fine to use "helper" functions to assist you in implementing the recursive algorithms for any part of the assignment.

We have provided a partial version of your header `GrammarSolver.h` that declares the members. You will need to modify it: add comments, declare any private members, and apply `const` as appropriate to members and parameters.

Your code should have no **memory leaks**. Don't use pointers or `new` and you won't need to explicitly free anything.

As for **commenting**, place a descriptive comment header on each file you submit. In your .h files, place detailed comment headers next to every member explaining its purpose, parameters, what it returns, any exceptions it throws, assumptions it makes, etc. You don't need to put any comment headers on those same members in the corresponding **.cpp** file, though you should place inline comments as needed on any complex code in the 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.