we want to make a map?
start symbol is somewhere in your map
look up grammar(StartSym)
and this gives back a vector of alternates (options)
you replace every non terimal in these alternates (terminals can just be printed out)
you replace the non terminals by calling grammar(alternate)
you randomly pick one alternate if there are more than 1^

Project 2:

Random Sentence Generator

Due: 10/14

In class we talked about grammars, but we haven't really done a lot with them. For this project, we're going to make a tool to work with grammars. This tool is actually similar to parser generators like yacc and javacc. However, instead of doing anything productive, we're going to focus on writing stupid, but legal, sentences.

Part 1: Grammars

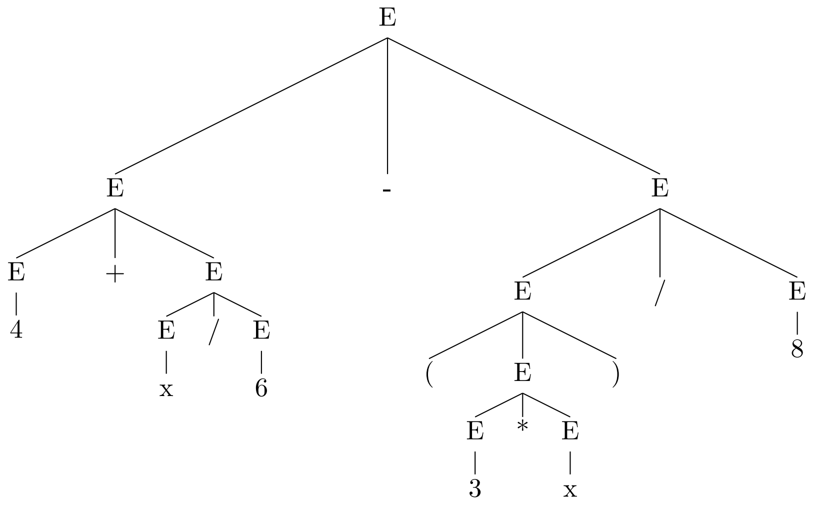
In the theory of computation a *language* is just a set of strings that are valid. We might think of the English language as the set of all valid English sentences. It is usually possible to define a language with a grammar. That is, every string that is in the language can be recognized by the grammar, and every string that is not in the language cannot be recognized by the grammar.

In class we talked about how we can use a grammar to recognize a sentence. We focused on examples like arithmetic expressions.

$$\begin{array}{lcl} E & \rightarrow & E+E \\ & | & E-E \\ & | & E*E \\ & | & E/E \\ & | & (E) \\ & | & Int \\ & | & Var \end{array}$$

This grammar can recognize any arithmetic expression made up of addition, subtraction, multiplication, division, parentheses, variables, and integers. We can see how the grammar can recognize an expression by constructing a parse tree.

$4 + (3 * x)/6 - x/8$
We start with E, and we keep expanding it until we get the whole expression.



We typically use grammars for their ability to recognize sentences in a language, but we can just as easily think of a grammar as generating a sentence in a language. We could instead start with E and generate our original sentence. You can read $A \Rightarrow B$ as A generates B.

E
$\Rightarrow E - E$
$\Rightarrow E + E - E$
$\Rightarrow 4 + E - E$
$\Rightarrow 4 + (E) - E$
$\Rightarrow 4 + (E * E) - E$
$\Rightarrow 4 + (3 * E) - E$
$\Rightarrow 4 + (3 * x) - E$
$\Rightarrow 4 + (3 * x) - E/E$
$\Rightarrow 4 + (3 * x) - x/E$
$\Rightarrow 4 + (3 * x) - x/8$

Since we were able to generate the sentence that we started with, that sentence is in our language. We're going to use this second way of looking at grammars for this project.

Part 2: Random Sentence Generation

For this project we want to read in a grammar file, and generate a random sentence in that language. But before we can read in a grammar, we need to know exactly what a grammar looks like. In other words, we need a grammar grammar.

Here's what our grammar grammar should look like.

- A grammar is a sequence of clauses
- Each clause begins with a non-terminal symbol followed by an arrow.
- Each clause has 1 or more alternates that are separated by a vertical bar.
- Each clause ends with a semicolon.
- Each alternate contains one or more symbols.
- A symbol is either a terminal, or a non-terminal.
- Each non-terminal symbol starts with an underscore character.

We can formally write out our grammar like this.

$$\begin{array}{lcl}
Grammar & \rightarrow & NonTerminal\ Clauses \\
Clauses & \rightarrow & Clause\ Clauses \\
 & | & Clause\ EOF \\
Clause & \rightarrow & NonTerminal\ \texttt{->}\ Alternates \\
Alternates & \rightarrow & Alternate\ \texttt{|}\ Alternates \\
 & | & Alternate\texttt{;} \\
Alternate & \rightarrow & Symbol\ Alternate \\
 & | & Symbol \\
Symbol & \rightarrow & Terminal \\
 & | & NonTerminal \\
NonTerminal & \rightarrow & \_\ .^* \\
Terminal & \rightarrow & [\hat{}\,\_\,]\ .^*
\end{array}$$

We can use the clue.g file as an example.

```
_Clue

_Clue -> It was _Person with the _Weapon in the _Place . ;

_Person -> Professor Plum  | Miss Scarlet | Mrs. Peacock
        |  Colonel Mustard | Mr. Green | Mrs. White ;

_Weapon -> knife | revolver | candlestick | wrench | lead pipe | rope ;

_Place -> library | conservatory | dining room | ballroom
       | dining room | billiards room | study | lounge | hall ;
```

There are 4 clauses. The Non-terminals are `_Clue`, `_Person`, `_Weapon`, and `_Place`. The `_Person` clause has 6 alternates, separated by `|`, and ending in a `;`. The very first Non-terminal in a grammar is the *start symbol*. We will use that to generate a random sentence in the Clue grammar.

We start with:

```
_Clue
```

Since there is only one clause, and one alternate, we replace it with:

```
It was _Person with the _Weapon in the _Place .
```

Now we can randomly replace `_Person` with one of the 6 alternates.

```
It was Mr. Green with the _Weapon in the _Place .
```

We can do the same for `_Weapon`.

```
It was Mr. Green with the candlestick in the _Place .
```

Finally we do the same for `_Place`.

```
It was Mr. Green with the candlestick in the hall .
```

And now, we've generated a random, legal sentence in the language of Clue accusations. For this project, your job is to write a C++ program that will read a grammar from standard input and write a random sentence in the language to standard output. You must turn in all .cpp and .h files that are necessary to run the program. If everything is in a single .cpp file, then you can submit that. Otherwise you should zip all files together.

Part 3: Lexing a Grammar

Everything from here on out is just a guide. If you have a different idea on how to implement the generator, then go for it! The important part is that it meets the specification.

This part is actually much easier than you'd expect. Every token in our grammar file is separated by white space. This works really well with C++'s I/O stream operators.

replacing escape characters

This is part of the project, but it's not worth many points, and you don't need it to complete the rest of the project. Get the rest of the project working and come back to this part. One area where `cin` doesn't work particularly well for is escape characters. `"\n"` as two separate characters. You should create a function that replaces any two characters representing an escape character with the correct escape character. These characters include [`'\b', '\n', '\r', '\t', '\a', '\f', '\v', '\'', '\"'`]. You might find the string stream class helpful here.

```
#include<sstream>
using namespace std;
...
stringstream ss;
ss << "You can add ";
ss << "strings to the string stream ";
ss << "to build up longer strings.";
ss.str() // then you can convert all the text to a single string.
```

You use the string stream like `cout` or `cin`. After you're done building up the string stream, you can use the `.str()` method to return a string. This is also useful for converting between numeric types and strings, but that's not necessary for this project.

Part 4: Parsing a Grammar

Now that we have all of the tokens in our grammar, we need to do something with them. Normally We'd parse them into an AST, but there's actually a data structure that will fit better here. Remember a grammar has two parts, the non-terminal symbol, and the alternate it can be replaced by. This is a perfect use for a Map data structure. Our map should take a non-terminal symbol, and give us back a list of alternate.

C++ Data Structures: map, unordered_map and vector

The `map`, and `unordered_map` classes are mostly the same, the main difference is in their implementation. `map` is a tree map, and `unordered_map` is a hash map.

They both have several methods, but the only ones you need right now are index assignment operator, and the index operator. We can create a map just like any other class in C++, but there are two type variables in the template.

```
map<string, int> chars;
```

This map counts the characters in a string. We can add to the map by using the index assignment operator. This is just like assigning to a value in an array. If the value doesn't exist, the map will create it, but if it does exist, the map will update the value.

```
chars["foo"] = 3;
chars["couch"] = 7;
chars["potato"]  = 6;
chars["couch"] = 5;
```

At this point `chars` has the keys `"foo"` with value 3, `"chars"` with value 5, and `"potato"` with value 6. If we want to look up the value associated with a key, we can use the index operator. The following example prints the number 4.

```
cout << chars["couch"] << endl;
```

If we're going to make a grammar, then we want a map of alternates. An alternate can just be a list of symbols. Here's my representation, but you might find one that works better for you.

```
using symbol = string;
using alternate = vector<symbol>;
using grammar = map<symbol, vector<alternate>;
```

I've used `vector` for my list here. A `vector` is just a growable array.
It behaves just like an array in C, but I can call the `push_back` method to add something to the end of the vector. The following example print out 5, 3, then 42.

```
vector<int> nums;
nums.push_back(3);
nums.push_back(5);
nums.push_back(42);
cout << nums[1] << " " << nums[0] << " " << nums.back() << endl;
```

Building Up the Grammar

Now that we have a data structure, we can fill it in with our tokens. Remember, the first token we read is the start symbol. It's also the first non-terminal in our grammar.

$$
\begin{array}{lcl}
\mathit{Grammar} & \rightarrow & \mathit{NonTerminal\ Clauses} \\
\mathit{Clauses} & \rightarrow & \mathit{Clause\ Clauses} \\
 & | & \mathit{Clause\ EOF} \\
\mathit{Clause} & \rightarrow & \mathit{NonTerminal}\ \texttt{->}\ \mathit{Alternates} \\
\mathit{Alternates} & \rightarrow & \mathit{Alternate}\ \texttt{|}\ \mathit{Alternates} \\
 & | & \mathit{Alternate}\texttt{;} \\
\mathit{Alternate} & \rightarrow & \mathit{Symbol\ Alternate} \\
 & | & \mathit{Symbol} \\
\mathit{Symbol} & \rightarrow & \mathit{Terminal} \\
 & | & \mathit{NonTerminal} \\
\mathit{NonTerminal} & \rightarrow & \_\ .^* \\
\mathit{Terminal} & \rightarrow & [\hat{}\ \_\ ]\ .^*
\end{array}
$$

In class we saw that we can translate a grammar into a parser by thinking of non-terminals as functions. Here an example of what the Alternates rule might look like.

```
readAlternates(tokens, grammar, nonterm)
    alternate = readAlternate(tokens)
    append alternate to garmmar[nonterm]
    if head(tokens) == ";"
        remove ; from tokens
        return
    if head(tokens) == "|"
        remove | from tokens
        readAlternates(tokens, grammar)
```

Part 5: Generation

Now that we've finally read in a grammar, we want to generate a random sentence. After the last part, the algorithm for this is surprisingly straightforward. We start with the start symbol. If our symbol is a non-terminal, then we can print it. Otherwise, we randomly choose an alternate from the grammar, loop through all the symbols in it, and print them out.

Let's run our Clue example from before. Our grammar should look like the following

```
start_symbol = "_Clue"

grammar["_Clue"] = {It was _Person with the _Weapon in the _Place .}
grammar["_Person"] = {Professor Plum,
                      Miss Scarlet,
                      Mrs. Peacock,
                      Colonel Mustard,
                      Mr. Green,
                      Mrs. White}
grammar["_Weapon"] = {knife, revolver, candlestick, wrench, lead pipe, rope}

grammar["_Place"] = {library, conservatory, dining room, ballroom,
                    dining room, billiards room, study, lounge, hall}
```

`_Clue`
This is replace with: `It was _Person with the _Weapon in the _Place .`
Print out `It was`
  `_Person` is a non-terminal, so randomly pick an alternate.
  We picked `Mr. Green`,
  so we print out `Mr.` and `Green`.
Print out `with the`
  `_Weapon` is a non-terminal, so randomly pick an alternate.
  We picked `candlestick`,
  so we print out `candlestick`
Print out `in the`
  `_Place` is a non-terminal, so randomly pick an alternate.
  We picked `hall`,
  so we print out `hall`
Print out `.`

You can start to see the recursive nature of this process.

random numbers

The last thing we need to deal with is, how do we get random numbers. C++ has a rich library for generating random numbers with good mathematical properties, which we are going to entirely ignore. We're going to use the C method and `<rand.h>` and `<time.h>`. At the start of our program we need to initialize the random number generator with

```
srand(time(NULL));
```

Then whenever we want a random integer between 0 and $n - 1$, we can use `rand() % n`.