

## Program 3(b) The Random Sentence Generator (RSG)

Due: Thursday, October 16, 10:00 p.m.

But wait! I can't possibly do this assignment in the 32 hours between the end of class and that due date!

Well, then, you better generate a really good request for an extension. How about:

**Tactic 1: Wear down the instructor's patience.** I need an extension because I used up all my paper and then I didn't know I was in this class and then my karma wasn't good last week and on top of that my dog ate my notes and as if that wasn't enough I had to finish my doctoral thesis this week and then I had to do laundry and on top of that my karma wasn't good last week and on top that I just didn't feel like working and then I thought I already graduated and as if that wasn't enough I lost my mind and on top of that all my pencils broke.

**Tactic 2: Plead innocence.** I need an extension because I forgot it would require work and then I didn't know I was in this class.

**Tactic 3: Honesty.** I need an extension because I just didn't feel like working

**Note: Real due date is Thursday, October 23, 10:00 p.m.**

In this assignment you will use your Dictionary class from Program 3(a) together with other data structures (from the STL) to make a C++ program that can generate such sentences for you. The Random Sentence Generator is a handy and marvelous piece of technology that creates random sentences from a pattern known as a *grammar*. (High Wizards, such as students of CS 301, call this particular form a *context-free grammar*.) A grammar is a template that describes the various combinations of words that can be used to form valid sentences. There are profoundly useful grammars available (from the course web site, stolen from Stanford) to generate extension requests, generic Star Trek plots, your average James Bond movie, "Dear John" letters, and more. You can even create your own grammar. Fun for the whole family!

### What is a grammar?

A grammar is just a set of rules for some language, be it English, the C++ programming language, or an invented language. In CS 301 they make a formal study; we just have fun in this course.

Here is a simple grammar:

```

The poem grammar.
{
<start>
The <object> <verb> tonight. ;
}
{
<object>
waves ;
big yellow flowers ;
slugs ;
}
{
<verb>
sigh <adverb> ;
portend like <object> ;
die <adverb> ;
}
{
<adverb>
warily ;
grumpily ;
}

```

Two possible poems generated by this grammar are, “The big yellow flowers sigh warily tonight.” and “The slugs portend like waves tonight.” Essentially, the strings in the angle brackets behave as variables that expand according to the rules in the grammar.

More precisely, each string in brackets is known as a “nonterminal.” A nonterminal is a placeholder that will expand into another sequence of words when generating a poem. In contrast, a “terminal” is a normal word that is not changed to anything else when expanding the grammar. The name “terminal” is supposed to make you think of a dead-end—no further expansion is possible from here.

We will write a grammar as a set of definitions. A definition of a grammar consists of a nonterminal and its set of “productions” (or “expansions”). We will terminate each production with a semi-colon (;) and we will not use semicolons anywhere else in our grammar files. There will always be at least one and sometimes many productions that are expansions for any given nonterminal. For instance, in the Poem grammar above, there are two productions for the nonterminal <adverb>, one to “warily” and one to “grumpily”. A production is just a sequence of terminals and/or nonterminals; that is, a sequence of words and/or nonterminals.

A production can be empty (i.e., just consist of the termination semi-colon), which makes it possible for a nonterminal to expand to nothing.

We will enclose each definition inside of curly braces (i.e., inside a pair { }).

Note that anything *not* inside a pair of curly braces is a comment and should be ignored

by your program. All of the components of the input file: braces, words, and semi-colons will be separated from each other by some sort of white space (spaces, tabs, newlines), so you will be able to use those as delimiters when reading the grammar in. And you can discard those white spaces, since they are not important. You may assume that there are no embedded spaces in the name of any non-terminal.

Once you have read in the grammar, you will be able to use it to produce random expansions from it. You will always begin with the specific non-terminal `<start>`. Every grammar will include this non-terminal (although it will not always be the first definition in the grammar). For a non-terminal, choose a random production from its set of productions. (I'll review random numbers at the end of this handout.) Take the words from the chosen production in sequence, (recursively) expanding any which are themselves non-terminals as you go. For example:

```
<start>
The <object> <verb> tonight.
The big yellow flowers <verb> tonight.
The big yellow flowers sigh <adverb> tonight.
The big yellow flowers sigh warily tonight.
```

Since we are choosing productions at random, doing the derivation a second time might produce a different result, and running the entire program again should also result in different patterns.

## Designing Your Approach

I *strongly* advise you to take time to sketch out your plans for the data structures and algorithms that you will use for this task *before* you start any coding. This assignment can actually be pretty easy and short if you make use of your Dictionary class and a some useful things such as string and vector from the STL and/or C++ in general.

You are *required* to use your Dictionary class to store the non-terminals of your grammar. The key would be the name of the non-terminal, and the definition/value would be the set of productions for that non-terminal.

Note that there is quite a hierarchy here: the value is a *set* of productions, and then each production is a sequence of terminals and nonterminals.

I suggest that you will find life easy if you store the set of productions for a non-terminal as a vector. The productions themselves will want to be either a vector or a list of strings. Think carefully about these choices ahead of time—make choices that make the rest of the assignment easy.

If it makes your life easier, you can store adjacent terminals (i.e., just plain words and/or punctuation marks) all in one string.

The broad stages of your algorithm will include the following.

## Parsing and storing the grammar file

You will start by reading the grammar file into your Dictionary. This reading of the grammar file will be a large chunk of the total work for this assignment. Your goal is to set things up so that the grammar is very easy to use in the rest of the program.

Be sure to test that your reading is working before going farther in your coding.

## Expanding the grammar

Once the grammar is loaded up, begin with the `<start>` production and expand it to generate a random sentence. Note that the algorithm to do this is naturally recursive.

The grammar will always contain a `<start>` non-terminal to begin the expansion. It will not necessarily be the first definition in the file, but it will always be defined eventually. Your code can assume that the grammar files are syntactically correct (i.e., have a start definition, have the correct punctuation and format as described earlier, a non-terminal has only one definition, don't have some sort of endless recursive cycle in the expansion, etc.)

The names of non-terminals should be considered case-insensitively; for example `<NOUN>` matches both `<Noun>` and `<noun>`.

## Printing the result

It is probably easiest for you to print the result as you go—printing any string that isn't a non-terminal when your expansion comes to that string.

You will be writing your output to `cout`.

I am providing you with one very useful utility function called `PrintWithWrap` that is on the course web page. When you call it, it will print really long strings with appropriate newlines put in in place of spaces so that the line doesn't overflow. By default, repeated calls keep printing in the same place as the last call. To override this behavior, either pass `PrintWithWrap` an explicit newline, or give use its second argument, a defaulted `bool` that tells whether to reset at the left margin.

Note: you may get very strange results if you mix direct printing to `cout` with calls to `PrintWithWrap`.

In general, you should probably leave a space after punctuation characters such as a comma or a period. (The open quotation character is an exception.) We will be happy if your output looks reasonably nice; it does not have to be perfect.

## Some useful odds and ends from C and C++

### Random numbers

To generate random numbers, we will rely on functions from the two C libraries `<stdlib.h>` and `<time.h>`.

The C++ function `rand()` returns a (pseudo-)random number in the range from 0 to `RAND_MAX`. It should be called after the function `srand` is called to set the seed for the random numbers, at least if you want truly random results. Sometimes, for instance, for debugging purposes, you might want the same sequence of random numbers over and over again. In that case, you would set the random seed to some fixed number. This is all illustrated in the following short program.

```
#include <stdlib.h>
#include <iostream>
#include <time.h>
using namespace std;

// Very brief demo of random number generation
// Robert H. Sloan
// Fall term, 2002

int
main()
{
    srand (100);                // Initialize random # generator
    for (int i = 0; i < 5; i++)
        cout << rand () << endl;

    cout << "Now some different each time RANDOM random numbers" << endl;

    srand (time(NULL));        // Initialize random # generator
    for (int i = 0; i < 5; i++)
        cout << rand () << endl;

    return 0;
}
```

Note that the first 5 random numbers will be the same every time the program is executed; the second 5 will be different. (On the machine on my desktop, running Solaris 7, and using g++ version 2.95, the first 5 numbers are

```
12662
23392
22561
20718
6314
```

.) What those five numbers are will be different on different machines.

To pick a random number between 0 and  $n$ , you would use `rand()%n`.

## Classifying chars, changing case

From the C library `<ctype.h>` you may find the function `ispunct()` which tells whether a char is a punctuation char and `isspace()`, which tells whether a char is a white space char useful.

There are in fact a whole bunch of these functions on chars, others include `isalpha`, `isupper`, and `islower`.

Also in that library are the functions on chars `toupper()` and `tolower()`. Both functions take a char input and return a char.

For `toupper()` if the the input char is a lowercase letter, then the uppercase version of that char is returned; otherwise the char is returned unaltered. The obvious symmetric thing is done by `tolower()`.

## Actual specification

Your program should take one command-line argument, which is the name of the grammar file to read. Your program should create three random expansions from the grammar and then exit.

You will find 8 sample grammars for you to play with in `/web/instruct/i202/WWW/grammars`, and also on the course web page (once the TA or I have posted them).

## Acknowledgment

This assignment is a minor variation on the Stanford freshman computer science assignment presented by Julie Zelenski at the annual meeting of the ACM Special Interest Group on Computer Science Education. Listed by Julie Zelenski as originated by Mike Cleron.