Assignment 1c: Random Sentence Generator

Assignment originated by Mike Cleron, polished off by Nick and Julie

The Inspiration

In the past decade or so, computers have revolutionized student life. In addition to providing no end of entertainment and distractions, computers also have also facilitated all sorts of student work from English papers to calculus. One important area of student labor that has been painfully neglected is the task of filling up space in papers, Ph.D. dissertations, extension requests, etc. with important sounding and somewhat grammatically correct random sequences. An area which has been neglected—that is, until now.

Due: Wednesday, October 25th at 11:59 p.m.

The final part of the C assignment is to take your ADT work from the first two parts, put on your client hat, and use these modules to construct a non-trivial C program that fills in this gaping hole. The Random Sentence Generator is a handy and marvelous piece of technology to create random sentences from a pattern known as a *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 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! Let's show you the value of this practical and wonderful tool:

- Tactic #1: Wear down the TA's patience.

  I need an extension because I used up all my paper and then my dorm burned down and then I didn't know I was in this class and then I lost my mind 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 of that I just didn't feel like working and then I skied into a tree and then I got stuck in a blizzard at Tahoe and as if that wasn't enough I thought I already graduated and as if that wasn't enough I lost my mind and in addition I spent all weekend hung-over and then I had to go to the Winter Olympics this week 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.

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. If you go on to study computer science, you will learn much more about languages and grammars in a formal sense. For now, we will introduce to you a particular kind of grammar called a Context Free Grammar (**CFG**). Here is an example of 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 ;
}
```

According to this grammar, two possible poems are "The big yellow flowers sigh warily tonight" and "The slugs portend like waves tonight." Essentially, the strings in brackets (<>) are variables which expand according to the rules in the grammar.

More precisely, each string in brackets is known as a "non-terminal". A non-terminal is a placeholder that will expand to 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 conjure up the image that it is a dead-end— no further expansion is possible from here.

A definition consists of a non-terminal and its set of "productions" or "expansions", each of which is terminated by a semi-colon ';'. There will always be at least one and potentially several productions that are expansions for the non-terminal. A production is just a sequence of words, some of which may be non-terminals. A production can be empty (i.e. just consist of the terminating semi-colon) which makes it possible for a non-terminal to expand to nothing. The entire definition is enclosed in curly braces '{' '}'. The following definition of "<verb>" has three productions:

```
{
<verb>
   sigh <adverb> ;
   portend like <object> ;
   die <adverb> ;
}
```

Comments and other irrelevant text may be outside the curly braces and should be ignored (quickly advancing over the extraneous stuff outside the braces is a good use of the Scanner's skipping functions, I might add). All 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 parsing the grammar. And you can discard the white-space delimiter tokens since they are not important. No token will be larger than 128 characters long, so you have an upper bound on the buffer needed when reading a word; however, when you store the words, you should not use such an excessive amount of space, use only what's needed. In order to read the grammar files, you will find the Scanner routines from HW1a quite handy.[1]

Once you have read in the grammar, you will be able to produce random expansions from it. You begin with the single non-terminal <start>. For a non-terminal, consider its definition, which will contains a set of productions. Choose one of the productions at random. 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.                    — expand <start>
The big yellow flowers <verb> tonight.          — expand <object>
The big yellow flowers sigh <adverb> tonight.   — expand <verb>
The big yellow flowers sigh warily tonight.     — expand <adverb>
```

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

Before you start any coding, you should sketch out your plans for the data structures and algorithms you will use to solve the task at hand. We're not going to give you a lot of specific advice, we want you to consider the options and make good choices on your own. Take the time to make quality decisions—there are various paths to choose from, and you want to be sure to choose the options that make the job easier.

[1] In fact, re-read this last paragraph again to be sure you don't miss the hints about how to set up the scanner.

Parsing the grammar files

Once you have your attack outlined, start by reading the grammar file into your data structure leveraging as much of HW1a and b as you can. Parsing the file should come out fairly cleanly using the `Scanner`. Be sure to decompose the various steps in reading the file into small reasonable routines that you can develop and test in stages. You should do all of the parsing work during the file-reading phase— your goal is to put the grammar into a format that makes it very easy to traverse and print expansions later without doing any further manipulations on the grammar.

Storing The Grammar

You should definitely use your handy ADTs to store and manipulate the various components of the grammar. Store the terminals and non-terminals as strings allocated to appropriate size (i.e. do not store using a large fixed-size buffer). You will use the `DArray` and `HashTable` ADTs to organize the productions, definitions, and the outer collection of all the definitions. Remember that a hashtable is particularly good for doing quick lookups, so data you often need to search would best be organized in a hashtable. To help you choose good allocation sizes, the number of different non-terminals in a grammar is on the order of 10 to 20, and the number of words per production is often small, say just one or two, but sometimes can be as long as 20 or 30. Be sure to use good variable and field names to distinguish the various types of arrays and tables in your program—the compiler's type checking will not distinguish among them at all and vacuous names like `array`, `word`, `table` don't provide much context for the reader. You also will want to `#define` useful names for the punctuation characters used to delimit the grammar markers parts, it is much easier for the reader to interpret a name like `ProductionEndMarker` that looking for characters like `'}'` and `';'` within C code which itself heavily uses those punctuation characters.

Expanding the grammar

Once the grammar is loaded up, begin with the `<start>` production and expand it to generate a random sentence. Be sure to use your earlier work from HW1b to enable easy lookup and straightforward mapping over your collections. Note that the algorithm to traverse the data structure and print the terminals 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 above, a non-terminal has only one definition, don't have some sort of endless recursive cycle in the expansion, etc.). The one error condition you should catch reasonably (an `assert` is fine) is the case where a non-terminal is used but not defined. It is fine to catch this when expanding the grammar and encountering the undefined non-terminal rather than attempting to check the consistency of the entire grammar while reading it.

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

Printing The Result

When generating the output, you do not need to store the result in some intermediate data-structure— just print the terminals as you expand. Each terminal should be preceded by a space when printed except the terminals that begin with punctuation like periods, comma, dashes, etc. which look dumb with leading spaces.[2] In order to make your output neatly wrap in regular line breaks, we provide you with a `PrintWithWrap` utility function to assist with this. The function keeps track of the number of characters printed so far on the current line and starts a new line (breaking at a space character rather than in the middle of a word) as it gets close to the right margin. You need to be sure to do **all** your output printing through this function. If you accidentally intermingle direct uses of `printf`, you will get strange line-wrapping behavior.

Choosing The Grammar File

Your program should take one argument, which is the name of the grammar file to read. As with all UNIX programs, you can give a full or relative path as an argument; the relative path will save you typing. For example, to read from the `dump.g` grammar file which is the `grammars` subdirectory of the current directory:

```
% rsg grammars/dump.g
```

Your program should create three random expansions from the grammar and exit. Before exit, your program should free all its dynamically allocated memory.

Living in the Land of the ADT Client

The RSG makes good use of all the ADTs you constructed in part a and b, and if you really "grokked" the `void*` memory manipulations and how to use the `DArray` and `HashTable` in the search client, then putting them to work in this final piece is pretty straightforward. But even with fully working and thoroughly debugged ADTs, it still easy to make little slip-ups in using them as a client, so be careful and don't wait until the last minute to tackle this assignment.

As we recommended for the HW1b search client, force yourself to stay in the role of client. Use only the description in the `.h` files and don't think too much about what you know is going on behind the wall, since often that only serves to confuse you.

---

[2] You can use the ANSI `ispunct()` function to check if a character is a punctuaction mark. This rule about leading spaces is just a rough heuristic, because some punctuation (quotes for example) might look better with spaces. Don't stress about the minor details, we're looking for something simple that is right most of the time and it's okay if it is little off for some cases.

During the process of writing this program, try to evaluate how your generic ADTs help or hinder the construction of this program. Does the C language support or discourage this kind of modular construction? Do you think that you will have a need in the future for something like the `DArray` or `Hashtable` again? Would you look forward to using them again? Why or why not?

A few other suggestions

- Don't forget to leverage earlier work from your HW1b's `search.c` when you find yourself in need of a hash function.
- For random numbers, we recommend the `random` function since it has better randomizing behavior that the standard ANSI `rand` functtion. However, you are free to use either, as long as your program produces random results between runs. See the man pages for srandom / srand for seeding the generator, you can use the current time (man 2 time) as the seed. Revisit Chapter 8 of your 106A text if you don't recall the steps used to convert a random result into an integer within some range. (There is a Purify issue involving `random` you can ignore.)
- We will test your RSG with your version of the `Scanner`, `DArray`, and `Hashtable` ADTs. However while in development, you might find it helpful to work with our known good versions if you are suspicious of yours. In the `hw1c_sample` directory, there are compiled `.o`'s that you can use. See the `README` file in that directory for the instructions about how to use them instead of your versions.
- As always, we expect that your code compile cleanly— i.e. without any warnings— and that it run under Purify without finding any errors.

Grading

Because this assignment is somewhat less complex, it is worth about the same as HW1a, which is about half of HW1b. A few functionality points are reserved for rewarding correctly working ADTs, but most will be determined by your program's ability to parse and expand the grammar files correctly without any errors or memory troubles. The design and readability scores are based on just your RSG client code. For design, we will evaluate how effectively the ADTs are used to design an efficient and tidy storage solution for the grammars, the directness of your algorithms, the decomposition of your program into sensible sub-routines, and so on. The readability score reminds you that we consider it essential that you hand in a neat and readable program with helpful naming choices, consistent style conventions and an appropriate level of commenting. We request the printout in a compact but legible form.

Getting Started

The starting project is in the leland directory `/usr/class/cs107/assignments/hw1c`**.** This directory contains a skeleton `rsg.c` file, a makefile that builds the project, and subdirectory of grammar files (files named with the extension `".g"`). You want to make your own copy of the project directory and will need to add your previous `Scanner`, `DArray`, and `HashTable` files. You can also get the starting files via anonymous ftp to

`ftp.stanford.edu` or from the class Web site `http://cse.stanford.edu/classes/cs107/`.