

Please note that some of the resources used in this assignment require a Stanford Network Account and therefore may not be accessible.

CS107
Spring 2008

Handout 04
April 4, 2008

Assignment 1: Random Sentence Generator

Based on an old CS107 assignment written by Julie Zelenski. First three pages are taken verbatim from someone else's handout.

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 that's been neglected, that is, until now.

Due: Sunday, April 13th at 11:59 p.m.

The Random Sentence Generator is a handy and marvelous piece of technology to create random sentences from a structure known as 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 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 had to go to an alligator wrestling meet, and then, just when my mojo was getting back on its feet, I just didn't feel like working, and, well I'm a little embarrassed about this, but I had to practice for the Winter Olympics, and on top of that my roommate ate my disk, and right about then well, it's all a haze, and then my dorm burned down, and just then I had tons of midterms and tons of papers, and right about then I lost a lot of money on the four-square semi-finals, oh, and then I had recurring dreams about my notes, and just then I forgot how to write, and right about then my dog ate my dreams, and just then I had to practice for an intramural monster truck meet, oh, and then the bookstore was out of erasers, and on top of that my roommate ate my sense of purpose, and then get this, the programming language was inadequately abstract.
- **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 a set of rules for some language, be it English, C++, Scheme, or something you just invent for fun. ☺ If you continue 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 CFG:

```
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's something of a dead end, that no further expansion is possible.

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 for each non-terminal. A production is just a sequence of words, some of which themselves 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 evaporate into nothingness. 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. 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 that we're able to treat them as delimiters when parsing the grammar.

Once you have read in the grammar (and that part's easy, because I'm already providing a `readGrammar` function for you), you will be able to produce random expansions. You always begin with the single non-terminal `<start>`. For a non-terminal, consider its definition, which will contain a set of productions. Choose one of the productions at random. Take the words from the chosen production in sequence, (recursively) expanding any that 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, a second generation would probably produce a different sentence.

What To Do, What To Do

Without a doubt, the most difficult and time-consuming portion of Assignment 1 has very little to do with C++ coding. We expect you to spend a majority of your time getting used UNIX, **emacs**, and Makefiles. In past quarters, the first assignment has been a little more intense than this one, but this time I'm giving a smaller problem where much of the code has already been written for you. I've taken care of the not-so-sexy task of parsing the command line and reading in the specified grammar file to build a grammar object in the form of an STL **map** (which is similar to the CS106 **Map**, but just different enough that you'll need to read Handout 03 before you can make use of it.) Here's a high-level outline of how I'd approach the problem over the next seven days.

- Read the Unix Basics handout, which is being distributed today as Handout 05. Read just enough of it so that you know how to log into a UNIX workstation, create directories, list directory contents, and execute other programs. Wonder TA Ryan Park will dedicate all of today's and tomorrow's discussion section to

UNIX, but he'll focus primarily on the C++ development tools and less on the basics.

- Go to the Terman cluster, or **ssh** into one of their machines. The machines in the Terman cluster are known as the **Pods**, and they are the machines we'll be using to grade your work. (When you **ssh** into a machine, you **ssh** to **pod.stanford.edu**.) Log in, create a directory called **cs107**, descend into it, and copy over the Assignment 1 files. Here's a transcript of what I get when I do what I just told you to do:

```
jerry> mkdir cs107
jerry> cd cs107
jerry> ls
total 0
jerry> cp -r /usr/class/cs107/assignments/assn-1-rsg .
jerry> ls
total 2
drwxr-x--- 3 poohbear 37 2048 Sep 25 16:54 assn-1-rsg
jerry> cd assn-1-rsg/
jerry> pwd
/afs/ir.stanford.edu/users/p/o/poohbear/cs107/assn-1-rsg
jerry> ls
total 18
jerry> ls
total 798
-rw-r----- 1 poohbear operator 1482 2008-04-04 08:48 definition.cc
-rw-r----- 1 poohbear operator 2784 2008-04-04 08:48 definition.h
-rw----- 1 poohbear operator 851 2008-04-04 08:48 Makefile
-rw-r----- 1 poohbear operator 1543 2008-04-04 08:48 production.cc
-rw-r----- 1 poohbear operator 2854 2008-04-04 08:48 production.h
-rw-r----- 1 poohbear operator 959 2008-04-04 08:48 random.cc
-rw-r--r-- 1 poohbear operator 1074 2008-04-04 08:48 random.h
-rw-r--r-- 1 poohbear operator 1022 2008-04-04 08:48 README
-rw-r----- 1 poohbear operator 2867 2008-04-04 08:48 rsg.cc
-rwxr-xr-x 1 poohbear operator 395430 2008-04-04 08:48 rsg-sample-linux
-rwxr-xr-x 1 poohbear operator 400522 2008-04-04 08:48 rsg-sample-solaris
```

I create a directory called **cs107** with the **mkdir** (short for **make directory**) command. I descend into that directory using **cd** (short for **change directory**). I then copy over files from official CS107 space into my new little directory using **cp** (yes, for **copy**). The source of the copy is **/usr/class/cs107/assignments/assn-1-rsg**, the destination is the current working directory (which is what the dot/period always stands for), and the **-r** flag tells the **cp** command that the copy should be recursive, which means all files within the **assn-1-rsg** should be copied over. The **ls** (**list**) command lists all of the top level items within the current working directory, and the **pwd** (**present working directory**) posts the absolute path of where you are (yes, my username is **poohbear**—don't ask). In the final listing, you see evidence that all this may be working toward C++ coding after all.

Play with the sample RSG program a couple hundred times so you know what you're working toward. Here're a few sample runs:

```
jerry> ln -s /usr/class/cs107/assignments/assn-1-rsg-data
grammars
jerry> ./rsg-sample-linux grammars/excuse.g
Version #1: -----
    I need an extension because I lost a lot of money on the
four-square semi-finals, and I'm sure you've heard this
before, but I didn't know I was in this class, and if you can
believe it, well, it's all a haze, and just then it was just too
nice outside.

Version #2: -----
    I need an extension because I used up all my paper, and, well
I'm a little embarrassed about this, but I had to finish my
doctoral thesis, and just then I used up all my paper.

Version #3: -----
    I need an extension because all my pencils broke, and if you
can believe it, I had tons of midterms and mega papers.

jerry> ./rsg-sample-solaris grammars/poem.g
Version #1: -----
    The slugs die warily tonight.

Version #2: -----
    The slugs sigh grumpily tonight.

Version #3: -----
    The big yellow flowers portend like slugs tonight.
```

Notice you're invoking **rsg-sample-solaris** as the executable instead of other executables like **cp**, **ls**, or **pwd**. (The **./** tells UNIX to look in the current working directory for the executable called **rsg-sample-solaris**. Yes, you'd think it wouldn't need to be told, but for sophisticated reasons you need to be clear about where UNIX should be looking. It's possible to configure your environment so that you don't need the dot, but let's worry about that later.)

- Make it a point to attend next Tuesday's discussion section. We'll cover enough Unix so that you can tackle the coding and debugging process.
- Use **emacs** to read **production.h**, **definition.h**, and **rsg.cc**. I give you fully functional implementations of a **Production** class and a **Definition** class. The **Definition** class encapsulates a nonterminal pairing with a collection of **Productions**, where a **Production** itself models a sequence of items a nonterminal might expand to. The **rsg.cc** is a partial implementation that reads a flat-text grammar file into a **map<string, Definition>**. You'll want to compile and build the starter code to see what it does. You do this by typing **make**:

```

jerry> make
g++ -g -Wall -MM rsg.cc random.cc production.cc definition.cc > ...
g++ -g -Wall -c -o rsg.o rsg.cc
g++ -g -Wall -c -o random.o random.cc
g++ -g -Wall -c -o production.o production.cc
g++ -g -Wall -c -o definition.o definition.cc
g++ -o rsg rsg.o random.o production.o definition.o
jerry> ./rsg grammars/bond.g
The grammar file called "grammars/bond.g" contains 37 definitions.
jerry> ./rsg grammars/poem.g
The grammar file called "grammars/poem.g" contains 4 definitions.
jerry> ./rsg grammars/trek.g
The grammar file called "grammars/trek.g" contains 35 definitions.
jerry> ./rsg grammars/excuse.g
The grammar file called "grammars/excuse.g" contains 7 definitions.

```

The starter code loads the grammar for you, and then prints out the number of definitions it contains. This is fairly evident from the sample run above. Your task is to transform the `rsg.cc` to do more than that (and if you want to, you can change the other `.h` and `.cc` files if you want to, though I don't think you do.)

- Come up with an algorithm that, when seeded with "`<start>`", manages with the tightest and most clever of recursions to transform "`<start>`" into a randomly generated sequence of terminals. Once you've accumulated this sequence of terminals (I recommend an STL `vector`, which is different enough from the CS106 `vector` that you'll want to read about it in Handout 03), you can traverse the `vector` of terminals and print them one by one, using your big brain to figure out when to print a new line, when to include intervening spaces and when not to. (Don't stress over the new line and spacing business too much, though. It's hardly the point of the assignment.) Repeat the random sentence generation exactly two more times without reading the grammar in again. Ultimately try to replicate the functionality of the sample executable without sweating over the petty details. I'm much more interested in elegant code with a readable narrative than I am in the minutiae involved in getting the white space perfect every time.
- You can assume that the grammar files are properly formatted and that the contents of the file are always loaded without incident. The only thing you're required to detect is the situation where some expansion references an undefined nonterminal (In which case you can just quit the program by calling `assert` or `exit`.)

Some Advice

Start this weekend. You'll soon learn that the amount of code you need to write is laughably little. But Assignment 1 isn't about dense C++ coding or clever C hacking—it's about plopping you down in a new UNIX world and letting you explore. Hang out at the Terman cluster so you can ask CS107 TAs for advice.

Swing by Jerry's office hours on Monday and Wednesday mornings if you need a brief tutorial. And read the other handouts!

All that being said, you need to code as well. ☺

You also need to heed the advice that you always ignore. Compile and test often. Don't try to write everything first and compile afterwards. Instead (and this is the best advice I can give any developer, young or old) you should contrive lots of little milestones that sit along the path between start and finish. Work toward that final product by slowly evolving your code into something incrementally closer to the place you want to be.

You never want to stray too far from a working system. The safest thing to do is to perturb a working system in the direction of your goal, but making sure the perturbation is small enough that it's easily reversed if things go wrong.

Submission

There's a link on the CS107 web page that leads to another page, and this second page outlines how to submit your work. As opposed to the CS106 courses, you electronically submit your code from an **pod** (because that's where you do your final testing anyway) by running a simple script.

If you submit and later decide to submit an even better version, go ahead and submit a second (or third, or seventh) time; we'll only grade the last one. Be sure to give yourself more than enough time before the hard midnight deadline to submit. If you need to take a late day, that's cool—just take them, and submit when you're done. We'll catalogue late days behind the scenes and let you know if you're ever close to exhausting all of your free ones.

As far as grading is concerned, we'll be grading this first assignment with a simple $\sqrt{+}$ or $\sqrt{-}$. All other assignments will be graded using the traditional A+ through F scale. We recognize that the first assignment here might very well be the most frustrating, so I'm trying to minimize any UNIX-driven angst by adopting a patty cake approach to grading this first time.

Here's the short list of things I really care about (for all assignments, not just this one):

- Code unification. If you're writing the same block of code twice, then you need a fantastic reason to not factor the code into a helper routine called from both places. If you need to parameterize things a bit, then do it.
- Clear, concise function names and compact implementations. Function and methods implementations should be short enough that they either cover one algorithmic detail, or they call a series of helper functions and methods in order to accomplish some larger task.
- No global variables ever! Think of Joan Crawford screaming at her daughter. For once, Joan would be right. The implementation of any function should be framed in

terms of the arguments passed in and the values synthesized internally, and it should be able to compile no matter what code base the code get used for. A function that depends on global variables is like a laptop that **requires** a direct Ethernet connection to the Internet. Functions without global are wireless. Which would you want to use?

- Don't make unnecessary copies of large data structures, and don't make lots of little copies of small ones. Don't use dynamic memory allocation unless there's an excellent reason for doing so. And don't orphan any memory either. Programmers select C and C++ as programming languages because they want access to and control over memory. But with that control comes the responsibility to free up anything you allocate.
- Documentation of particularly dense code. Think about the type of comments you'd find most useful in six months if you were to reminisce and revisit your cs107 folder. I don't need comments for everything—I'd be a hypocrite if I asked you to document everything when I don't always myself. But when the code is complex and there's no obvious story being told, tell the story in English. Mention edge cases. Talk about pre-conditions that need to be met if the implementation is to work. Stuff like that.

There you have it. Enjoy this first assignment, and feel free to tap us as resources if you get stuck.